

Node.js v16.7.0 documentation

About this documentation

Welcome to the official API reference documentation for Node.js!

Node.js is a JavaScript runtime built on the [V8 JavaScript engine](#).

Contributing

Report errors in this documentation in [the issue tracker](#). See [the contributing guide](#) for directions on how to submit pull requests.

Stability index

Throughout the documentation are indications of a section's stability. Some APIs are so proven and so relied upon that they are unlikely to ever change at all. Others are brand new and experimental, or known to be hazardous.

The stability indices are as follows:

Stability: 0 - Deprecated. The feature may emit warnings. Backward compatibility is not guaranteed.

Stability: 1 - Experimental. The feature is not subject to [Semantic Versioning](#) rules. Non-backward compatible changes or removal may occur in any future release. Use of the feature is not recommended in production environments.

Stability: 2 - Stable. Compatibility with the npm ecosystem is a high priority.

Stability: 3 - Legacy. The feature is no longer recommended for use. While it likely will not be removed, and is still covered by semantic-versioning guarantees, use of the feature should be avoided.

Use caution when making use of Experimental features, particularly within modules. Users may not be aware that experimental features are being used. Bugs or behavior changes may surprise users when Experimental API modifications occur. To avoid surprises, use of an Experimental feature may need a command-line flag. Experimental features may also emit a [warning](#).

Stability overview

JSON output

Every `.html` document has a corresponding `.json` document. This is for IDEs and other utilities that consume the documentation.

System calls and man pages

Node.js functions which wrap a system call will document that. The docs link to the corresponding man pages which describe how the system call works.

Most Unix system calls have Windows analogues. Still, behavior differences may be unavoidable.

Usage and example

Usage

```
node [options] [V8 options] [script.js | -e "script" | - ] [arguments]
```

Please see the [Command-line options](#) document for more information.

Example

An example of a [web server](#) written with Node.js which responds with `'Hello, World!'`:

Commands in this document start with `$` or `>` to replicate how they would appear in a user's terminal. Do not include the `$` and `>` characters. They are there to show the start of each command.

Lines that don't start with `$` or `>` character show the output of the previous command.

First, make sure to have downloaded and installed Node.js. See [Installing Node.js via package manager](#) for further install information.

Now, create an empty project folder called `projects`, then navigate into it.

Linux and Mac:

```
$ mkdir ~/projects
$ cd ~/projects
```

Windows CMD:

```
> mkdir %USERPROFILE%\projects
> cd %USERPROFILE%\projects
```

Windows PowerShell:

```
> mkdir $env:USERPROFILE\projects
> cd $env:USERPROFILE\projects
```

Next, create a new source file in the `projects` folder and call it `hello-world.js`.

Open `hello-world.js` in any preferred text editor and paste in the following content:

```
const http = require('http');

const hostname = '127.0.0.1';
const port = 3000;
```



```
const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello, World!\n');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

Save the file, go back to the terminal window, and enter the following command:

```
$ node hello-world.js
```

Output like this should appear in the terminal:

```
Server running at http://127.0.0.1:3000/
```

Now, open any preferred web browser and visit `http://127.0.0.1:3000`.

If the browser displays the string `Hello, World!`, that indicates the server is working.

Assert

Stability: 2 - Stable

Source Code: [lib/assert.js](#)

The `assert` module provides a set of assertion functions for verifying invariants.

Strict assertion mode

In strict assertion mode, non-strict methods behave like their corresponding strict methods. For example, `assert.deepEqual()` will behave like `assert.deepStrictEqual()`.

In strict assertion mode, error messages for objects display a diff. In legacy assertion mode, error messages for objects display the objects, often truncated.

To use strict assertion mode:

```
import { strict as assert } from 'assert';const assert = require('assert').strict;
```

```
import assert from 'assert/strict';const assert = require('assert/strict');
```

Example error diff:

```

import { strict as assert } from 'assert';

assert.deepEqual([[1, 2, 3]], 4, 5, [[1, 2, '3']], 4, 5));
// AssertionError: Expected inputs to be strictly deep-equal:
// + actual - expected ... Lines skipped
//
//   [
//     [
//       ...
//       2,
//     +   3
//     -   '3'
//     ],
//     ...
//     5
//   ]const assert = require('assert/strict');

assert.deepEqual([[1, 2, 3]], 4, 5, [[1, 2, '3']], 4, 5));
// AssertionError: Expected inputs to be strictly deep-equal:
// + actual - expected ... Lines skipped
//
//   [
//     [
//       ...
//       2,
//     +   3
//     -   '3'
//     ],
//     ...
//     5
//   ]

```

To deactivate the colors, use the `NO_COLOR` or `NODE_DISABLE_COLORS` environment variables. This will also deactivate the colors in the REPL. For more on color support in terminal environments, read the `getColorDepth()` documentation.

Legacy assertion mode

Legacy assertion mode uses the `Abstract Equality Comparison` in:

- `assert.deepEqual()`
- `assert.equal()`
- `assert.notDeepEqual()`
- `assert.notEqual()`

To use legacy assertion mode:

```

import assert from 'assert';const assert = require('assert');

```

Whenever possible, use the `strict assertion mode` instead. Otherwise, the `Abstract Equality Comparison` may cause surprising results. This is especially true for `assert.deepEqual()`, where the comparison rules are lax:

```
// WARNING: This does not throw an AssertionError!  
assert.deepEqual(/a/gi, new Date());
```

Class: assert.AssertionError

- Extends: `<errors.Error>`

Indicates the failure of an assertion. All errors thrown by the `assert` module will be instances of the `AssertionError` class.

new assert.AssertionError(options)

- `options` `<Object>`
 - `message` `<string>` If provided, the error message is set to this value.
 - `actual` `<any>` The `actual` property on the error instance.
 - `expected` `<any>` The `expected` property on the error instance.
 - `operator` `<string>` The `operator` property on the error instance.
 - `stackStartFn` `<Function>` If provided, the generated stack trace omits frames before this function.

A subclass of `Error` that indicates the failure of an assertion.

All instances contain the built-in `Error` properties (`message` and `name`) and:

- `actual` `<any>` Set to the `actual` argument for methods such as `assert.strictEqual()`.
- `expected` `<any>` Set to the `expected` value for methods such as `assert.strictEqual()`.
- `generatedMessage` `<boolean>` Indicates if the message was auto-generated (`true`) or not.
- `code` `<string>` Value is always `ERR_ASSERTION` to show that the error is an assertion error.
- `operator` `<string>` Set to the passed in operator value.

```
import assert from 'assert';  
  
// Generate an AssertionError to compare the error message later:  
const { message } = new assert.AssertionError({  
  actual: 1,  
  expected: 2,  
  operator: 'strictEqual'  
});  
  
// Verify error output:  
try {  
  assert.strictEqual(1, 2);  
} catch (err) {  
  assert(err instanceof assert.AssertionError);  
  assert.strictEqual(err.message, message);  
  assert.strictEqual(err.name, 'AssertionError');  
  assert.strictEqual(err.actual, 1);  
  assert.strictEqual(err.expected, 2);  
  assert.strictEqual(err.code, 'ERR_ASSERTION');  
  assert.strictEqual(err.operator, 'strictEqual');  
  assert.strictEqual(err.generatedMessage, true);  
}const assert = require('assert');
```

```
// Generate an AssertionError to compare the error message later:
const { message } = new assert.AssertionError({
  actual: 1,
  expected: 2,
  operator: 'strictEqual'
});

// Verify error output:
try {
  assert.strictEqual(1, 2);
} catch (err) {
  assert(err instanceof assert.AssertionError);
  assert.strictEqual(err.message, message);
  assert.strictEqual(err.name, 'AssertionError');
  assert.strictEqual(err.actual, 1);
  assert.strictEqual(err.expected, 2);
  assert.strictEqual(err.code, 'ERR_ASSERTION');
  assert.strictEqual(err.operator, 'strictEqual');
  assert.strictEqual(err.generatedMessage, true);
}
```

Class: `assert.CallTracker`

Stability: 1 - Experimental

This feature is currently experimental and behavior might still change.

`new assert.CallTracker()`

Creates a new `CallTracker` object which can be used to track if functions were called a specific number of times. The `tracker.verify()` must be called for the verification to take place. The usual pattern would be to call it in a `process.on('exit')` handler.

```
import assert from 'assert';
import process from 'process';

const tracker = new assert.CallTracker();

function func() {}

// callfunc() must be called exactly 1 time before tracker.verify().
const callfunc = tracker.calls(func, 1);

callfunc();

// Calls tracker.verify() and verifies if all tracker.calls() functions have
// been called exact times.
process.on('exit', () => {
  tracker.verify();
});
```

```
});const assert = require('assert');

const tracker = new assert.CallTracker();

function func() {}

// callfunc() must be called exactly 1 time before tracker.verify().
const callfunc = tracker.calls(func, 1);

callfunc();

// Calls tracker.verify() and verifies if all tracker.calls() functions have
// been called exact times.
process.on('exit', () => {
  tracker.verify();
});
```

tracker.calls([fn][, exact])

- **fn** `<Function>` **Default:** A no-op function.
- **exact** `<number>` **Default:** 1.
- Returns: `<Function>` that wraps **fn**.

The wrapper function is expected to be called exactly **exact** times. If the function has not been called exactly **exact** times when **tracker.verify()** is called, then **tracker.verify()** will throw an error.

```
import assert from 'assert';

// Creates call tracker.
const tracker = new assert.CallTracker();

function func() {}

// Returns a function that wraps func() that must be called exact times
// before tracker.verify().
const callfunc = tracker.calls(func);const assert = require('assert');

// Creates call tracker.
const tracker = new assert.CallTracker();

function func() {}

// Returns a function that wraps func() that must be called exact times
// before tracker.verify().
const callfunc = tracker.calls(func);
```

tracker.report()

- Returns: `<Array>` of objects containing information about the wrapper functions returned by **tracker.calls()**.
- Object `<Object>`

- `message` `<string>`
- `actual` `<number>` The actual number of times the function was called.
- `expected` `<number>` The number of times the function was expected to be called.
- `operator` `<string>` The name of the function that is wrapped.
- `stack` `<Object>` A stack trace of the function.

The arrays contains information about the expected and actual number of calls of the functions that have not been called the expected number of times.

```
import assert from 'assert';

// Creates call tracker.
const tracker = new assert.CallTracker();

function func() {}

function foo() {}

// Returns a function that wraps func() that must be called exact times
// before tracker.verify().
const callfunc = tracker.calls(func, 2);

// Returns an array containing information on callfunc()
tracker.report();
// [
//   {
//     message: 'Expected the func function to be executed 2 time(s) but was
//     executed 0 time(s).',
//     actual: 0,
//     expected: 2,
//     operator: 'func',
//     stack: stack trace
//   }
// ]const assert = require('assert');

// Creates call tracker.
const tracker = new assert.CallTracker();

function func() {}

function foo() {}

// Returns a function that wraps func() that must be called exact times
// before tracker.verify().
const callfunc = tracker.calls(func, 2);

// Returns an array containing information on callfunc()
tracker.report();
// [
//   {
```

```
//   message: 'Expected the func function to be executed 2 time(s) but was
//   executed 0 time(s).',
//   actual: 0,
//   expected: 2,
//   operator: 'func',
//   stack: stack trace
// }
// ]
```

tracker.verify()

Iterates through the list of functions passed to `tracker.calls()` and will throw an error for functions that have not been called the expected number of times.

```
import assert from 'assert';

// Creates call tracker.
const tracker = new assert.CallTracker();

function func() {}

// Returns a function that wraps func() that must be called exact times
// before tracker.verify().
const callfunc = tracker.calls(func, 2);

callfunc();

// Will throw an error since callfunc() was only called once.
tracker.verify();const assert = require('assert');

// Creates call tracker.
const tracker = new assert.CallTracker();

function func() {}

// Returns a function that wraps func() that must be called exact times
// before tracker.verify().
const callfunc = tracker.calls(func, 2);

callfunc();

// Will throw an error since callfunc() was only called once.
tracker.verify();
```

assert(value[, message])

- `value` `<any>` The input that is checked for being truthy.
- `message` `<string> | <Error>`

An alias of `assert.ok()`.

assert.deepEqual(actual, expected[, message])

- `actual` `<any>`
- `expected` `<any>`
- `message` `<string> | <Error>`

Strict assertion mode

An alias of `assert.deepStrictEqual()`.

Legacy assertion mode

Stability: 3 - Legacy: Use `assert.deepStrictEqual()` instead.

Tests for deep equality between the `actual` and `expected` parameters. Consider using `assert.deepStrictEqual()` instead. `assert.deepEqual()` can have surprising results.

Deep equality means that the enumerable "own" properties of child objects are also recursively evaluated by the following rules.

Comparison details

- Primitive values are compared with the `Abstract Equality Comparison` (`==`) with the exception of `NaN`. It is treated as being identical in case both sides are `NaN`.
- `Type tags` of objects should be the same.
- Only `enumerable "own" properties` are considered.
- `Error` names and messages are always compared, even if these are not enumerable properties.
- `Object wrappers` are compared both as objects and unwrapped values.
- `Object` properties are compared unordered.
- `Map` keys and `Set` items are compared unordered.
- Recursion stops when both sides differ or both sides encounter a circular reference.
- Implementation does not test the `[[Prototype]]` of objects.
- `Symbol` properties are not compared.
- `WeakMap` and `WeakSet` comparison does not rely on their values.

The following example does not throw an `AssertionError` because the primitives are considered equal by the `Abstract Equality Comparison` (`==`).

```
import assert from 'assert';
// WARNING: This does not throw an AssertionError!

assert.deepEqual('+00000000', false);const assert = require('assert');
// WARNING: This does not throw an AssertionError!

assert.deepEqual('+00000000', false);
```

"Deep" equality means that the enumerable "own" properties of child objects are evaluated also:


```

import assert from 'assert';

const obj1 = {
  a: {
    b: 1
  }
};
const obj2 = {
  a: {
    b: 2
  }
};
const obj3 = {
  a: {
    b: 1
  }
};
const obj4 = Object.create(obj1);

assert.deepEqual(obj1, obj1);
// OK

// Values of b are different:
assert.deepEqual(obj1, obj2);
// AssertionError: { a: { b: 1 } } deepEqual { a: { b: 2 } }

assert.deepEqual(obj1, obj3);
// OK

// Prototypes are ignored:
assert.deepEqual(obj1, obj4);
// AssertionError: { a: { b: 1 } } deepEqual {}const assert = require('assert');

const obj1 = {
  a: {
    b: 1
  }
};
const obj2 = {
  a: {
    b: 2
  }
};
const obj3 = {
  a: {
    b: 1
  }
};
const obj4 = Object.create(obj1);

```

```

assert.deepEqual(obj1, obj1);
// OK

// Values of b are different:
assert.deepEqual(obj1, obj2);
// AssertionError: { a: { b: 1 } } deepEqual { a: { b: 2 } }

assert.deepEqual(obj1, obj3);
// OK

// Prototypes are ignored:
assert.deepEqual(obj1, obj4);
// AssertionError: { a: { b: 1 } } deepEqual {}

```

If the values are not equal, an `AssertionError` is thrown with a `message` property set equal to the value of the `message` parameter. If the `message` parameter is undefined, a default error message is assigned. If the `message` parameter is an instance of an `Error` then it will be thrown instead of the `AssertionError`.

assert.deepEqual(actual, expected[, message])

- `actual` `<any>`
- `expected` `<any>`
- `message` `<string> | <Error>`

Tests for deep equality between the `actual` and `expected` parameters. "Deep" equality means that the enumerable "own" properties of child objects are recursively evaluated also by the following rules.

Comparison details

- Primitive values are compared using the `SameValue Comparison`, used by `Object.is()`.
- `Type tags` of objects should be the same.
- `[[Prototype]]` of objects are compared using the `Strict Equality Comparison`.
- Only `enumerable "own" properties` are considered.
- `Error` names and messages are always compared, even if these are not enumerable properties.
- Enumerable own `Symbol` properties are compared as well.
- `Object wrappers` are compared both as objects and unwrapped values.
- `Object` properties are compared unordered.
- `Map` keys and `Set` items are compared unordered.
- Recursion stops when both sides differ or both sides encounter a circular reference.
- `WeakMap` and `WeakSet` comparison does not rely on their values. See below for further details.

```

import assert from 'assert/strict';

// This fails because 1 !== '1'.
deepStrictEqual({ a: 1 }, { a: '1' });
// AssertionError: Expected inputs to be strictly deep-equal:
// + actual - expected
//
// {

```

```

// +   a: 1
// -   a: '1'
//   }

// The following objects don't have own properties
const date = new Date();
const object = {};
const fakeDate = {};
Object.setPrototypeOf(fakeDate, Date.prototype);

// Different [[Prototype]]:
assert.deepStrictEqual(object, fakeDate);
// AssertionError: Expected inputs to be strictly deep-equal:
// + actual - expected
//
// + {}
// - Date {}

// Different type tags:
assert.deepStrictEqual(date, fakeDate);
// AssertionError: Expected inputs to be strictly deep-equal:
// + actual - expected
//
// + 2018-04-26T00:49:08.604Z
// - Date {}

assert.deepStrictEqual(NaN, NaN);
// OK, because of the SameValue comparison

// Different unwrapped numbers:
assert.deepStrictEqual(new Number(1), new Number(2));
// AssertionError: Expected inputs to be strictly deep-equal:
// + actual - expected
//
// + [Number: 1]
// - [Number: 2]

assert.deepStrictEqual(new String('foo'), Object('foo'));
// OK because the object and the string are identical when unwrapped.

assert.deepStrictEqual(-0, -0);
// OK

// Different zeros using the SameValue Comparison:
assert.deepStrictEqual(0, -0);
// AssertionError: Expected inputs to be strictly deep-equal:
// + actual - expected
//
// + 0
// - -0

```

```

const symbol1 = Symbol();
const symbol2 = Symbol();
assert.deepStrictEqual({ [symbol1]: 1 }, { [symbol1]: 1 });
// OK, because it is the same symbol on both objects.

assert.deepStrictEqual({ [symbol1]: 1 }, { [symbol2]: 1 });
// AssertionError [ERR_ASSERTION]: Inputs identical but not reference equal:
//
// {
//   [Symbol()]: 1
// }

const weakMap1 = new WeakMap();
const weakMap2 = new WeakMap([[{}], {}]);
const weakMap3 = new WeakMap();
weakMap3.unequal = true;

assert.deepStrictEqual(weakMap1, weakMap2);
// OK, because it is impossible to compare the entries

// Fails because weakMap3 has a property that weakMap1 does not contain:
assert.deepStrictEqual(weakMap1, weakMap3);
// AssertionError: Expected inputs to be strictly deep-equal:
// + actual - expected
//
//   WeakMap {
// +   [items unknown]
// -   [items unknown],
// -   unequal: true
//   }const assert = require('assert/strict');

// This fails because 1 !== '1'.
assert.deepStrictEqual({ a: 1 }, { a: '1' });
// AssertionError: Expected inputs to be strictly deep-equal:
// + actual - expected
//
//   {
// +   a: 1
// -   a: '1'
//   }

// The following objects don't have own properties
const date = new Date();
const object = {};
const fakeDate = {};
Object.setPrototypeOf(fakeDate, Date.prototype);

// Different [[Prototype]]:
assert.deepStrictEqual(object, fakeDate);
// AssertionError: Expected inputs to be strictly deep-equal:
// + actual - expected

```

```

//
// + {}
// - Date {}

// Different type tags:
assert.deepStrictEqual(date, fakeDate);
// AssertionError: Expected inputs to be strictly deep-equal:
// + actual - expected
//
// + 2018-04-26T00:49:08.604Z
// - Date {}

assert.deepStrictEqual(NaN, NaN);
// OK, because of the SameValue comparison

// Different unwrapped numbers:
assert.deepStrictEqual(new Number(1), new Number(2));
// AssertionError: Expected inputs to be strictly deep-equal:
// + actual - expected
//
// + [Number: 1]
// - [Number: 2]

assert.deepStrictEqual(new String('foo'), Object('foo'));
// OK because the object and the string are identical when unwrapped.

assert.deepStrictEqual(-0, -0);
// OK

// Different zeros using the SameValue Comparison:
assert.deepStrictEqual(0, -0);
// AssertionError: Expected inputs to be strictly deep-equal:
// + actual - expected
//
// + 0
// - -0

const symbol1 = Symbol();
const symbol2 = Symbol();
assert.deepStrictEqual({ [symbol1]: 1 }, { [symbol1]: 1 });
// OK, because it is the same symbol on both objects.

assert.deepStrictEqual({ [symbol1]: 1 }, { [symbol2]: 1 });
// AssertionError [ERR_ASSERTION]: Inputs identical but not reference equal:
//
// {
//   [Symbol()]: 1
// }

const weakMap1 = new WeakMap();
const weakMap2 = new WeakMap([[{}], {}]);

```

```

const weakMap3 = new WeakMap();
weakMap3.unequal = true;

assert.deepStrictEqual(weakMap1, weakMap2);
// OK, because it is impossible to compare the entries

// Fails because weakMap3 has a property that weakMap1 does not contain:
assert.deepStrictEqual(weakMap1, weakMap3);
// AssertionError: Expected inputs to be strictly deep-equal:
// + actual - expected
//
//   WeakMap {
// +   [items unknown]
// -   [items unknown],
// -   unequal: true
//   }

```

If the values are not equal, an `AssertionError` is thrown with a `message` property set equal to the value of the `message` parameter. If the `message` parameter is undefined, a default error message is assigned. If the `message` parameter is an instance of an `Error` then it will be thrown instead of the `AssertionError`.

assert.doesNotMatch(string, regexp[, message])

- `string` `<string>`
- `regexp` `<RegExp>`
- `message` `<string> | <Error>`

Expects the `string` input not to match the regular expression.

```

import assert from 'assert/strict';

assert.doesNotMatch('I will fail', /fail/);
// AssertionError [ERR_ASSERTION]: The input was expected to not match the ...

assert.doesNotMatch(123, /pass/);
// AssertionError [ERR_ASSERTION]: The "string" argument must be of type string.

assert.doesNotMatch('I will pass', /different/);
// OKconst assert = require('assert/strict');

assert.doesNotMatch('I will fail', /fail/);
// AssertionError [ERR_ASSERTION]: The input was expected to not match the ...

assert.doesNotMatch(123, /pass/);
// AssertionError [ERR_ASSERTION]: The "string" argument must be of type string.

assert.doesNotMatch('I will pass', /different/);
// OK

```

If the values do match, or if the `string` argument is of another type than `string`, an `AssertionError` is thrown with a `message` property set equal to the value of the `message` parameter. If the `message` parameter is undefined, a default error message is assigned. If the `message` parameter is an instance of an `Error` then it will be thrown instead of the `AssertionError`.

`assert.doesNotReject(asyncFn[, error][, message])`

- `asyncFn` `<Function>` | `<Promise>`
- `error` `<RegExp>` | `<Function>`
- `message` `<string>`

Awaits the `asyncFn` promise or, if `asyncFn` is a function, immediately calls the function and awaits the returned promise to complete. It will then check that the promise is not rejected.

If `asyncFn` is a function and it throws an error synchronously, `assert.doesNotReject()` will return a rejected `Promise` with that error. If the function does not return a promise, `assert.doesNotReject()` will return a rejected `Promise` with an `ERR_INVALID_RETURN_VALUE` error. In both cases the error handler is skipped.

Using `assert.doesNotReject()` is actually not useful because there is little benefit in catching a rejection and then rejecting it again. Instead, consider adding a comment next to the specific code path that should not reject and keep error messages as expressive as possible.

If specified, `error` can be a `Class`, `RegExp` or a validation function. See `assert.throws()` for more details.

Besides the async nature to await the completion behaves identically to `assert.doesNotThrow()`.

```
import assert from 'assert/strict';

await assert.doesNotReject(
  async () => {
    throw new TypeError('Wrong value');
  },
  SyntaxError
);const assert = require('assert/strict');

(async () => {
  await assert.doesNotReject(
    async () => {
      throw new TypeError('Wrong value');
    },
    SyntaxError
  );
})();
```

```
import assert from 'assert/strict';

assert.doesNotReject(Promise.reject(new TypeError('Wrong value')))
  .then(() => {
    // ...
  });
```

```
const assert = require('assert/strict');

assert.doesNotReject(Promise.reject(new TypeError('Wrong value')))
  .then(() => {
    // ...
  });
```

assert.doesNotThrow(fn[, error][, message])

- `fn` `<Function>`
- `error` `<RegExp> | <Function>`
- `message` `<string>`

Asserts that the function `fn` does not throw an error.

Using `assert.doesNotThrow()` is actually not useful because there is no benefit in catching an error and then rethrowing it. Instead, consider adding a comment next to the specific code path that should not throw and keep error messages as expressive as possible.

When `assert.doesNotThrow()` is called, it will immediately call the `fn` function.

If an error is thrown and it is the same type as that specified by the `error` parameter, then an `AssertionError` is thrown. If the error is of a different type, or if the `error` parameter is undefined, the error is propagated back to the caller.

If specified, `error` can be a `Class`, `RegExp` or a validation function. See `assert.throws()` for more details.

The following, for instance, will throw the `TypeError` because there is no matching error type in the assertion:

```
import assert from 'assert/strict';

assert.doesNotThrow(
  () => {
    throw new TypeError('Wrong value');
  },
  SyntaxError
);
```

```
const assert = require('assert/strict');

assert.doesNotThrow(
  () => {
    throw new TypeError('Wrong value');
  },
  SyntaxError
);
```

However, the following will result in an `AssertionError` with the message 'Got unwanted exception...':

```
import assert from 'assert/strict';
```



```
assert.doesNotThrow(
  () => {
    throw new TypeError('Wrong value');
  },
  TypeError
);
```

```
const assert = require('assert/strict');

assert.doesNotThrow(
  () => {
    throw new TypeError('Wrong value');
  },
  TypeError
);
```

If an `AssertionError` is thrown and a value is provided for the `message` parameter, the value of `message` will be appended to the `AssertionError` message:

```
import assert from 'assert/strict';

assert.doesNotThrow(
  () => {
    throw new TypeError('Wrong value');
  },
  /Wrong value/,
  'Whoops'
);
// Throws: AssertionError: Got unwanted exception: Whoops
```

```
const assert = require('assert/strict');

assert.doesNotThrow(
  () => {
    throw new TypeError('Wrong value');
  },
  /Wrong value/,
  'Whoops'
);
// Throws: AssertionError: Got unwanted exception: Whoops
```

assert.equal(actual, expected[, message])

- `actual` `<any>`
- `expected` `<any>`
- `message` `<string> | <Error>`

Strict assertion mode

An alias of `assert.strictEqual()`.

Legacy assertion mode

Stability: 3 - Legacy: Use `assert.strictEqual()` instead.

Tests shallow, coercive equality between the `actual` and `expected` parameters using the `Abstract Equality Comparison` (`==`). `NaN` is special handled and treated as being identical in case both sides are `NaN`.

```
import assert from 'assert';

assert.equal(1, 1);
// OK, 1 == 1
assert.equal(1, '1');
// OK, 1 == '1'
assert.equal(NaN, NaN);
// OK

assert.equal(1, 2);
// AssertionError: 1 == 2
assert.equal({ a: { b: 1 } }, { a: { b: 1 } });
// AssertionError: { a: { b: 1 } } == { a: { b: 1 } }const assert = require('assert');

assert.equal(1, 1);
// OK, 1 == 1
assert.equal(1, '1');
// OK, 1 == '1'
assert.equal(NaN, NaN);
// OK

assert.equal(1, 2);
// AssertionError: 1 == 2
assert.equal({ a: { b: 1 } }, { a: { b: 1 } });
// AssertionError: { a: { b: 1 } } == { a: { b: 1 } }
```

If the values are not equal, an `AssertionError` is thrown with a `message` property set equal to the value of the `message` parameter. If the `message` parameter is undefined, a default error message is assigned. If the `message` parameter is an instance of an `Error` then it will be thrown instead of the `AssertionError`.

`assert.fail([message])`

- `message` `<string>` | `<Error>` Default: 'Failed'

Throws an `AssertionError` with the provided error message or a default error message. If the `message` parameter is an instance of an `Error` then it will be thrown instead of the `AssertionError`.

```
import assert from 'assert/strict';

assert.fail();
```

```
// AssertionError [ERR_ASSERTION]: Failed

assert.fail('boom');
// AssertionError [ERR_ASSERTION]: boom

assert.fail(new TypeError('need array'));
// TypeError: need arrayconst assert = require('assert/strict');

assert.fail();
// AssertionError [ERR_ASSERTION]: Failed

assert.fail('boom');
// AssertionError [ERR_ASSERTION]: boom

assert.fail(new TypeError('need array'));
// TypeError: need array
```

Using `assert.fail()` with more than two arguments is possible but deprecated. See below for further details.

`assert.fail(actual, expected[, message[, operator[, stackStartFn]])`

Stability: 0 - Deprecated: Use `assert.fail([message])` or other assert functions instead.

- `actual` `<any>`
- `expected` `<any>`
- `message` `<string> | <Error>`
- `operator` `<string>` **Default:** `'!=='`
- `stackStartFn` `<Function>` **Default:** `assert.fail`

If `message` is falsy, the error message is set as the values of `actual` and `expected` separated by the provided `operator`. If just the two `actual` and `expected` arguments are provided, `operator` will default to `'!=='`. If `message` is provided as third argument it will be used as the error message and the other arguments will be stored as properties on the thrown object. If `stackStartFn` is provided, all stack frames above that function will be removed from `stacktrace` (see `Error.captureStackTrace`). If no arguments are given, the default message `Failed` will be used.

```
import assert from 'assert/strict';

assert.fail('a', 'b');
// AssertionError [ERR_ASSERTION]: 'a' !== 'b'

assert.fail(1, 2, undefined, '>');
// AssertionError [ERR_ASSERTION]: 1 > 2

assert.fail(1, 2, 'fail');
// AssertionError [ERR_ASSERTION]: fail

assert.fail(1, 2, 'whoops', '>');
// AssertionError [ERR_ASSERTION]: whoops
```

```

assert.fail(1, 2, new TypeError('need array'));
// TypeError: need arrayconst assert = require('assert/strict');

assert.fail('a', 'b');
// AssertionError [ERR_ASSERTION]: 'a' != 'b'

assert.fail(1, 2, undefined, '>');
// AssertionError [ERR_ASSERTION]: 1 > 2

assert.fail(1, 2, 'fail');
// AssertionError [ERR_ASSERTION]: fail

assert.fail(1, 2, 'whoops', '>');
// AssertionError [ERR_ASSERTION]: whoops

assert.fail(1, 2, new TypeError('need array'));
// TypeError: need array

```

In the last three cases `actual`, `expected`, and `operator` have no influence on the error message.

Example use of `stackStartFn` for truncating the exception's stacktrace:

```

import assert from 'assert/strict';

function suppressFrame() {
  assert.fail('a', 'b', undefined, '!==', suppressFrame);
}
suppressFrame();
// AssertionError [ERR_ASSERTION]: 'a' !== 'b'
//   at repl:1:1
//   at ContextifyScript.Script.runInThisContext (vm.js:44:33)
//   ...const assert = require('assert/strict');

function suppressFrame() {
  assert.fail('a', 'b', undefined, '!==', suppressFrame);
}
suppressFrame();
// AssertionError [ERR_ASSERTION]: 'a' !== 'b'
//   at repl:1:1
//   at ContextifyScript.Script.runInThisContext (vm.js:44:33)
//   ...

```

assert.ifError(value)

- `value` `<any>`

Throws `value` if `value` is not `undefined` or `null`. This is useful when testing the `error` argument in callbacks. The stack trace contains all frames from the error passed to `ifError()` including the potential new frames for `ifError()` itself.

```

import assert from 'assert/strict';

assert.ifError(null);
// OK
assert.ifError(0);
// AssertionError [ERR_ASSERTION]: ifError got unwanted exception: 0
assert.ifError('error');
// AssertionError [ERR_ASSERTION]: ifError got unwanted exception: 'error'
assert.ifError(new Error());
// AssertionError [ERR_ASSERTION]: ifError got unwanted exception: Error

// Create some random error frames.
let err;
(function errorFrame() {
  err = new Error('test error');
})();

(function ifErrorFrame() {
  assert.ifError(err);
})();
// AssertionError [ERR_ASSERTION]: ifError got unwanted exception: test error
//      at ifErrorFrame
//      at errorFrame
const assert = require('assert/strict');

assert.ifError(null);
// OK
assert.ifError(0);
// AssertionError [ERR_ASSERTION]: ifError got unwanted exception: 0
assert.ifError('error');
// AssertionError [ERR_ASSERTION]: ifError got unwanted exception: 'error'
assert.ifError(new Error());
// AssertionError [ERR_ASSERTION]: ifError got unwanted exception: Error

// Create some random error frames.
let err;
(function errorFrame() {
  err = new Error('test error');
})();

(function ifErrorFrame() {
  assert.ifError(err);
})();
// AssertionError [ERR_ASSERTION]: ifError got unwanted exception: test error
//      at ifErrorFrame
//      at errorFrame

```

assert.match(string, regexp[, message])

- `string` `<string>`

- `regexp` `<RegExp>`
- `message` `<string> | <Error>`

Expects the `string` input to match the regular expression.

```
import assert from 'assert/strict';

assert.match('I will fail', /pass/);
// AssertionError [ERR_ASSERTION]: The input did not match the regular ...

assert.match(123, /pass/);
// AssertionError [ERR_ASSERTION]: The "string" argument must be of type string.

assert.match('I will pass', /pass/);
// OKconst assert = require('assert/strict');

assert.match('I will fail', /pass/);
// AssertionError [ERR_ASSERTION]: The input did not match the regular ...

assert.match(123, /pass/);
// AssertionError [ERR_ASSERTION]: The "string" argument must be of type string.

assert.match('I will pass', /pass/);
// OK
```

If the values do not match, or if the `string` argument is of another type than `string`, an `AssertionError` is thrown with a `message` property set equal to the value of the `message` parameter. If the `message` parameter is undefined, a default error message is assigned. If the `message` parameter is an instance of an `Error` then it will be thrown instead of the `AssertionError`.

assert.notDeepEqual(actual, expected[, message])

- `actual` `<any>`
- `expected` `<any>`
- `message` `<string> | <Error>`

Strict assertion mode

An alias of `assert.notDeepStrictEqual()`.

Legacy assertion mode

Stability: 3 - Legacy: Use `assert.notDeepStrictEqual()` instead.

Tests for any deep inequality. Opposite of `assert.deepEqual()`.

```
import assert from 'assert';

const obj1 = {
  a: {
    b: 1
```

```

    }
  };
  const obj2 = {
    a: {
      b: 2
    }
  };
  const obj3 = {
    a: {
      b: 1
    }
  };
  const obj4 = Object.create(obj1);

  assert.notDeepEqual(obj1, obj1);
  // AssertionError: { a: { b: 1 } } notDeepEqual { a: { b: 1 } }

  assert.notDeepEqual(obj1, obj2);
  // OK

  assert.notDeepEqual(obj1, obj3);
  // AssertionError: { a: { b: 1 } } notDeepEqual { a: { b: 1 } }

  assert.notDeepEqual(obj1, obj4);
  // OKconst assert = require('assert');

  const obj1 = {
    a: {
      b: 1
    }
  };
  const obj2 = {
    a: {
      b: 2
    }
  };
  const obj3 = {
    a: {
      b: 1
    }
  };
  const obj4 = Object.create(obj1);

  assert.notDeepEqual(obj1, obj1);
  // AssertionError: { a: { b: 1 } } notDeepEqual { a: { b: 1 } }

  assert.notDeepEqual(obj1, obj2);
  // OK

  assert.notDeepEqual(obj1, obj3);
  // AssertionError: { a: { b: 1 } } notDeepEqual { a: { b: 1 } }

```

```
assert.notDeepEqual(obj1, obj4);  
// OK
```

If the values are deeply equal, an `AssertionError` is thrown with a `message` property set equal to the value of the `message` parameter. If the `message` parameter is undefined, a default error message is assigned. If the `message` parameter is an instance of an `Error` then it will be thrown instead of the `AssertionError`.

`assert.notDeepStrictEqual(actual, expected[, message])`

- `actual` `<any>`
- `expected` `<any>`
- `message` `<string> | <Error>`

Tests for deep strict inequality. Opposite of `assert.deepStrictEqual()`.

```
import assert from 'assert/strict';  
  
assert.notDeepStrictEqual({ a: 1 }, { a: '1' });  
// OK  
const assert = require('assert/strict');  
  
assert.notDeepStrictEqual({ a: 1 }, { a: '1' });  
// OK
```

If the values are deeply and strictly equal, an `AssertionError` is thrown with a `message` property set equal to the value of the `message` parameter. If the `message` parameter is undefined, a default error message is assigned. If the `message` parameter is an instance of an `Error` then it will be thrown instead of the `AssertionError`.

`assert.notEqual(actual, expected[, message])`

- `actual` `<any>`
- `expected` `<any>`
- `message` `<string> | <Error>`

Strict assertion mode

An alias of `assert.notStrictEqual()`.

Legacy assertion mode

Stability: 3 - Legacy: Use `assert.notStrictEqual()` instead.

Tests shallow, coercive inequality with the `Abstract Equality Comparison` (`!=`). `NaN` is special handled and treated as being identical in case both sides are `NaN`.

```
import assert from 'assert';  
  
assert.notEqual(1, 2);  
// OK
```



```

assert.notEqual(1, 1);
// AssertionError: 1 != 1

assert.notEqual(1, '1');
// AssertionError: 1 != '1'const assert = require('assert');

assert.notEqual(1, 2);
// OK

assert.notEqual(1, 1);
// AssertionError: 1 != 1

assert.notEqual(1, '1');
// AssertionError: 1 != '1'

```

If the values are equal, an `AssertionError` is thrown with a `message` property set equal to the value of the `message` parameter. If the `message` parameter is undefined, a default error message is assigned. If the `message` parameter is an instance of an `Error` then it will be thrown instead of the `AssertionError`.

assert.notStrictEqual(actual, expected[, message])

- `actual` `<any>`
- `expected` `<any>`
- `message` `<string> | <Error>`

Tests strict inequality between the `actual` and `expected` parameters as determined by the [SameValue Comparison](#).

```

import assert from 'assert/strict';

assert.notStrictEqual(1, 2);
// OK

assert.notStrictEqual(1, 1);
// AssertionError [ERR_ASSERTION]: Expected "actual" to be strictly unequal to:
//
// 1

assert.notStrictEqual(1, '1');
// OKconst assert = require('assert/strict');

assert.notStrictEqual(1, 2);
// OK

assert.notStrictEqual(1, 1);
// AssertionError [ERR_ASSERTION]: Expected "actual" to be strictly unequal to:
//
// 1

```

```
assert.notStrictEqual(1, '1');  
// OK
```

If the values are strictly equal, an `AssertionError` is thrown with a `message` property set equal to the value of the `message` parameter. If the `message` parameter is undefined, a default error message is assigned. If the `message` parameter is an instance of an `Error` then it will be thrown instead of the `AssertionError`.

assert.ok(value[, message])

- `value` `<any>`
- `message` `<string> | <Error>`

Tests if `value` is truthy. It is equivalent to `assert.equal(!!value, true, message)`.

If `value` is not truthy, an `AssertionError` is thrown with a `message` property set equal to the value of the `message` parameter. If the `message` parameter is undefined, a default error message is assigned. If the `message` parameter is an instance of an `Error` then it will be thrown instead of the `AssertionError`. If no arguments are passed in at all `message` will be set to the string: `'No value argument passed to `assert.ok()`'`.

Be aware that in the `repl` the error message will be different to the one thrown in a file! See below for further details.

```
import assert from 'assert/strict';  
  
assert.ok(true);  
// OK  
assert.ok(1);  
// OK  
  
assert.ok();  
// AssertionError: No value argument passed to `assert.ok()`  
  
assert.ok(false, 'it\'s false');  
// AssertionError: it's false  
  
// In the repl:  
assert.ok(typeof 123 === 'string');  
// AssertionError: false == true  
  
// In a file (e.g. test.js):  
assert.ok(typeof 123 === 'string');  
// AssertionError: The expression evaluated to a falsy value:  
//  
//   assert.ok(typeof 123 === 'string')  
  
assert.ok(false);  
// AssertionError: The expression evaluated to a falsy value:  
//  
//   assert.ok(false)  
  
assert.ok(0);  
// AssertionError: The expression evaluated to a falsy value:
```

```
//
//  assert.ok(0)const assert = require('assert/strict');

assert.ok(true);
// OK
assert.ok(1);
// OK

assert.ok();
// AssertionError: No value argument passed to `assert.ok()`

assert.ok(false, 'it\'s false');
// AssertionError: it's false

// In the repl:
assert.ok(typeof 123 === 'string');
// AssertionError: false == true

// In a file (e.g. test.js):
assert.ok(typeof 123 === 'string');
// AssertionError: The expression evaluated to a falsy value:
//
//  assert.ok(typeof 123 === 'string')

assert.ok(false);
// AssertionError: The expression evaluated to a falsy value:
//
//  assert.ok(false)

assert.ok(0);
// AssertionError: The expression evaluated to a falsy value:
//
//  assert.ok(0)
```

```
import assert from 'assert/strict';

// Using `assert()` works the same:
assert(0);
// AssertionError: The expression evaluated to a falsy value:
//
//  assert(0)const assert = require('assert');

// Using `assert()` works the same:
assert(0);
// AssertionError: The expression evaluated to a falsy value:
//
//  assert(0)
```

assert.rejects(asyncFn[, error][, message])

- `asyncFn` `<Function> | <Promise>`
- `error` `<RegExp> | <Function> | <Object> | <Error>`
- `message` `<string>`

Awaits the `asyncFn` promise or, if `asyncFn` is a function, immediately calls the function and awaits the returned promise to complete. It will then check that the promise is rejected.

If `asyncFn` is a function and it throws an error synchronously, `assert.rejects()` will return a rejected `Promise` with that error. If the function does not return a promise, `assert.rejects()` will return a rejected `Promise` with an `ERR_INVALID_RETURN_VALUE` error. In both cases the error handler is skipped.

Besides the async nature to await the completion behaves identically to `assert.throws()`.

If specified, `error` can be a `Class`, `RegExp`, a validation function, an object where each property will be tested for, or an instance of error where each property will be tested for including the non-enumerable `message` and `name` properties.

If specified, `message` will be the message provided by the `AssertionError` if the `asyncFn` fails to reject.

```
import assert from 'assert/strict';

await assert.rejects(
  async () => {
    throw new TypeError('Wrong value');
  },
  {
    name: 'TypeError',
    message: 'Wrong value'
  }
);const assert = require('assert/strict');

(async () => {
  await assert.rejects(
    async () => {
      throw new TypeError('Wrong value');
    },
    {
      name: 'TypeError',
      message: 'Wrong value'
    }
  );
})();
```

```
import assert from 'assert/strict';

await assert.rejects(
  async () => {
    throw new TypeError('Wrong value');
  },
  (err) => {
    assert.strictEqual(err.name, 'TypeError');
    assert.strictEqual(err.message, 'Wrong value');
  }
);
```

```

    return true;
  }
};const assert = require('assert/strict');

(async () => {
  await assert.rejects(
    async () => {
      throw new TypeError('Wrong value');
    },
    (err) => {
      assert.strictEqual(err.name, 'TypeError');
      assert.strictEqual(err.message, 'Wrong value');
      return true;
    }
  );
})();

```

```

import assert from 'assert/strict';

assert.rejects(
  Promise.reject(new Error('Wrong value')),
  Error
).then(() => {
  // ...
});const assert = require('assert/strict');

assert.rejects(
  Promise.reject(new Error('Wrong value')),
  Error
).then(() => {
  // ...
});

```

`error` cannot be a string. If a string is provided as the second argument, then `error` is assumed to be omitted and the string will be used for `message` instead. This can lead to easy-to-miss mistakes. Please read the example in [assert.throws\(\)](#) carefully if using a string as the second argument gets considered.

assert.strictEqual(actual, expected[, message])

- `actual` `<any>`
- `expected` `<any>`
- `message` `<string> | <Error>`

Tests strict equality between the `actual` and `expected` parameters as determined by the [SameValue Comparison](#).

```

import assert from 'assert/strict';

assert.strictEqual(1, 2);
// AssertionError [ERR_ASSERTION]: Expected inputs to be strictly equal:

```

```

//
// 1 !== 2

assert.strictEqual(1, 1);
// OK

assert.strictEqual('Hello foobar', 'Hello World!');
// AssertionError [ERR_ASSERTION]: Expected inputs to be strictly equal:
// + actual - expected
//
// + 'Hello foobar'
// - 'Hello World!'
//      ^

const apples = 1;
const oranges = 2;
assert.strictEqual(apples, oranges, `apples ${apples} !== oranges ${oranges}`);
// AssertionError [ERR_ASSERTION]: apples 1 !== oranges 2

assert.strictEqual(1, '1', new TypeError('Inputs are not identical'));
// TypeError: Inputs are not identicalconst assert = require('assert/strict');

assert.strictEqual(1, 2);
// AssertionError [ERR_ASSERTION]: Expected inputs to be strictly equal:
//
// 1 !== 2

assert.strictEqual(1, 1);
// OK

assert.strictEqual('Hello foobar', 'Hello World!');
// AssertionError [ERR_ASSERTION]: Expected inputs to be strictly equal:
// + actual - expected
//
// + 'Hello foobar'
// - 'Hello World!'
//      ^

const apples = 1;
const oranges = 2;
assert.strictEqual(apples, oranges, `apples ${apples} !== oranges ${oranges}`);
// AssertionError [ERR_ASSERTION]: apples 1 !== oranges 2

assert.strictEqual(1, '1', new TypeError('Inputs are not identical'));
// TypeError: Inputs are not identical

```

If the values are not strictly equal, an `AssertionError` is thrown with a `message` property set equal to the value of the `message` parameter. If the `message` parameter is undefined, a default error message is assigned. If the `message` parameter is an instance of an `Error` then it will be thrown instead of the `AssertionError`.

assert.throws(fn[, error][, message])

- `fn` `<Function>`
- `error` `<RegExp> | <Function> | <Object> | <Error>`
- `message` `<string>`

Expects the function `fn` to throw an error.

If specified, `error` can be a `Class`, `RegExp`, a validation function, a validation object where each property will be tested for strict deep equality, or an instance of error where each property will be tested for strict deep equality including the non-enumerable `message` and `name` properties. When using an object, it is also possible to use a regular expression, when validating against a string property. See below for examples.

If specified, `message` will be appended to the message provided by the `AssertionError` if the `fn` call fails to throw or in case the error validation fails.

Custom validation object/error instance:

```
import assert from 'assert/strict';

const err = new TypeError('Wrong value');
err.code = 404;
err.foo = 'bar';
err.info = {
  nested: true,
  baz: 'text'
};
err.reg = /abc/i;

assert.throws(
  () => {
    throw err;
  },
  {
    name: 'TypeError',
    message: 'Wrong value',
    info: {
      nested: true,
      baz: 'text'
    }
    // Only properties on the validation object will be tested for.
    // Using nested objects requires all properties to be present. Otherwise
    // the validation is going to fail.
  }
);

// Using regular expressions to validate error properties:
throws(
  () => {
    throw err;
  },
  {

```

```

// The `name` and `message` properties are strings and using regular
// expressions on those will match against the string. If they fail, an
// error is thrown.
name: /^TypeError$/,
message: /Wrong/,
foo: 'bar',
info: {
  nested: true,
  // It is not possible to use regular expressions for nested properties!
  baz: 'text'
},
// The `reg` property contains a regular expression and only if the
// validation object contains an identical regular expression, it is going
// to pass.
reg: /abc/i
}
);

// Fails due to the different `message` and `name` properties:
throws(
  () => {
    const otherErr = new Error('Not found');
    // Copy all enumerable properties from `err` to `otherErr`.
    for (const [key, value] of Object.entries(err)) {
      otherErr[key] = value;
    }
    throw otherErr;
  },
  // The error's `message` and `name` properties will also be checked when using
  // an error as validation object.
  err
);const assert = require('assert/strict');

const err = new TypeError('Wrong value');
err.code = 404;
err.foo = 'bar';
err.info = {
  nested: true,
  baz: 'text'
};
err.reg = /abc/i;

assert.throws(
  () => {
    throw err;
  },
  {
    name: 'TypeError',
    message: 'Wrong value',
    info: {
      nested: true,

```



```

    baz: 'text'
  }
  // Only properties on the validation object will be tested for.
  // Using nested objects requires all properties to be present. Otherwise
  // the validation is going to fail.
}
);

// Using regular expressions to validate error properties:
throws(
  () => {
    throw err;
  },
  {
    // The `name` and `message` properties are strings and using regular
    // expressions on those will match against the string. If they fail, an
    // error is thrown.
    name: /^TypeError$/,
    message: /wrong/,
    foo: 'bar',
    info: {
      nested: true,
      // It is not possible to use regular expressions for nested properties!
      baz: 'text'
    },
    // The `reg` property contains a regular expression and only if the
    // validation object contains an identical regular expression, it is going
    // to pass.
    reg: /abc/i
  }
);

// Fails due to the different `message` and `name` properties:
throws(
  () => {
    const otherErr = new Error('Not found');
    // Copy all enumerable properties from `err` to `otherErr`.
    for (const [key, value] of Object.entries(err)) {
      otherErr[key] = value;
    }
    throw otherErr;
  },
  // The error's `message` and `name` properties will also be checked when using
  // an error as validation object.
  err
);

```

Validate instanceof using constructor:

```
import assert from 'assert/strict';
```

```

assert.throws(
  () => {
    throw new Error('Wrong value');
  },
  Error
);const assert = require('assert/strict');

assert.throws(
  () => {
    throw new Error('Wrong value');
  },
  Error
);

```

Validate error message using `RegExp` :

Using a regular expression runs `.toString` on the error object, and will therefore also include the error name.

```

import assert from 'assert/strict';

assert.throws(
  () => {
    throw new Error('Wrong value');
  },
  /^Error: Wrong value$/
);const assert = require('assert/strict');

assert.throws(
  () => {
    throw new Error('Wrong value');
  },
  /^Error: Wrong value$/
);

```

Custom error validation:

The function must return `true` to indicate all internal validations passed. It will otherwise fail with an `AssertionError` .

```

import assert from 'assert/strict';

assert.throws(
  () => {
    throw new Error('Wrong value');
  },
  (err) => {
    assert(err instanceof Error);
    assert(/value/.test(err));
    // Avoid returning anything from validation functions besides `true`.
    // Otherwise, it's not clear what part of the validation failed. Instead,
    // throw an error about the specific validation that failed (as done in this
    // example) and add as much helpful debugging information to that error as

```

```

    // possible.
    return true;
  },
  'unexpected error'
);const assert = require('assert/strict');

assert.throws(
  () => {
    throw new Error('Wrong value');
  },
  (err) => {
    assert(err instanceof Error);
    assert(/value/.test(err));
    // Avoid returning anything from validation functions besides `true`.
    // Otherwise, it's not clear what part of the validation failed. Instead,
    // throw an error about the specific validation that failed (as done in this
    // example) and add as much helpful debugging information to that error as
    // possible.
    return true;
  },
  'unexpected error'
);

```

`error` cannot be a string. If a string is provided as the second argument, then `error` is assumed to be omitted and the string will be used for `message` instead. This can lead to easy-to-miss mistakes. Using the same message as the thrown error message is going to result in an `ERR_AMBIGUOUS_ARGUMENT` error. Please read the example below carefully if using a string as the second argument gets considered:

```

import assert from 'assert/strict';

function throwingFirst() {
  throw new Error('First');
}

function throwingSecond() {
  throw new Error('Second');
}

function notThrowing() {}

// The second argument is a string and the input function threw an Error.
// The first case will not throw as it does not match for the error message
// thrown by the input function!
assert.throws(throwingFirst, 'Second');
// In the next example the message has no benefit over the message from the
// error and since it is not clear if the user intended to actually match
// against the error message, Node.js throws an `ERR_AMBIGUOUS_ARGUMENT` error.
assert.throws(throwingSecond, 'Second');
// TypeError [ERR_AMBIGUOUS_ARGUMENT]

// The string is only used (as message) in case the function does not throw:

```

```

assert.throws(notThrowing, 'Second');
// AssertionError [ERR_ASSERTION]: Missing expected exception: Second

// If it was intended to match for the error message do this instead:
// It does not throw because the error messages match.
assert.throws(throwingSecond, /Second$/);

// If the error message does not match, an AssertionError is thrown.
assert.throws(throwingFirst, /Second$/);
// AssertionError [ERR_ASSERTION]

```

```

const assert = require('assert/strict');

function throwingFirst() {
  throw new Error('First');
}

function throwingSecond() {
  throw new Error('Second');
}

function notThrowing() {}

// The second argument is a string and the input function threw an Error.
// The first case will not throw as it does not match for the error message
// thrown by the input function!
assert.throws(throwingFirst, 'Second');
// In the next example the message has no benefit over the message from the
// error and since it is not clear if the user intended to actually match
// against the error message, Node.js throws an `ERR_AMBIGUOUS_ARGUMENT` error.
assert.throws(throwingSecond, 'Second');
// TypeError [ERR_AMBIGUOUS_ARGUMENT]

// The string is only used (as message) in case the function does not throw:
assert.throws(notThrowing, 'Second');
// AssertionError [ERR_ASSERTION]: Missing expected exception: Second

// If it was intended to match for the error message do this instead:
// It does not throw because the error messages match.
assert.throws(throwingSecond, /Second$/);

// If the error message does not match, an AssertionError is thrown.
assert.throws(throwingFirst, /Second$/);
// AssertionError [ERR_ASSERTION]

```

Due to the confusing error-prone notation, avoid a string as the second argument.

Asynchronous Context Tracking

Source Code: [lib/async_hooks.js](#)

Introduction

These classes are used to associate state and propagate it throughout callbacks and promise chains. They allow storing data throughout the lifetime of a web request or any other asynchronous duration. It is similar to thread-local storage in other languages.

The `AsyncLocalStorage` and `AsyncResource` classes are part of the `async_hooks` module:

```
import async_hooks from 'async_hooks'; const async_hooks = require('async_hooks');
```

Class: AsyncLocalStorage

This class creates stores that stay coherent through asynchronous operations.

While you can create your own implementation on top of the `async_hooks` module, `AsyncLocalStorage` should be preferred as it is a performant and memory safe implementation that involves significant optimizations that are non-obvious to implement.

The following example uses `AsyncLocalStorage` to build a simple logger that assigns IDs to incoming HTTP requests and includes them in messages logged within each request.

```
import http from 'http';
import { AsyncLocalStorage } from 'async_hooks';

const asyncLocalStorage = new AsyncLocalStorage();

function logWithId(msg) {
  const id = asyncLocalStorage.getStore();
  console.log(`${id !== undefined ? id : '-'}:`, msg);
}

let idSeq = 0;
http.createServer((req, res) => {
  asyncLocalStorage.run(idSeq++, () => {
    logWithId('start');
    // Imagine any chain of async operations here
    setImmediate(() => {
      logWithId('finish');
      res.end();
    });
  });
}).listen(8080);

http.get('http://localhost:8080');
http.get('http://localhost:8080');
// Prints:
// 0: start
```

```
// 1: start
// 0: finish
// 1: finishconst http = require('http');
const { AsyncLocalStorage } = require('async_hooks');

const asyncLocalStorage = new AsyncLocalStorage();

function logWithId(msg) {
  const id = asyncLocalStorage.getStore();
  console.log(`${id !== undefined ? id : '-'}:`, msg);
}

let idSeq = 0;
http.createServer((req, res) => {
  asyncLocalStorage.run(idSeq++, () => {
    logWithId('start');
    // Imagine any chain of async operations here
    setImmediate(() => {
      logWithId('finish');
      res.end();
    });
  });
}).listen(8080);

http.get('http://localhost:8080');
http.get('http://localhost:8080');
// Prints:
// 0: start
// 1: start
// 0: finish
// 1: finish
```

Each instance of `AsyncLocalStorage` maintains an independent storage context. Multiple instances can safely exist simultaneously without risk of interfering with each other data.

new AsyncLocalStorage()

Creates a new instance of `AsyncLocalStorage`. Store is only provided within a `run()` call or after an `enterWith()` call.

asyncLocalStorage.disable()

Stability: 1 - Experimental

Disables the instance of `AsyncLocalStorage`. All subsequent calls to `asyncLocalStorage.getStore()` will return `undefined` until `asyncLocalStorage.run()` or `asyncLocalStorage.enterWith()` is called again.

When calling `asyncLocalStorage.disable()`, all current contexts linked to the instance will be exited.

Calling `asyncLocalStorage.disable()` is required before the `asyncLocalStorage` can be garbage collected. This does not apply to stores provided by the `asyncLocalStorage`, as those objects are garbage collected along with the corresponding async resources.

Use this method when the `asyncLocalStorage` is not in use anymore in the current process.

`asyncLocalStorage.getStore()`

- Returns: `<any>`

Returns the current store. If called outside of an asynchronous context initialized by calling `asyncLocalStorage.run()` or `asyncLocalStorage.enterWith()`, it returns `undefined`.

`asyncLocalStorage.enterWith(store)`

Stability: 1 - Experimental

- `store` `<any>`

Transitions into the context for the remainder of the current synchronous execution and then persists the store through any following asynchronous calls.

Example:

```
const store = { id: 1 };
// Replaces previous store with the given store object
asyncLocalStorage.enterWith(store);
asyncLocalStorage.getStore(); // Returns the store object
someAsyncOperation(() => {
  asyncLocalStorage.getStore(); // Returns the same object
});
```

This transition will continue for the *entire* synchronous execution. This means that if, for example, the context is entered within an event handler subsequent event handlers will also run within that context unless specifically bound to another context with an `AsyncResource`. That is why `run()` should be preferred over `enterWith()` unless there are strong reasons to use the latter method.

```
const store = { id: 1 };

emitter.on('my-event', () => {
  asyncLocalStorage.enterWith(store);
});
emitter.on('my-event', () => {
  asyncLocalStorage.getStore(); // Returns the same object
});

asyncLocalStorage.getStore(); // Returns undefined
emitter.emit('my-event');
asyncLocalStorage.getStore(); // Returns the same object
```

`asyncLocalStorage.run(store, callback[, ...args])`

- `store` `<any>`
- `callback` `<Function>`
- `...args` `<any>`

Runs a function synchronously within a context and returns its return value. The store is not accessible outside of the callback function or the asynchronous operations created within the callback.

The optional `args` are passed to the callback function.

If the callback function throws an error, the error is thrown by `run()` too. The stacktrace is not impacted by this call and the context is exited.

Example:

```
const store = { id: 2 };
try {
  asyncLocalStorage.run(store, () => {
    asyncLocalStorage.getStore(); // Returns the store object
    throw new Error();
  });
} catch (e) {
  asyncLocalStorage.getStore(); // Returns undefined
  // The error will be caught here
}
```

asyncLocalStorage.exit(callback[, ...args])

Stability: 1 - Experimental

- `callback` `<Function>`
- `...args` `<any>`

Runs a function synchronously outside of a context and returns its return value. The store is not accessible within the callback function or the asynchronous operations created within the callback. Any `getStore()` call done within the callback function will always return `undefined`.

The optional `args` are passed to the callback function.

If the callback function throws an error, the error is thrown by `exit()` too. The stacktrace is not impacted by this call and the context is re-entered.

Example:

```
// Within a call to run
try {
  asyncLocalStorage.getStore(); // Returns the store object or value
  asyncLocalStorage.exit(() => {
    asyncLocalStorage.getStore(); // Returns undefined
    throw new Error();
  });
} catch (e) {
  asyncLocalStorage.getStore(); // Returns the same object or value
  // The error will be caught here
}
```

Usage with `async/await`

If, within an async function, only one `await` call is to run within a context, the following pattern should be used:

```
async function fn() {
  await asyncLocalStorage.run(new Map(), () => {
    asyncLocalStorage.getStore().set('key', value);
    return foo(); // The return value of foo will be awaited
  });
}
```

In this example, the store is only available in the callback function and the functions called by `foo`. Outside of `run`, calling `getStore` will return `undefined`.

Troubleshooting: Context loss

In most cases your application or library code should have no issues with `AsyncLocalStorage`. But in rare cases you may face situations when the current store is lost in one of the asynchronous operations. In those cases, consider the following options.

If your code is callback-based, it is enough to promisify it with `util.promisify()`, so it starts working with native promises.

If you need to keep using callback-based API, or your code assumes a custom thenable implementation, use the `AsyncResource` class to associate the asynchronous operation with the correct execution context. To do so, you will need to identify the function call responsible for the context loss. You can do that by logging the content of `asyncLocalStorage.getStore()` after the calls you suspect are responsible for the loss. When the code logs `undefined`, the last callback called is probably responsible for the context loss.

Class: AsyncResource

The class `AsyncResource` is designed to be extended by the embedder's async resources. Using this, users can easily trigger the lifetime events of their own resources.

The `init` hook will trigger when an `AsyncResource` is instantiated.

The following is an overview of the `AsyncResource` API.

```
import { AsyncResource, executionAsyncId } from 'async_hooks';

// AsyncResource() is meant to be extended. Instantiating a
// new AsyncResource() also triggers init. If triggerAsyncId is omitted then
// async_hook.executionAsyncId() is used.
const asyncResource = new AsyncResource(
  type, { triggerAsyncId: executionAsyncId(), requireManualDestroy: false }
);

// Run a function in the execution context of the resource. This will
// * establish the context of the resource
// * trigger the AsyncHooks before callbacks
// * call the provided function `fn` with the supplied arguments
// * trigger the AsyncHooks after callbacks
// * restore the original execution context
asyncResource.runInAsyncScope(fn, thisArg, ...args);

// Call AsyncHooks destroy callbacks.
```

```

asyncResource.emitDestroy();

// Return the unique ID assigned to the AsyncResource instance.
asyncResource.asyncId();

// Return the trigger ID for the AsyncResource instance.
asyncResource.triggerAsyncId();const { AsyncResource, executionAsyncId } = require('async_hooks');

// AsyncResource() is meant to be extended. Instantiating a
// new AsyncResource() also triggers init. If triggerAsyncId is omitted then
// async_hook.executionAsyncId() is used.
const asyncResource = new AsyncResource(
  type, { triggerAsyncId: executionAsyncId(), requireManualDestroy: false }
);

// Run a function in the execution context of the resource. This will
// * establish the context of the resource
// * trigger the AsyncHooks before callbacks
// * call the provided function `fn` with the supplied arguments
// * trigger the AsyncHooks after callbacks
// * restore the original execution context
asyncResource.runInAsyncScope(fn, thisArg, ...args);

// Call AsyncHooks destroy callbacks.
asyncResource.emitDestroy();

// Return the unique ID assigned to the AsyncResource instance.
asyncResource.asyncId();

// Return the trigger ID for the AsyncResource instance.
asyncResource.triggerAsyncId();

```

new AsyncResource(type[, options])

- `type` `<string>` The type of async event.
- `options` `<Object>`
 - `triggerAsyncId` `<number>` The ID of the execution context that created this async event. **Default:** `executionAsyncId()`.
 - `requireManualDestroy` `<boolean>` If set to `true`, disables `emitDestroy` when the object is garbage collected. This usually does not need to be set (even if `emitDestroy` is called manually), unless the resource's `asyncId` is retrieved and the sensitive API's `emitDestroy` is called with it. When set to `false`, the `emitDestroy` call on garbage collection will only take place if there is at least one active `destroy` hook. **Default:** `false`.

Example usage:

```

class DBQuery extends AsyncResource {
  constructor(db) {
    super('DBQuery');
    this.db = db;
  }

  getInfo(query, callback) {

```

```

    this.db.get(query, (err, data) => {
      this.runInAsyncScope(callback, null, err, data);
    });
  }

  close() {
    this.db = null;
    this.emitDestroy();
  }
}

```

Static method: `AsyncResource.bind(fn[, type, [thisArg]])`

- `fn` `<Function>` The function to bind to the current execution context.
- `type` `<string>` An optional name to associate with the underlying `AsyncResource`.
- `thisArg` `<any>`

Binds the given function to the current execution context.

The returned function will have an `asyncResource` property referencing the `AsyncResource` to which the function is bound.

`asyncResource.bind(fn[, thisArg])`

- `fn` `<Function>` The function to bind to the current `AsyncResource`.
- `thisArg` `<any>`

Binds the given function to execute to this `AsyncResource`'s scope.

The returned function will have an `asyncResource` property referencing the `AsyncResource` to which the function is bound.

`asyncResource.runInAsyncScope(fn[, thisArg, ...args])`

- `fn` `<Function>` The function to call in the execution context of this async resource.
- `thisArg` `<any>` The receiver to be used for the function call.
- `...args` `<any>` Optional arguments to pass to the function.

Call the provided function with the provided arguments in the execution context of the async resource. This will establish the context, trigger the `AsyncHooks` before callbacks, call the function, trigger the `AsyncHooks` after callbacks, and then restore the original execution context.

`asyncResource.emitDestroy()`

- Returns: `<AsyncResource>` A reference to `asyncResource`.

Call all `destroy` hooks. This should only ever be called once. An error will be thrown if it is called more than once. This **must** be manually called. If the resource is left to be collected by the GC then the `destroy` hooks will never be called.

`asyncResource.asyncId()`

- Returns: `<number>` The unique `asyncId` assigned to the resource.

`asyncResource.triggerAsyncId()`

- Returns: `<number>` The same `triggerAsyncId` that is passed to the `AsyncResource` constructor.

Using `AsyncResource` for a `Worker` thread pool

The following example shows how to use the `AsyncResource` class to properly provide async tracking for a `Worker` pool. Other resource pools, such as database connection pools, can follow a similar model.

Assuming that the task is adding two numbers, using a file named `task_processor.js` with the following content:

```
import { parentPort } from 'worker_threads';
parentPort.on('message', (task) => {
  parentPort.postMessage(task.a + task.b);
});const { parentPort } = require('worker_threads');
parentPort.on('message', (task) => {
  parentPort.postMessage(task.a + task.b);
});
```

a Worker pool around it could use the following structure:

```
import { AsyncResource } from 'async_hooks';
import { EventEmitter } from 'events';
import path from 'path';
import { Worker } from 'worker_threads';

const kTaskInfo = Symbol('kTaskInfo');
const kWorkerFreedEvent = Symbol('kWorkerFreedEvent');

class WorkerPoolTaskInfo extends AsyncResource {
  constructor(callback) {
    super('WorkerPoolTaskInfo');
    this.callback = callback;
  }

  done(err, result) {
    this.runInAsyncScope(this.callback, null, err, result);
    this.emitDestroy(); // `TaskInfo`s are used only once.
  }
}

export default class WorkerPool extends EventEmitter {
  constructor(numThreads) {
    super();
    this.numThreads = numThreads;
    this.workers = [];
    this.freeWorkers = [];
    this.tasks = [];

    for (let i = 0; i < numThreads; i++)
      this.addNewWorker();

    // Any time the kWorkerFreedEvent is emitted, dispatch
    // the next task pending in the queue, if any.
    this.on(kWorkerFreedEvent, () => {
      if (this.tasks.length > 0) {
        const { task, callback } = this.tasks.shift();
```

```

        this.runTask(task, callback);
    }
});
}

addNewWorker() {
    const worker = new Worker(new URL('task_processor.js', import.meta.url));
    worker.on('message', (result) => {
        // In case of success: Call the callback that was passed to `runTask`,
        // remove the `TaskInfo` associated with the Worker, and mark it as free
        // again.
        worker[kTaskInfo].done(null, result);
        worker[kTaskInfo] = null;
        this.freeWorkers.push(worker);
        this.emit(kWorkerFreedEvent);
    });
    worker.on('error', (err) => {
        // In case of an uncaught exception: Call the callback that was passed to
        // `runTask` with the error.
        if (worker[kTaskInfo])
            worker[kTaskInfo].done(err, null);
        else
            this.emit('error', err);
        // Remove the worker from the list and start a new Worker to replace the
        // current one.
        this.workers.splice(this.workers.indexOf(worker), 1);
        this.addNewWorker();
    });
    this.workers.push(worker);
    this.freeWorkers.push(worker);
    this.emit(kWorkerFreedEvent);
}

runTask(task, callback) {
    if (this.freeWorkers.length === 0) {
        // No free threads, wait until a worker thread becomes free.
        this.tasks.push({ task, callback });
        return;
    }

    const worker = this.freeWorkers.pop();
    worker[kTaskInfo] = new WorkerPoolTaskInfo(callback);
    worker.postMessage(task);
}

close() {
    for (const worker of this.workers) worker.terminate();
}

}const { AsyncResource } = require('async_hooks');
const { EventEmitter } = require('events');
const path = require('path');
```

```

const { Worker } = require('worker_threads');

const kTaskInfo = Symbol('kTaskInfo');
const kWorkerFreedEvent = Symbol('kWorkerFreedEvent');

class WorkerPoolTaskInfo extends AsyncResource {
  constructor(callback) {
    super('WorkerPoolTaskInfo');
    this.callback = callback;
  }

  done(err, result) {
    this.runInAsyncScope(this.callback, null, err, result);
    this.emitDestroy(); // `TaskInfo`s are used only once.
  }
}

class WorkerPool extends EventEmitter {
  constructor(numThreads) {
    super();
    this.numThreads = numThreads;
    this.workers = [];
    this.freeWorkers = [];
    this.tasks = [];

    for (let i = 0; i < numThreads; i++)
      this.addNewWorker();

    // Any time the kWorkerFreedEvent is emitted, dispatch
    // the next task pending in the queue, if any.
    this.on(kWorkerFreedEvent, () => {
      if (this.tasks.length > 0) {
        const { task, callback } = this.tasks.shift();
        this.runTask(task, callback);
      }
    });
  }

  addNewWorker() {
    const worker = new Worker(path.resolve(__dirname, 'task_processor.js'));
    worker.on('message', (result) => {
      // In case of success: Call the callback that was passed to `runTask`,
      // remove the `TaskInfo` associated with the Worker, and mark it as free
      // again.
      worker[kTaskInfo].done(null, result);
      worker[kTaskInfo] = null;
      this.freeWorkers.push(worker);
      this.emit(kWorkerFreedEvent);
    });
    worker.on('error', (err) => {
      // In case of an uncaught exception: Call the callback that was passed to

```

```

    // `runTask` with the error.
    if (worker[kTaskInfo])
        worker[kTaskInfo].done(err, null);
    else
        this.emit('error', err);
    // Remove the worker from the list and start a new Worker to replace the
    // current one.
    this.workers.splice(this.workers.indexOf(worker), 1);
    this.addNewWorker();
  });
  this.workers.push(worker);
  this.freeWorkers.push(worker);
  this.emit(kWorkerFreedEvent);
}

runTask(task, callback) {
  if (this.freeWorkers.length === 0) {
    // No free threads, wait until a worker thread becomes free.
    this.tasks.push({ task, callback });
    return;
  }

  const worker = this.freeWorkers.pop();
  worker[kTaskInfo] = new WorkerPoolTaskInfo(callback);
  worker.postMessage(task);
}

close() {
  for (const worker of this.workers) worker.terminate();
}
}

module.exports = WorkerPool;

```

Without the explicit tracking added by the `WorkerPoolTaskInfo` objects, it would appear that the callbacks are associated with the individual `Worker` objects. However, the creation of the `Worker`s is not associated with the creation of the tasks and does not provide information about when tasks were scheduled.

This pool could be used as follows:

```

import WorkerPool from './worker_pool.js';
import os from 'os';

const pool = new WorkerPool(os.cpus().length);

let finished = 0;
for (let i = 0; i < 10; i++) {
  pool.runTask({ a: 42, b: 100 }, (err, result) => {
    console.log(i, err, result);
    if (++finished === 10)
      pool.close();
  });
}

```

```

});
}const WorkerPool = require('./worker_pool.js');
const os = require('os');

const pool = new WorkerPool(os.cpus().length);

let finished = 0;
for (let i = 0; i < 10; i++) {
  pool.runTask({ a: 42, b: 100 }, (err, result) => {
    console.log(i, err, result);
    if (++finished === 10)
      pool.close();
  });
}
}

```

Integrating AsyncResource with EventEmitter

Event listeners triggered by an `EventEmitter` may be run in a different execution context than the one that was active when `eventEmitter.on()` was called.

The following example shows how to use the `AsyncResource` class to properly associate an event listener with the correct execution context. The same approach can be applied to a `Stream` or a similar event-driven class.

```

import { createServer } from 'http';
import { AsyncResource, executionAsyncId } from 'async_hooks';

const server = createServer((req, res) => {
  req.on('close', AsyncResource.bind(() => {
    // Execution context is bound to the current outer scope.
  }));
  req.on('close', () => {
    // Execution context is bound to the scope that caused 'close' to emit.
  });
  res.end();
}).listen(3000);const { createServer } = require('http');
const { AsyncResource, executionAsyncId } = require('async_hooks');

const server = createServer((req, res) => {
  req.on('close', AsyncResource.bind(() => {
    // Execution context is bound to the current outer scope.
  }));
  req.on('close', () => {
    // Execution context is bound to the scope that caused 'close' to emit.
  });
  res.end();
}).listen(3000);

```

Async hooks

Source Code: [lib/async_hooks.js](#)

The `async_hooks` module provides an API to track asynchronous resources. It can be accessed using:

```
import async_hooks from 'async_hooks'; const async_hooks = require('async_hooks');
```

Terminology

An asynchronous resource represents an object with an associated callback. This callback may be called multiple times, for example, the `'connection'` event in `net.createServer()`, or just a single time like in `fs.open()`. A resource can also be closed before the callback is called. `AsyncHook` does not explicitly distinguish between these different cases but will represent them as the abstract concept that is a resource.

If `Worker`s are used, each thread has an independent `async_hooks` interface, and each thread will use a new set of async IDs.

Overview

Following is a simple overview of the public API.

```
import async_hooks from 'async_hooks';

// Return the ID of the current execution context.
const eid = async_hooks.executionAsyncId();

// Return the ID of the handle responsible for triggering the callback of the
// current execution scope to call.
const tid = async_hooks.triggerAsyncId();

// Create a new AsyncHook instance. All of these callbacks are optional.
const asyncHook =
  async_hooks.createHook({ init, before, after, destroy, promiseResolve });

// Allow callbacks of this AsyncHook instance to call. This is not an implicit
// action after running the constructor, and must be explicitly run to begin
// executing callbacks.
asyncHook.enable();

// Disable listening for new asynchronous events.
asyncHook.disable();

//
// The following are the callbacks that can be passed to createHook().
//

// init is called during object construction. The resource may not have
// completed construction when this callback runs, therefore all fields of the
```

```

// resource referenced by "asyncId" may not have been populated.
function init(asyncId, type, triggerAsyncId, resource) { }

// Before is called just before the resource's callback is called. It can be
// called 0-N times for handles (such as TCPWrap), and will be called exactly 1
// time for requests (such as FSReqCallback).
function before(asyncId) { }

// After is called just after the resource's callback has finished.
function after(asyncId) { }

// Destroy is called when the resource is destroyed.
function destroy(asyncId) { }

// promiseResolve is called only for promise resources, when the
// `resolve` function passed to the `Promise` constructor is invoked
// (either directly or through other means of resolving a promise).
function promiseResolve(asyncId) { }const async_hooks = require('async_hooks');

// Return the ID of the current execution context.
const eid = async_hooks.executionAsyncId();

// Return the ID of the handle responsible for triggering the callback of the
// current execution scope to call.
const tid = async_hooks.triggerAsyncId();

// Create a new AsyncHook instance. All of these callbacks are optional.
const asyncHook =
  async_hooks.createHook({ init, before, after, destroy, promiseResolve });

// Allow callbacks of this AsyncHook instance to call. This is not an implicit
// action after running the constructor, and must be explicitly run to begin
// executing callbacks.
asyncHook.enable();

// Disable listening for new asynchronous events.
asyncHook.disable();

//
// The following are the callbacks that can be passed to createHook().
//

// init is called during object construction. The resource may not have
// completed construction when this callback runs, therefore all fields of the
// resource referenced by "asyncId" may not have been populated.
function init(asyncId, type, triggerAsyncId, resource) { }

// Before is called just before the resource's callback is called. It can be
// called 0-N times for handles (such as TCPWrap), and will be called exactly 1
// time for requests (such as FSReqCallback).
function before(asyncId) { }

```

```
// After is called just after the resource's callback has finished.
function after(asyncId) { }

// Destroy is called when the resource is destroyed.
function destroy(asyncId) { }

// promiseResolve is called only for promise resources, when the
// `resolve` function passed to the `Promise` constructor is invoked
// (either directly or through other means of resolving a promise).
function promiseResolve(asyncId) { }
```

async_hooks.createHook(callbacks)

- `callbacks` `<Object>` The `HookCallbacks` to register
 - `init` `<Function>` The `init` callback.
 - `before` `<Function>` The `before` callback.
 - `after` `<Function>` The `after` callback.
 - `destroy` `<Function>` The `destroy` callback.
 - `promiseResolve` `<Function>` The `promiseResolve` callback.
- Returns: `<AsyncHook>` Instance used for disabling and enabling hooks

Registers functions to be called for different lifetime events of each async operation.

The callbacks `init()` / `before()` / `after()` / `destroy()` are called for the respective asynchronous event during a resource's lifetime.

All callbacks are optional. For example, if only resource cleanup needs to be tracked, then only the `destroy` callback needs to be passed. The specifics of all functions that can be passed to `callbacks` is in the `HookCallbacks` section.

```
import { createHook } from 'async_hooks';

const asyncHook = createHook({
  init(asyncId, type, triggerAsyncId, resource) { },
  destroy(asyncId) { }
});const async_hooks = require('async_hooks');

const asyncHook = async_hooks.createHook({
  init(asyncId, type, triggerAsyncId, resource) { },
  destroy(asyncId) { }
});
```

The callbacks will be inherited via the prototype chain:

```
class MyAsyncCallbacks {
  init(asyncId, type, triggerAsyncId, resource) { }
  destroy(asyncId) {}
}

class MyAddedCallbacks extends MyAsyncCallbacks {
```

```

before(asyncId) { }
after(asyncId) { }
}

const asyncHook = async_hooks.createHook(new MyAddedCallbacks());

```

Because promises are asynchronous resources whose lifecycle is tracked via the async hooks mechanism, the `init()`, `before()`, `after()`, and `destroy()` callbacks *must not* be async functions that return promises.

Error handling

If any `AsyncHook` callbacks throw, the application will print the stack trace and exit. The exit path does follow that of an uncaught exception, but all `'uncaughtException'` listeners are removed, thus forcing the process to exit. The `'exit'` callbacks will still be called unless the application is run with `--abort-on-uncaught-exception`, in which case a stack trace will be printed and the application exits, leaving a core file.

The reason for this error handling behavior is that these callbacks are running at potentially volatile points in an object's lifetime, for example during class construction and destruction. Because of this, it is deemed necessary to bring down the process quickly in order to prevent an unintentional abort in the future. This is subject to change in the future if a comprehensive analysis is performed to ensure an exception can follow the normal control flow without unintentional side effects.

Printing in AsyncHooks callbacks

Because printing to the console is an asynchronous operation, `console.log()` will cause the AsyncHooks callbacks to be called. Using `console.log()` or similar asynchronous operations inside an AsyncHooks callback function will thus cause an infinite recursion. An easy solution to this when debugging is to use a synchronous logging operation such as `fs.writeFileSync(file, msg, flag)`. This will print to the file and will not invoke AsyncHooks recursively because it is synchronous.

```

import { writeFileSync } from 'fs';
import { format } from 'util';

function debug(...args) {
  // Use a function like this one when debugging inside an AsyncHooks callback
  writeFileSync('log.out', `${format(...args)}\n`, { flag: 'a' });
}const fs = require('fs');
const util = require('util');

function debug(...args) {
  // Use a function like this one when debugging inside an AsyncHooks callback
  fs.writeFileSync('log.out', `${util.format(...args)}\n`, { flag: 'a' });
}

```

If an asynchronous operation is needed for logging, it is possible to keep track of what caused the asynchronous operation using the information provided by AsyncHooks itself. The logging should then be skipped when it was the logging itself that caused AsyncHooks callback to call. By doing this the otherwise infinite recursion is broken.

Class: AsyncHook

The class `AsyncHook` exposes an interface for tracking lifetime events of asynchronous operations.

`asyncHook.enable()`

- Returns: `<AsyncHook>` A reference to `asyncHook`.

Enable the callbacks for a given `AsyncHook` instance. If no callbacks are provided, enabling is a no-op.

The `AsyncHook` instance is disabled by default. If the `AsyncHook` instance should be enabled immediately after creation, the following pattern can be used.

```
import { createHook } from 'async_hooks';

const hook = createHook(callbacks).enable();const async_hooks = require('async_hooks');

const hook = async_hooks.createHook(callbacks).enable();
```

asyncHook.disable()

- Returns: `<AsyncHook>` A reference to `asyncHook`.

Disable the callbacks for a given `AsyncHook` instance from the global pool of `AsyncHook` callbacks to be executed. Once a hook has been disabled it will not be called again until enabled.

For API consistency `disable()` also returns the `AsyncHook` instance.

Hook callbacks

Key events in the lifetime of asynchronous events have been categorized into four areas: instantiation, before/after the callback is called, and when the instance is destroyed.

init(asyncId, type, triggerAsyncId, resource)

- `asyncId` `<number>` A unique ID for the async resource.
- `type` `<string>` The type of the async resource.
- `triggerAsyncId` `<number>` The unique ID of the async resource in whose execution context this async resource was created.
- `resource` `<Object>` Reference to the resource representing the async operation, needs to be released during *destroy*.

Called when a class is constructed that has the *possibility* to emit an asynchronous event. This *does not* mean the instance must call `before` / `after` before `destroy` is called, only that the possibility exists.

This behavior can be observed by doing something like opening a resource then closing it before the resource can be used. The following snippet demonstrates this.

```
import { createServer } from 'net';

createServer().listen(function() { this.close(); });
// OR
clearTimeout(setTimeout(() => {}, 10));require('net').createServer().listen(function() { this.close(); });
// OR
clearTimeout(setTimeout(() => {}, 10));
```

Every new resource is assigned an ID that is unique within the scope of the current Node.js instance.

type

The `type` is a string identifying the type of resource that caused `init` to be called. Generally, it will correspond to the name of the resource's constructor.

```
FSEVENTWRAP, FSREQCALLBACK, GETADDRINFOREQWRAP, GETNAMEINFOREQWRAP, HTTPINCOMINGMESSAGE,
HTTPCLIENTREQUEST, JSSTREAM, PIPECONNECTWRAP, PIPEWRAP, PROCESSWRAP, QUERYWRAP,
SHUTDOWNWRAP, SIGNALWRAP, STATWATCHER, TCPCONNECTWRAP, TCPSERVERWRAP, TCPWRAP,
TTYWRAP, UDPSENDWRAP, UDPWRAP, WRITEWRAP, ZLIB, SSLCONNECTION, PBKDF2REQUEST,
RANDOMBYTESREQUEST, TLSWRAP, Microtask, Timeout, Immediate, TickObject
```

There is also the `PROMISE` resource type, which is used to track `Promise` instances and asynchronous work scheduled by them.

Users are able to define their own `type` when using the public embedder API.

It is possible to have type name collisions. Embedders are encouraged to use unique prefixes, such as the npm package name, to prevent collisions when listening to the hooks.

`triggerAsyncId`

`triggerAsyncId` is the `asyncId` of the resource that caused (or "triggered") the new resource to initialize and that caused `init` to call. This is different from `async_hooks.executionAsyncId()` that only shows *when* a resource was created, while `triggerAsyncId` shows *why* a resource was created.

The following is a simple demonstration of `triggerAsyncId`:

```
import { createHook, executionAsyncId } from 'async_hooks';
import { stdout } from 'process';
import net from 'net';

createHook({
  init(asyncId, type, triggerAsyncId) {
    const eid = executionAsyncId();
    fs.writeFileSync(
      stdout.fd,
      `${type}(${asyncId}): trigger: ${triggerAsyncId} execution: ${eid}\n`);
  }
}).enable();

net.createServer((conn) => {}).listen(8080);const { createHook, executionAsyncId } = require('async_hooks');
const { stdout } = require('process');
const net = require('net');

createHook({
  init(asyncId, type, triggerAsyncId) {
    const eid = executionAsyncId();
    fs.writeFileSync(
      stdout.fd,
      `${type}(${asyncId}): trigger: ${triggerAsyncId} execution: ${eid}\n`);
  }
}).enable();

net.createServer((conn) => {}).listen(8080);
```

Output when hitting the server with `nc localhost 8080`:

```
TCPSERVERWRAP(5): trigger: 1 execution: 1
TCPWRAP(7): trigger: 5 execution: 0
```

The `TCPSERVERWRAP` is the server which receives the connections.

The `TCPWRAP` is the new connection from the client. When a new connection is made, the `TCPWrap` instance is immediately constructed. This happens outside of any JavaScript stack. (An `executionAsyncId()` of `0` means that it is being executed from C++ with no JavaScript stack above it.) With only that information, it would be impossible to link resources together in terms of what caused them to be created, so `triggerAsyncId` is given the task of propagating what resource is responsible for the new resource's existence.

resource

`resource` is an object that represents the actual async resource that has been initialized. This can contain useful information that can vary based on the value of `type`. For instance, for the `GETADDRINFOREQWRAP` resource type, `resource` provides the host name used when looking up the IP address for the host in `net.Server.listen()`. The API for accessing this information is not supported, but using the Embedder API, users can provide and document their own resource objects. For example, such a resource object could contain the SQL query being executed.

In some cases the resource object is reused for performance reasons, it is thus not safe to use it as a key in a `WeakMap` or add properties to it.

Asynchronous context example

The following is an example with additional information about the calls to `init` between the `before` and `after` calls, specifically what the callback to `listen()` will look like. The output formatting is slightly more elaborate to make calling context easier to see.

```
const { fd } = process.stdout;

let indent = 0;
async_hooks.createHook({
  init(asyncId, type, triggerAsyncId) {
    const eid = async_hooks.executionAsyncId();
    const indentStr = ' '.repeat(indent);
    fs.writeSync(
      fd,
      `${indentStr}${type}${asyncId}):` +
      ` trigger: ${triggerAsyncId} execution: ${eid}\n`);
  },
  before(asyncId) {
    const indentStr = ' '.repeat(indent);
    fs.writeSync(fd, `${indentStr}before:  ${asyncId}\n`);
    indent += 2;
  },
  after(asyncId) {
    indent -= 2;
    const indentStr = ' '.repeat(indent);
    fs.writeSync(fd, `${indentStr}after:  ${asyncId}\n`);
  },
  destroy(asyncId) {
    const indentStr = ' '.repeat(indent);
    fs.writeSync(fd, `${indentStr}destroy:  ${asyncId}\n`);
  },
}).enable();
```

```
net.createServer(() => {}).listen(8080, () => {
  // Let's wait 10ms before logging the server started.
  setTimeout(() => {
    console.log('>>>', async_hooks.executionAsyncId());
  }, 10);
});
```

Output from only starting the server:

```
TCPSEVERWRAP(5): trigger: 1 execution: 1
TickObject(6): trigger: 5 execution: 1
before: 6
  Timeout(7): trigger: 6 execution: 6
after: 6
destroy: 6
before: 7
>>> 7
  TickObject(8): trigger: 7 execution: 7
after: 7
before: 8
after: 8
```

As illustrated in the example, `executionAsyncId()` and `execution` each specify the value of the current execution context; which is delineated by calls to `before` and `after`.

Only using `execution` to graph resource allocation results in the following:

```
root(1)
  ^
  |
TickObject(6)
  ^
  |
Timeout(7)
```

The `TCPSEVERWRAP` is not part of this graph, even though it was the reason for `console.log()` being called. This is because binding to a port without a host name is a *synchronous* operation, but to maintain a completely asynchronous API the user's callback is placed in a `process.nextTick()`. Which is why `TickObject` is present in the output and is a 'parent' for `.listen()` callback.

The graph only shows *when* a resource was created, not *why*, so to track the *why* use `triggerAsyncId`. Which can be represented with the following graph:

```
bootstrap(1)
  |
  v
TCPSEVERWRAP(5)
  |
  v
TickObject(6)
  |
```


v

Timeout(7)

before(asyncId)

- `asyncId` `<number>`

When an asynchronous operation is initiated (such as a TCP server receiving a new connection) or completes (such as writing data to disk) a callback is called to notify the user. The `before` callback is called just before said callback is executed. `asyncId` is the unique identifier assigned to the resource about to execute the callback.

The `before` callback will be called 0 to N times. The `before` callback will typically be called 0 times if the asynchronous operation was cancelled or, for example, if no connections are received by a TCP server. Persistent asynchronous resources like a TCP server will typically call the `before` callback multiple times, while other operations like `fs.open()` will call it only once.

after(asyncId)

- `asyncId` `<number>`

Called immediately after the callback specified in `before` is completed.

If an uncaught exception occurs during execution of the callback, then `after` will run *after* the `'uncaughtException'` event is emitted or a `domain`'s handler runs.

destroy(asyncId)

- `asyncId` `<number>`

Called after the resource corresponding to `asyncId` is destroyed. It is also called asynchronously from the embedder API `emitDestroy()`.

Some resources depend on garbage collection for cleanup, so if a reference is made to the `resource` object passed to `init` it is possible that `destroy` will never be called, causing a memory leak in the application. If the resource does not depend on garbage collection, then this will not be an issue.

promiseResolve(asyncId)

- `asyncId` `<number>`

Called when the `resolve` function passed to the `Promise` constructor is invoked (either directly or through other means of resolving a promise).

`resolve()` does not do any observable synchronous work.

The `Promise` is not necessarily fulfilled or rejected at this point if the `Promise` was resolved by assuming the state of another `Promise`.

```
new Promise((resolve) => resolve(true)).then((a) => {});
```

calls the following callbacks:

```
init for PROMISE with id 5, trigger id: 1
  promise resolve 5      # corresponds to resolve(true)
init for PROMISE with id 6, trigger id: 5 # the Promise returned by then()
  before 6              # the then() callback is entered
  promise resolve 6      # the then() callback resolves the promise by returning
  after 6
```

async_hooks.executionAsyncResource()

- Returns: `<Object>` The resource representing the current execution. Useful to store data within the resource.

Resource objects returned by `executionAsyncResource()` are most often internal Node.js handle objects with undocumented APIs. Using any functions or properties on the object is likely to crash your application and should be avoided.

Using `executionAsyncResource()` in the top-level execution context will return an empty object as there is no handle or request object to use, but having an object representing the top-level can be helpful.

```
import { open } from 'fs';
import { executionAsyncId, executionAsyncResource } from 'async_hooks';

console.log(executionAsyncId(), executionAsyncResource()); // 1 {}
open(new URL(import.meta.url), 'r', (err, fd) => {
  console.log(executionAsyncId(), executionAsyncResource()); // 7 FSReqWrap
});const { open } = require('fs');
const { executionAsyncId, executionAsyncResource } = require('async_hooks');

console.log(executionAsyncId(), executionAsyncResource()); // 1 {}
open(__filename, 'r', (err, fd) => {
  console.log(executionAsyncId(), executionAsyncResource()); // 7 FSReqWrap
});
```

This can be used to implement continuation local storage without the use of a tracking `Map` to store the metadata:

```
import { createServer } from 'http';
import {
  executionAsyncId,
  executionAsyncResource,
  createHook
} from 'async_hooks';
const sym = Symbol('state'); // Private symbol to avoid pollution

createHook({
  init(asyncId, type, triggerAsyncId, resource) {
    const cr = executionAsyncResource();
    if (cr) {
      resource[sym] = cr[sym];
    }
  }
}).enable();

const server = createServer((req, res) => {
  executionAsyncResource()[sym] = { state: req.url };
  setTimeout(function() {
    res.end(JSON.stringify(executionAsyncResource()[sym]));
  }, 100);
}).listen(3000);const { createServer } = require('http');
const {
  executionAsyncId,
  executionAsyncResource,
```

```

createHook
} = require('async_hooks');
const sym = Symbol('state'); // Private symbol to avoid pollution

createHook({
  init(asyncId, type, triggerAsyncId, resource) {
    const cr = executionAsyncResource();
    if (cr) {
      resource[sym] = cr[sym];
    }
  }
}).enable();

const server = createServer((req, res) => {
  executionAsyncResource()[sym] = { state: req.url };
  setTimeout(function() {
    res.end(JSON.stringify(executionAsyncResource()[sym]));
  }, 100);
}).listen(3000);

```

async_hooks.executionAsyncId()

- Returns: `<number>` The `asyncId` of the current execution context. Useful to track when something calls.

```

import { executionAsyncId } from 'async_hooks';

console.log(executionAsyncId()); // 1 - bootstrap
fs.open(path, 'r', (err, fd) => {
  console.log(executionAsyncId()); // 6 - open()
});const async_hooks = require('async_hooks');

console.log(async_hooks.executionAsyncId()); // 1 - bootstrap
fs.open(path, 'r', (err, fd) => {
  console.log(async_hooks.executionAsyncId()); // 6 - open()
});

```

The ID returned from `executionAsyncId()` is related to execution timing, not causality (which is covered by `triggerAsyncId()`):

```

const server = net.createServer((conn) => {
  // Returns the ID of the server, not of the new connection, because the
  // callback runs in the execution scope of the server's MakeCallback().
  async_hooks.executionAsyncId();

}).listen(port, () => {
  // Returns the ID of a TickObject (process.nextTick()) because all
  // callbacks passed to .listen() are wrapped in a nextTick().
  async_hooks.executionAsyncId();
});

```

Promise contexts may not get precise `executionAsyncIds` by default. See the section on [promise execution tracking](#).

async_hooks.triggerAsyncId()

- Returns: `<number>` The ID of the resource responsible for calling the callback that is currently being executed.

```
const server = net.createServer((conn) => {
  // The resource that caused (or triggered) this callback to be called
  // was that of the new connection. Thus the return value of triggerAsyncId()
  // is the asyncId of "conn".
  async_hooks.triggerAsyncId();

}).listen(port, () => {
  // Even though all callbacks passed to .listen() are wrapped in a nextTick()
  // the callback itself exists because the call to the server's .listen()
  // was made. So the return value would be the ID of the server.
  async_hooks.triggerAsyncId();
});
```

Promise contexts may not get valid `triggerAsyncId`s by default. See the section on [promise execution tracking](#).

Promise execution tracking

By default, promise executions are not assigned `asyncId`s due to the relatively expensive nature of the [promise introspection API](#) provided by V8. This means that programs using promises or `async / await` will not get correct execution and trigger ids for promise callback contexts by default.

```
import { executionAsyncId, triggerAsyncId } from 'async_hooks';

Promise.resolve(1729).then(() => {
  console.log(`eid ${executionAsyncId()} tid ${triggerAsyncId()}`);
});
// produces:
// eid 1 tid 0const { executionAsyncId, triggerAsyncId } = require('async_hooks');

Promise.resolve(1729).then(() => {
  console.log(`eid ${executionAsyncId()} tid ${triggerAsyncId()}`);
});
// produces:
// eid 1 tid 0
```

Observe that the `then()` callback claims to have executed in the context of the outer scope even though there was an asynchronous hop involved. Also, the `triggerAsyncId` value is `0`, which means that we are missing context about the resource that caused (triggered) the `then()` callback to be executed.

Installing async hooks via `async_hooks.createHook` enables promise execution tracking:

```
import { createHook, executionAsyncId, triggerAsyncId } from 'async_hooks';
createHook({ init() {} }).enable(); // forces PromiseHooks to be enabled.
Promise.resolve(1729).then(() => {
  console.log(`eid ${executionAsyncId()} tid ${triggerAsyncId()}`);
});
```

```
// produces:
// eid 7 tid 6const { createHook, exectionAsyncId, triggerAsyncId } = require('async_hooks');

createHook({ init() {} }).enable(); // forces PromiseHooks to be enabled.
Promise.resolve(1729).then(() => {
  console.log(`eid ${executionAsyncId()} tid ${triggerAsyncId()}`);
});
// produces:
// eid 7 tid 6
```

In this example, adding any actual hook function enabled the tracking of promises. There are two promises in the example above; the promise created by `Promise.resolve()` and the promise returned by the call to `then()`. In the example above, the first promise got the `asyncId` 6 and the latter got `asyncId` 7. During the execution of the `then()` callback, we are executing in the context of promise with `asyncId` 7. This promise was triggered by async resource 6.

Another subtlety with promises is that `before` and `after` callbacks are run only on chained promises. That means promises not created by `then()` / `catch()` will not have the `before` and `after` callbacks fired on them. For more details see the details of the V8 [PromiseHooks](#) API.

JavaScript embedder API

Library developers that handle their own asynchronous resources performing tasks like I/O, connection pooling, or managing callback queues may use the `AsyncResource` JavaScript API so that all the appropriate callbacks are called.

Class: `AsyncResource`

The documentation for this class has moved [AsyncResource](#).

Class: `AsyncLocalStorage`

The documentation for this class has moved [AsyncLocalStorage](#).

Buffer

Stability: 2 - Stable

Source Code: [lib/buffer.js](#)

`Buffer` objects are used to represent a fixed-length sequence of bytes. Many Node.js APIs support `Buffer` s.

The `Buffer` class is a subclass of JavaScript's `Uint8Array` class and extends it with methods that cover additional use cases. Node.js APIs accept plain `Uint8Array` s wherever `Buffer` s are supported as well.

While the `Buffer` class is available within the global scope, it is still recommended to explicitly reference it via an import or require statement.

```
import { Buffer } from 'buffer';

// Creates a zero-filled Buffer of length 10.
const buf1 = Buffer.alloc(10);
```

```

// Creates a Buffer of length 10,
// filled with bytes which all have the value `1`.
const buf2 = Buffer.alloc(10, 1);

// Creates an uninitialized buffer of length 10.
// This is faster than calling Buffer.alloc() but the returned
// Buffer instance might contain old data that needs to be
// overwritten using fill(), write(), or other functions that fill the Buffer's
// contents.
const buf3 = Buffer.allocUnsafe(10);

// Creates a Buffer containing the bytes [1, 2, 3].
const buf4 = Buffer.from([1, 2, 3]);

// Creates a Buffer containing the bytes [1, 1, 1, 1] - the entries
// are all truncated using `(value & 255)` to fit into the range 0-255.
const buf5 = Buffer.from([257, 257.5, -255, '1']);

// Creates a Buffer containing the UTF-8-encoded bytes for the string 'tést':
// [0x74, 0xc3, 0xa9, 0x73, 0x74] (in hexadecimal notation)
// [116, 195, 169, 115, 116] (in decimal notation)
const buf6 = Buffer.from('tést');

// Creates a Buffer containing the Latin-1 bytes [0x74, 0xe9, 0x73, 0x74].
const buf7 = Buffer.from('tést', 'latin1'); const { Buffer } = require('buffer');

// Creates a zero-filled Buffer of length 10.
const buf1 = Buffer.alloc(10);

// Creates a Buffer of length 10,
// filled with bytes which all have the value `1`.
const buf2 = Buffer.alloc(10, 1);

// Creates an uninitialized buffer of length 10.
// This is faster than calling Buffer.alloc() but the returned
// Buffer instance might contain old data that needs to be
// overwritten using fill(), write(), or other functions that fill the Buffer's
// contents.
const buf3 = Buffer.allocUnsafe(10);

// Creates a Buffer containing the bytes [1, 2, 3].
const buf4 = Buffer.from([1, 2, 3]);

// Creates a Buffer containing the bytes [1, 1, 1, 1] - the entries
// are all truncated using `(value & 255)` to fit into the range 0-255.
const buf5 = Buffer.from([257, 257.5, -255, '1']);

// Creates a Buffer containing the UTF-8-encoded bytes for the string 'tést':
// [0x74, 0xc3, 0xa9, 0x73, 0x74] (in hexadecimal notation)
// [116, 195, 169, 115, 116] (in decimal notation)

```

```
const buf6 = Buffer.from('tést');

// Creates a Buffer containing the Latin-1 bytes [0x74, 0xe9, 0x73, 0x74].
const buf7 = Buffer.from('tést', 'latin1');
```

Buffers and character encodings

When converting between `Buffer`s and strings, a character encoding may be specified. If no character encoding is specified, UTF-8 will be used as the default.

```
import { Buffer } from 'buffer';

const buf = Buffer.from('hello world', 'utf8');

console.log(buf.toString('hex'));
// Prints: 68656c6c6f20776f726c64
console.log(buf.toString('base64'));
// Prints: aGVsbG8gd29ybGQ=

console.log(Buffer.from('fhqwhgads', 'utf8'));
// Prints: <Buffer 66 68 71 77 68 67 61 64 73>
console.log(Buffer.from('fhqwhgads', 'utf16le'));
// Prints: <Buffer 66 00 68 00 71 00 77 00 68 00 67 00 61 00 64 00 73 00>const { Buffer } = require('buffer');


const buf = Buffer.from('hello world', 'utf8');

console.log(buf.toString('hex'));
// Prints: 68656c6c6f20776f726c64
console.log(buf.toString('base64'));
// Prints: aGVsbG8gd29ybGQ=

console.log(Buffer.from('fhqwhgads', 'utf8'));
// Prints: <Buffer 66 68 71 77 68 67 61 64 73>
console.log(Buffer.from('fhqwhgads', 'utf16le'));
// Prints: <Buffer 66 00 68 00 71 00 77 00 68 00 67 00 61 00 64 00 73 00>
```

Node.js buffers accept all case variations of encoding strings that they receive. For example, UTF-8 can be specified as `'utf8'`, `'UTF8'` or `'uTf8'`.

The character encodings currently supported by Node.js are the following:

- `'utf8'` (alias: `'utf-8'`): Multi-byte encoded Unicode characters. Many web pages and other document formats use **UTF-8**. This is the default character encoding. When decoding a `Buffer` into a string that does not exclusively contain valid UTF-8 data, the Unicode replacement character `U+FFFD`  will be used to represent those errors.
- `'utf16le'` (alias: `'utf-16le'`): Multi-byte encoded Unicode characters. Unlike `'utf8'`, each character in the string will be encoded using either 2 or 4 bytes. Node.js only supports the **little-endian** variant of **UTF-16**.
- `'latin1'`: Latin-1 stands for **ISO-8859-1**. This character encoding only supports the Unicode characters from `U+0000` to `U+00FF`. Each character is encoded using a single byte. Characters that do not fit into that range are truncated and will be mapped to characters in that range.

Converting a `Buffer` into a string using one of the above is referred to as decoding, and converting a string into a `Buffer` is referred to as encoding.

Node.js also supports the following binary-to-text encodings. For binary-to-text encodings, the naming convention is reversed: Converting a `Buffer` into a string is typically referred to as encoding, and converting a string into a `Buffer` as decoding.

- `'base64'`: [Base64](#) encoding. When creating a `Buffer` from a string, this encoding will also correctly accept "URL and Filename Safe Alphabet" as specified in [RFC 4648, Section 5](#). Whitespace characters such as spaces, tabs, and new lines contained within the base64-encoded string are ignored.
- `'base64url'`: [base64url](#) encoding as specified in [RFC 4648, Section 5](#). When creating a `Buffer` from a string, this encoding will also correctly accept regular base64-encoded strings. When encoding a `Buffer` to a string, this encoding will omit padding.
- `'hex'`: Encode each byte as two hexadecimal characters. Data truncation may occur when decoding strings that do exclusively contain valid hexadecimal characters. See below for an example.

The following legacy character encodings are also supported:

- `'ascii'`: For 7-bit [ASCII](#) data only. When encoding a string into a `Buffer`, this is equivalent to using `'latin1'`. When decoding a `Buffer` into a string, using this encoding will additionally unset the highest bit of each byte before decoding as `'latin1'`. Generally, there should be no reason to use this encoding, as `'utf8'` (or, if the data is known to always be ASCII-only, `'latin1'`) will be a better choice when encoding or decoding ASCII-only text. It is only provided for legacy compatibility.
- `'binary'`: Alias for `'latin1'`. See [binary strings](#) for more background on this topic. The name of this encoding can be very misleading, as all of the encodings listed here convert between strings and binary data. For converting between strings and `Buffer`s, typically `'utf8'` is the right choice.
- `'ucs2'`, `'ucs-2'`: Aliases of `'utf16le'`. UCS-2 used to refer to a variant of UTF-16 that did not support characters that had code points larger than U+FFFF. In Node.js, these code points are always supported.

```
import { Buffer } from 'buffer';

Buffer.from('1ag', 'hex');
// Prints <Buffer 1a>, data truncated when first non-hexadecimal value
// ('g') encountered.

Buffer.from('1a7g', 'hex');
// Prints <Buffer 1a>, data truncated when data ends in single digit ('7').

Buffer.from('1634', 'hex');
// Prints <Buffer 16 34>, all data represented.const { Buffer } = require('buffer');

Buffer.from('1ag', 'hex');
// Prints <Buffer 1a>, data truncated when first non-hexadecimal value
// ('g') encountered.

Buffer.from('1a7g', 'hex');
// Prints <Buffer 1a>, data truncated when data ends in single digit ('7').

Buffer.from('1634', 'hex');
// Prints <Buffer 16 34>, all data represented.
```


Modern Web browsers follow the [WHATWG Encoding Standard](#) which aliases both `'latin1'` and `'ISO-8859-1'` to `'win-1252'`. This means that while doing something like `http.get()`, if the returned charset is one of those listed in the WHATWG specification it is possible that the server actually returned `'win-1252'`-encoded data, and using `'latin1'` encoding may incorrectly decode the characters.

Buffers and TypedArrays

`Buffer` instances are also JavaScript `Uint8Array` and `TypedArray` instances. All `TypedArray` methods are available on `Buffer`s. There are, however, subtle incompatibilities between the `Buffer` API and the `TypedArray` API.

In particular:

- While `TypedArray.prototype.slice()` creates a copy of part of the `TypedArray`, `Buffer.prototype.slice()` creates a view over the existing `Buffer` without copying. This behavior can be surprising, and only exists for legacy compatibility. `TypedArray.prototype.subarray()` can be used to achieve the behavior of `Buffer.prototype.slice()` on both `Buffer`s and other `TypedArray`s.
- `buf.toString()` is incompatible with its `TypedArray` equivalent.
- A number of methods, e.g. `buf.indexOf()`, support additional arguments.

There are two ways to create new `TypedArray` instances from a `Buffer`:

- Passing a `Buffer` to a `TypedArray` constructor will copy the `Buffer`'s contents, interpreted as an array of integers, and not as a byte sequence of the target type.

```
import { Buffer } from 'buffer';

const buf = Buffer.from([1, 2, 3, 4]);
const uint32array = new Uint32Array(buf);

console.log(uint32array);

// Prints: Uint32Array(4) [ 1, 2, 3, 4 ]const { Buffer } = require('buffer');

const buf = Buffer.from([1, 2, 3, 4]);
const uint32array = new Uint32Array(buf);

console.log(uint32array);

// Prints: Uint32Array(4) [ 1, 2, 3, 4 ]
```

- Passing the `Buffer`'s underlying `ArrayBuffer` will create a `TypedArray` that shares its memory with the `Buffer`.

```
import { Buffer } from 'buffer';

const buf = Buffer.from('hello', 'utf16le');
const uint16array = new Uint16Array(
  buf.buffer,
  buf.byteOffset,
  buf.length / Uint16Array.BYTES_PER_ELEMENT);

console.log(uint16array);
```

```
// Prints: Uint16Array(5) [ 104, 101, 108, 108, 111 ]const { Buffer } = require('buffer');

const buf = Buffer.from('hello', 'utf16le');
const uint16array = new Uint16Array(
  buf.buffer,
  buf.byteOffset,
  buf.length / Uint16Array.BYTES_PER_ELEMENT);

console.log(uint16array);

// Prints: Uint16Array(5) [ 104, 101, 108, 108, 111 ]
```

It is possible to create a new `Buffer` that shares the same allocated memory as a `TypedArray` instance by using the `TypedArray` object's `.buffer` property in the same way. `Buffer.from()` behaves like `new Uint8Array()` in this context.

```
import { Buffer } from 'buffer';

const arr = new Uint16Array(2);

arr[0] = 5000;
arr[1] = 4000;

// Copies the contents of `arr`.
const buf1 = Buffer.from(arr);

// Shares memory with `arr`.
const buf2 = Buffer.from(arr.buffer);

console.log(buf1);
// Prints: <Buffer 88 a0>
console.log(buf2);
// Prints: <Buffer 88 13 a0 0f>

arr[1] = 6000;

console.log(buf1);
// Prints: <Buffer 88 a0>
console.log(buf2);
// Prints: <Buffer 88 13 70 17>const { Buffer } = require('buffer');

const arr = new Uint16Array(2);

arr[0] = 5000;
arr[1] = 4000;

// Copies the contents of `arr`.
const buf1 = Buffer.from(arr);

// Shares memory with `arr`.
const buf2 = Buffer.from(arr.buffer);
```

```

console.log(buf1);
// Prints: <Buffer 88 a0>
console.log(buf2);
// Prints: <Buffer 88 13 a0 0f>

arr[1] = 6000;

console.log(buf1);
// Prints: <Buffer 88 a0>
console.log(buf2);
// Prints: <Buffer 88 13 70 17>

```

When creating a `Buffer` using a `TypedArray`'s `.buffer`, it is possible to use only a portion of the underlying `ArrayBuffer` by passing in `byteOffset` and `length` parameters.

```

import { Buffer } from 'buffer';

const arr = new Uint16Array(20);
const buf = Buffer.from(arr.buffer, 0, 16);

console.log(buf.length);
// Prints: 16

const { Buffer } = require('buffer');

const arr = new Uint16Array(20);
const buf = Buffer.from(arr.buffer, 0, 16);

console.log(buf.length);
// Prints: 16

```

The `Buffer.from()` and `TypedArray.from()` have different signatures and implementations. Specifically, the `TypedArray` variants accept a second argument that is a mapping function that is invoked on every element of the typed array:

- `TypedArray.from(source[, mapFn[, thisArg]])`

The `Buffer.from()` method, however, does not support the use of a mapping function:

- `Buffer.from(array)`
- `Buffer.from(buffer)`
- `Buffer.from(arrayBuffer[, byteOffset[, length]])`
- `Buffer.from(string[, encoding])`

Buffers and iteration

`Buffer` instances can be iterated over using `for..of` syntax:

```

import { Buffer } from 'buffer';

const buf = Buffer.from([1, 2, 3]);

```

```

for (const b of buf) {
  console.log(b);
}
// Prints:
// 1
// 2
// 3const { Buffer } = require('buffer');

const buf = Buffer.from([1, 2, 3]);

for (const b of buf) {
  console.log(b);
}
// Prints:
// 1
// 2
// 3

```

Additionally, the `buf.values()`, `buf.keys()`, and `buf.entries()` methods can be used to create iterators.

Class: Blob

Stability: 1 - Experimental

A **Blob** encapsulates immutable, raw data that can be safely shared across multiple worker threads.

new buffer.Blob([sources[, options]])

- **sources** `<string[]> | <ArrayBuffer[]> | <TypedArray[]> | <DataView[]> | <Blob[]>` An array of string, `<ArrayBuffer>`, `<TypedArray>`, `<DataView>`, or `<Blob>` objects, or any mix of such objects, that will be stored within the **Blob**.
- **options** `<Object>`
 - **endings** `<string>` One of either `'transparent'` or `'native'`. When set to `'native'`, line endings in string source parts will be converted to the platform native line-ending as specified by `require('os').EOL`.
 - **type** `<string>` The Blob content-type. The intent is for **type** to convey the MIME media type of the data, however no validation of the type format is performed.

Creates a new **Blob** object containing a concatenation of the given sources.

`<ArrayBuffer>`, `<TypedArray>`, `<DataView>`, and `<Buffer>` sources are copied into the 'Blob' and can therefore be safely modified after the 'Blob' is created.

String sources are encoded as UTF-8 byte sequences and copied into the Blob. Unmatched surrogate pairs within each string part will be replaced by Unicode U+FFFD replacement characters.

blob.arrayBuffer()

- Returns: `<Promise>`

Returns a promise that fulfills with an `<ArrayBuffer>` containing a copy of the **Blob** data.

blob.size

The total size of the `Blob` in bytes.

`blob.slice([start, [end, [type]]])`

- `start` `<number>` The starting index.
- `end` `<number>` The ending index.
- `type` `<string>` The content-type for the new `Blob`

Creates and returns a new `Blob` containing a subset of this `Blob` objects data. The original `Blob` is not altered.

`blob.stream()`

- Returns: `<ReadableStream>`

Returns a new `ReadableStream` that allows the content of the `Blob` to be read.

`blob.text()`

- Returns: `<Promise>`

Returns a promise that fulfills with the contents of the `Blob` decoded as a UTF-8 string.

`blob.type`

- Type: `<string>`

The content-type of the `Blob`.

`Blob` objects and `MessageChannel`

Once a `<Blob>` object is created, it can be sent via `MessagePort` to multiple destinations without transferring or immediately copying the data. The data contained by the `Blob` is copied only when the `arrayBuffer()` or `text()` methods are called.

```
import { Blob, Buffer } from 'buffer';
import { setTimeout as delay } from 'timers/promises';

const blob = new Blob(['hello there']);

const mc1 = new MessageChannel();
const mc2 = new MessageChannel();

mc1.port1.onmessage = async ({ data }) => {
  console.log(await data.arrayBuffer());
  mc1.port1.close();
};

mc2.port1.onmessage = async ({ data }) => {
  await delay(1000);
  console.log(await data.arrayBuffer());
  mc2.port1.close();
};

mc1.port2.postMessage(blob);
mc2.port2.postMessage(blob);
```

```
// The Blob is still usable after posting.
data.text().then(console.log);const { Blob, Buffer } = require('buffer');
const { setTimeout: delay } = require('timers/promises');

const blob = new Blob(['hello there']);

const mc1 = new MessageChannel();
const mc2 = new MessageChannel();

mc1.port1.onmessage = async ({ data }) => {
  console.log(await data.arrayBuffer());
  mc1.port1.close();
};

mc2.port1.onmessage = async ({ data }) => {
  await delay(1000);
  console.log(await data.arrayBuffer());
  mc2.port1.close();
};

mc1.port2.postMessage(blob);
mc2.port2.postMessage(blob);

// The Blob is still usable after posting.
data.text().then(console.log);
```

Class: Buffer

The `Buffer` class is a global type for dealing with binary data directly. It can be constructed in a variety of ways.

Static method: `Buffer.alloc(size[, fill[, encoding]])`

- `size` `<integer>` The desired length of the new `Buffer`.
- `fill` `<string> | <Buffer> | <Uint8Array> | <integer>` A value to pre-fill the new `Buffer` with. **Default:** `0`.
- `encoding` `<string>` If `fill` is a string, this is its encoding. **Default:** `'utf8'`.

Allocates a new `Buffer` of `size` bytes. If `fill` is `undefined`, the `Buffer` will be zero-filled.

```
import { Buffer } from 'buffer';

const buf = Buffer.alloc(5);

console.log(buf);
// Prints: <Buffer 00 00 00 00 00>const { Buffer } = require('buffer');

const buf = Buffer.alloc(5);

console.log(buf);
// Prints: <Buffer 00 00 00 00 00>
```

If `size` is larger than `buffer.constants.MAX_LENGTH` or smaller than 0, `ERR_INVALID_ARG_VALUE` is thrown.

If `fill` is specified, the allocated `Buffer` will be initialized by calling `buf.fill(fill)`.

```
import { Buffer } from 'buffer';

const buf = Buffer.alloc(5, 'a');

console.log(buf);
// Prints: <Buffer 61 61 61 61 61>

const { Buffer } = require('buffer');

const buf = Buffer.alloc(5, 'a');

console.log(buf);
// Prints: <Buffer 61 61 61 61 61>
```

If both `fill` and `encoding` are specified, the allocated `Buffer` will be initialized by calling `buf.fill(fill, encoding)`.

```
import { Buffer } from 'buffer';

const buf = Buffer.alloc(11, 'aGVsbG8gd29ybGQ=', 'base64');

console.log(buf);
// Prints: <Buffer 68 65 6c 6c 6f 20 77 6f 72 6c 64>

const { Buffer } = require('buffer');

const buf = Buffer.alloc(11, 'aGVsbG8gd29ybGQ=', 'base64');

console.log(buf);
// Prints: <Buffer 68 65 6c 6c 6f 20 77 6f 72 6c 64>
```

Calling `Buffer.alloc()` can be measurably slower than the alternative `Buffer.allocUnsafe()` but ensures that the newly created `Buffer` instance contents will never contain sensitive data from previous allocations, including data that might not have been allocated for `Buffer`s.

A `TypeError` will be thrown if `size` is not a number.

Static method: `Buffer.allocUnsafe(size)`

- `size` `<integer>` The desired length of the new `Buffer`.

Allocates a new `Buffer` of `size` bytes. If `size` is larger than `buffer.constants.MAX_LENGTH` or smaller than 0, `ERR_INVALID_ARG_VALUE` is thrown.

The underlying memory for `Buffer` instances created in this way is *not initialized*. The contents of the newly created `Buffer` are unknown and *may contain sensitive data*. Use `Buffer.alloc()` instead to initialize `Buffer` instances with zeroes.

```
import { Buffer } from 'buffer';

const buf = Buffer.allocUnsafe(10);

console.log(buf);
```

```
// Prints (contents may vary): <Buffer a0 8b 28 3f 01 00 00 00 50 32>

buf.fill(0);

console.log(buf);
// Prints: <Buffer 00 00 00 00 00 00 00 00 00 00>const { Buffer } = require('buffer');

const buf = Buffer.allocUnsafe(10);

console.log(buf);
// Prints (contents may vary): <Buffer a0 8b 28 3f 01 00 00 00 50 32>

buf.fill(0);

console.log(buf);
// Prints: <Buffer 00 00 00 00 00 00 00 00 00 00>
```

A `TypeError` will be thrown if `size` is not a number.

The `Buffer` module pre-allocates an internal `Buffer` instance of size `Buffer.poolSize` that is used as a pool for the fast allocation of new `Buffer` instances created using `Buffer.allocUnsafe()`, `Buffer.from(array)`, `Buffer.concat()`, and the deprecated `new Buffer(size)` constructor only when `size` is less than or equal to `Buffer.poolSize >> 1` (floor of `Buffer.poolSize` divided by two).

Use of this pre-allocated internal memory pool is a key difference between calling `Buffer.alloc(size, fill)` vs. `Buffer.allocUnsafe(size).fill(fill)`. Specifically, `Buffer.alloc(size, fill)` will *never* use the internal `Buffer` pool, while `Buffer.allocUnsafe(size).fill(fill)` will use the internal `Buffer` pool if `size` is less than or equal to half `Buffer.poolSize`. The difference is subtle but can be important when an application requires the additional performance that `Buffer.allocUnsafe()` provides.

Static method: `Buffer.allocUnsafeSlow(size)`

- `size <integer>` The desired length of the new `Buffer`.

Allocates a new `Buffer` of `size` bytes. If `size` is larger than `buffer.constants.MAX_LENGTH` or smaller than 0, `ERR_INVALID_ARG_VALUE` is thrown. A zero-length `Buffer` is created if `size` is 0.

The underlying memory for `Buffer` instances created in this way is *not initialized*. The contents of the newly created `Buffer` are unknown and *may contain sensitive data*. Use `buf.fill(0)` to initialize such `Buffer` instances with zeroes.

When using `Buffer.allocUnsafe()` to allocate new `Buffer` instances, allocations under 4 KB are sliced from a single pre-allocated `Buffer`. This allows applications to avoid the garbage collection overhead of creating many individually allocated `Buffer` instances. This approach improves both performance and memory usage by eliminating the need to track and clean up as many individual `ArrayBuffer` objects.

However, in the case where a developer may need to retain a small chunk of memory from a pool for an indeterminate amount of time, it may be appropriate to create an un-pooled `Buffer` instance using `Buffer.allocUnsafeSlow()` and then copying out the relevant bits.

```
import { Buffer } from 'buffer';

// Need to keep around a few small chunks of memory.
const store = [];

socket.on('readable', () => {
  let data;
```



```

while (null !== (data = readable.read())) {
  // Allocate for retained data.
  const sb = Buffer.allocUnsafeSlow(10);

  // Copy the data into the new allocation.
  data.copy(sb, 0, 0, 10);

  store.push(sb);
}
});const { Buffer } = require('buffer');

// Need to keep around a few small chunks of memory.
const store = [];

socket.on('readable', () => {
  let data;
  while (null !== (data = readable.read())) {
    // Allocate for retained data.
    const sb = Buffer.allocUnsafeSlow(10);

    // Copy the data into the new allocation.
    data.copy(sb, 0, 0, 10);

    store.push(sb);
  }
});

```

A `TypeError` will be thrown if `size` is not a number.

Static method: `Buffer.byteLength(string[, encoding])`

- `string` `<string>` | `<Buffer>` | `<TypedArray>` | `<DataView>` | `<ArrayBuffer>` | `<SharedArrayBuffer>` A value to calculate the length of.
- `encoding` `<string>` If `string` is a string, this is its encoding. **Default:** `'utf8'`.
- Returns: `<integer>` The number of bytes contained within `string`.

Returns the byte length of a string when encoded using `encoding`. This is not the same as `String.prototype.length`, which does not account for the encoding that is used to convert the string into bytes.

For `'base64'`, `'base64url'`, and `'hex'`, this function assumes valid input. For strings that contain non-base64/hex-encoded data (e.g. whitespace), the return value might be greater than the length of a `Buffer` created from the string.

```

import { Buffer } from 'buffer';

const str = '\u00bd + \u00bc = \u00be';

console.log(`${str}: ${str.length} characters, ` +
  `${Buffer.byteLength(str, 'utf8')} bytes`);
// Prints: ½ + ¼ = ¾: 9 characters, 12 bytes
const { Buffer } = require('buffer');

const str = '\u00bd + \u00bc = \u00be';

```

```
console.log(`${str}: ${str.length} characters, ` +
    `${Buffer.byteLength(str, 'utf8')} bytes`);

// Prints: ½ + ¼ = ¾: 9 characters, 12 bytes
```

When `string` is a `Buffer` / `DataView` / `TypedArray` / `ArrayBuffer` / `SharedArrayBuffer`, the byte length as reported by `.byteLength` is returned.

Static method: `Buffer.compare(buf1, buf2)`

- `buf1` `<Buffer>` | `<Uint8Array>`
- `buf2` `<Buffer>` | `<Uint8Array>`
- Returns: `<integer>` Either `-1`, `0`, or `1`, depending on the result of the comparison. See `buf.compare()` for details.

Compares `buf1` to `buf2`, typically for the purpose of sorting arrays of `Buffer` instances. This is equivalent to calling `buf1.compare(buf2)`.

```
import { Buffer } from 'buffer';

const buf1 = Buffer.from('1234');
const buf2 = Buffer.from('0123');
const arr = [buf1, buf2];

console.log(arr.sort(Buffer.compare));
// Prints: [ <Buffer 30 31 32 33>, <Buffer 31 32 33 34> ]
// (This result is equal to: [buf2, buf1].)const { Buffer } = require('buffer');

const buf1 = Buffer.from('1234');
const buf2 = Buffer.from('0123');
const arr = [buf1, buf2];

console.log(arr.sort(Buffer.compare));
// Prints: [ <Buffer 30 31 32 33>, <Buffer 31 32 33 34> ]
// (This result is equal to: [buf2, buf1].)
```

Static method: `Buffer.concat(list[, totalLength])`

- `list` `<Buffer[]>` | `<Uint8Array[]>` List of `Buffer` or `Uint8Array` instances to concatenate.
- `totalLength` `<integer>` Total length of the `Buffer` instances in `list` when concatenated.
- Returns: `<Buffer>`

Returns a new `Buffer` which is the result of concatenating all the `Buffer` instances in the `list` together.

If the list has no items, or if the `totalLength` is 0, then a new zero-length `Buffer` is returned.

If `totalLength` is not provided, it is calculated from the `Buffer` instances in `list` by adding their lengths.

If `totalLength` is provided, it is coerced to an unsigned integer. If the combined length of the `Buffer`s in `list` exceeds `totalLength`, the result is truncated to `totalLength`.

```
import { Buffer } from 'buffer';

// Create a single `Buffer` from a list of three `Buffer` instances.
```

```

const buf1 = Buffer.alloc(10);
const buf2 = Buffer.alloc(14);
const buf3 = Buffer.alloc(18);
const totalLength = buf1.length + buf2.length + buf3.length;

console.log(totalLength);
// Prints: 42

const bufA = Buffer.concat([buf1, buf2, buf3], totalLength);

console.log(bufA);
// Prints: <Buffer 00 00 00 00 00 ...>
console.log(bufA.length);
// Prints: 42const { Buffer } = require('buffer');

// Create a single `Buffer` from a list of three `Buffer` instances.

const buf1 = Buffer.alloc(10);
const buf2 = Buffer.alloc(14);
const buf3 = Buffer.alloc(18);
const totalLength = buf1.length + buf2.length + buf3.length;

console.log(totalLength);
// Prints: 42

const bufA = Buffer.concat([buf1, buf2, buf3], totalLength);

console.log(bufA);
// Prints: <Buffer 00 00 00 00 00 ...>
console.log(bufA.length);
// Prints: 42

```

`Buffer.concat()` may also use the internal `Buffer` pool like `Buffer.allocUnsafe()` does.

Static method: `Buffer.from(array)`

- `array` `<integer[]>`

Allocates a new `Buffer` using an `array` of bytes in the range `0 - 255`. Array entries outside that range will be truncated to fit into it.

```

import { Buffer } from 'buffer';

// Creates a new Buffer containing the UTF-8 bytes of the string 'buffer'.
const buf = Buffer.from([0x62, 0x75, 0x66, 0x66, 0x65, 0x72]);const { Buffer } = require('buffer');

// Creates a new Buffer containing the UTF-8 bytes of the string 'buffer'.
const buf = Buffer.from([0x62, 0x75, 0x66, 0x66, 0x65, 0x72]);

```

A `TypeError` will be thrown if `array` is not an `Array` or another type appropriate for `Buffer.from()` variants.

`Buffer.from(array)` and `Buffer.from(string)` may also use the internal `Buffer` pool like `Buffer.allocUnsafe()` does.

Static method: `Buffer.from(arrayBuffer[, byteOffset[, length]])`

- `arrayBuffer` `<ArrayBuffer> | <SharedArrayBuffer>` An `ArrayBuffer` , `SharedArrayBuffer` , for example the `.buffer` property of a `TypedArray` .
- `byteOffset` `<integer>` Index of first byte to expose. **Default:** `0` .
- `length` `<integer>` Number of bytes to expose. **Default:** `arrayBuffer.byteLength - byteOffset` .

This creates a view of the `ArrayBuffer` without copying the underlying memory. For example, when passed a reference to the `.buffer` property of a `TypedArray` instance, the newly created `Buffer` will share the same allocated memory as the `TypedArray` 's underlying `ArrayBuffer` .

```
import { Buffer } from 'buffer';

const arr = new Uint16Array(2);

arr[0] = 5000;
arr[1] = 4000;

// Shares memory with `arr`.
const buf = Buffer.from(arr.buffer);

console.log(buf);
// Prints: <Buffer 88 13 a0 0f>

// Changing the original Uint16Array changes the Buffer also.
arr[1] = 6000;

console.log(buf);
// Prints: <Buffer 88 13 70 17>const { Buffer } = require('buffer');

const arr = new Uint16Array(2);

arr[0] = 5000;
arr[1] = 4000;

// Shares memory with `arr`.
const buf = Buffer.from(arr.buffer);

console.log(buf);
// Prints: <Buffer 88 13 a0 0f>

// Changing the original Uint16Array changes the Buffer also.
arr[1] = 6000;

console.log(buf);
// Prints: <Buffer 88 13 70 17>
```

The optional `byteOffset` and `length` arguments specify a memory range within the `arrayBuffer` that will be shared by the `Buffer` .

```
import { Buffer } from 'buffer';
```

```

const ab = new ArrayBuffer(10);
const buf = Buffer.from(ab, 0, 2);

console.log(buf.length);
// Prints: 2
const { Buffer } = require('buffer');

const ab = new ArrayBuffer(10);
const buf = Buffer.from(ab, 0, 2);

console.log(buf.length);
// Prints: 2

```

A `TypeError` will be thrown if `arrayBuffer` is not an `ArrayBuffer` or a `SharedArrayBuffer` or another type appropriate for `Buffer.from()` variants.

It is important to remember that a backing `ArrayBuffer` can cover a range of memory that extends beyond the bounds of a `TypedArray` view. A new `Buffer` created using the `buffer` property of a `TypedArray` may extend beyond the range of the `TypedArray`:

```

import { Buffer } from 'buffer';

const arrA = Uint8Array.from([0x63, 0x64, 0x65, 0x66]); // 4 elements
const arrB = new Uint8Array(arrA.buffer, 1, 2); // 2 elements
console.log(arrA.buffer === arrB.buffer); // true

const buf = Buffer.from(arrB.buffer);
console.log(buf);
// Prints: <Buffer 63 64 65 66>
const { Buffer } = require('buffer');

const arrA = Uint8Array.from([0x63, 0x64, 0x65, 0x66]); // 4 elements
const arrB = new Uint8Array(arrA.buffer, 1, 2); // 2 elements
console.log(arrA.buffer === arrB.buffer); // true

const buf = Buffer.from(arrB.buffer);
console.log(buf);
// Prints: <Buffer 63 64 65 66>

```

Static method: `Buffer.from(buffer)`

- `buffer` `<Buffer>` | `<Uint8Array>` An existing `Buffer` or `Uint8Array` from which to copy data.

Copies the passed `buffer` data onto a new `Buffer` instance.

```

import { Buffer } from 'buffer';

const buf1 = Buffer.from('buffer');
const buf2 = Buffer.from(buf1);

buf1[0] = 0x61;

console.log(buf1.toString());
// Prints: auffer

```

```

console.log(buf2.toString());
// Prints: bufferconst { Buffer } = require('buffer');

const buf1 = Buffer.from('buffer');
const buf2 = Buffer.from(buf1);

buf1[0] = 0x61;

console.log(buf1.toString());
// Prints: auffer
console.log(buf2.toString());
// Prints: buffer

```

A `TypeError` will be thrown if `buffer` is not a `Buffer` or another type appropriate for `Buffer.from()` variants.

Static method: `Buffer.from(object[, offsetOrEncoding[, length]])`

- `object` `<Object>` An object supporting `Symbol.toPrimitive` or `valueOf()`.
- `offsetOrEncoding` `<integer> | <string>` A byte-offset or encoding.
- `length` `<integer>` A length.

For objects whose `valueOf()` function returns a value not strictly equal to `object`, returns `Buffer.from(object.valueOf(), offsetOrEncoding, length)`.

```

import { Buffer } from 'buffer';

const buf = Buffer.from(new String('this is a test'));
// Prints: <Buffer 74 68 69 73 20 69 73 20 61 20 74 65 73 74>const { Buffer } = require('buffer');

const buf = Buffer.from(new String('this is a test'));
// Prints: <Buffer 74 68 69 73 20 69 73 20 61 20 74 65 73 74>

```

For objects that support `Symbol.toPrimitive`, returns `Buffer.from(object[Symbol.toPrimitive]('string'), offsetOrEncoding)`.

```

import { Buffer } from 'buffer';

class Foo {
  [Symbol.toPrimitive]() {
    return 'this is a test';
  }
}

const buf = Buffer.from(new Foo(), 'utf8');
// Prints: <Buffer 74 68 69 73 20 69 73 20 61 20 74 65 73 74>const { Buffer } = require('buffer');

class Foo {
  [Symbol.toPrimitive]() {
    return 'this is a test';
  }
}

```

```
const buf = Buffer.from(new Foo(), 'utf8');  
// Prints: <Buffer 74 68 69 73 20 69 73 20 61 20 74 65 73 74>
```

A `TypeError` will be thrown if `object` does not have the mentioned methods or is not of another type appropriate for `Buffer.from()` variants.

Static method: `Buffer.from(string[, encoding])`

- `string` `<string>` A string to encode.
- `encoding` `<string>` The encoding of `string`. Default: `'utf8'`.

Creates a new `Buffer` containing `string`. The `encoding` parameter identifies the character encoding to be used when converting `string` into bytes.

```
import { Buffer } from 'buffer';  
  
const buf1 = Buffer.from('this is a tést');  
const buf2 = Buffer.from('7468697320697320612074c3a97374', 'hex');  
  
console.log(buf1.toString());  
// Prints: this is a tést  
console.log(buf2.toString());  
// Prints: this is a tést  
console.log(buf1.toString('latin1'));  
// Prints: this is a tÃ©st  
const { Buffer } = require('buffer');  
  
const buf1 = Buffer.from('this is a tést');  
const buf2 = Buffer.from('7468697320697320612074c3a97374', 'hex');  
  
console.log(buf1.toString());  
// Prints: this is a tést  
console.log(buf2.toString());  
// Prints: this is a tést  
console.log(buf1.toString('latin1'));  
// Prints: this is a tÃ©st
```

A `TypeError` will be thrown if `string` is not a string or another type appropriate for `Buffer.from()` variants.

Static method: `Buffer.isBuffer(obj)`

- `obj` `<Object>`
- Returns: `<boolean>`

Returns `true` if `obj` is a `Buffer`, `false` otherwise.

```
import { Buffer } from 'buffer';  
  
Buffer.isBuffer(Buffer.alloc(10)); // true  
Buffer.isBuffer(Buffer.from('foo')); // true  
Buffer.isBuffer('a string'); // false  
Buffer.isBuffer([]); // false
```

```
Buffer.isBuffer(new Uint8Array(1024)); // false
const { Buffer } = require('buffer');

Buffer.isBuffer(Buffer.alloc(10)); // true
Buffer.isBuffer(Buffer.from('foo')); // true
Buffer.isBuffer('a string'); // false
Buffer.isBuffer([]); // false
Buffer.isBuffer(new Uint8Array(1024)); // false
```

Static method: Buffer.isEncoding(encoding)

- `encoding` `<string>` A character encoding name to check.
- Returns: `<boolean>`

Returns `true` if `encoding` is the name of a supported character encoding, or `false` otherwise.

```
import { Buffer } from 'buffer';

console.log(Buffer.isEncoding('utf8'));
// Prints: true

console.log(Buffer.isEncoding('hex'));
// Prints: true

console.log(Buffer.isEncoding('utf/8'));
// Prints: false

console.log(Buffer.isEncoding(''));
// Prints: false
const { Buffer } = require('buffer');

console.log(Buffer.isEncoding('utf8'));
// Prints: true

console.log(Buffer.isEncoding('hex'));
// Prints: true

console.log(Buffer.isEncoding('utf/8'));
// Prints: false

console.log(Buffer.isEncoding(''));
// Prints: false
```

Class property: Buffer.poolSize

- `<integer>` Default: 8192

This is the size (in bytes) of pre-allocated internal `Buffer` instances used for pooling. This value may be modified.

buf[index]

- `index` `<integer>`

The index operator `[index]` can be used to get and set the octet at position `index` in `buf`. The values refer to individual bytes, so the legal value range is between `0x00` and `0xFF` (hex) or `0` and `255` (decimal).

This operator is inherited from `Uint8Array`, so its behavior on out-of-bounds access is the same as `Uint8Array`. In other words, `buf[index]` returns `undefined` when `index` is negative or greater or equal to `buf.length`, and `buf[index] = value` does not modify the buffer if `index` is negative or `>= buf.length`.

```
import { Buffer } from 'buffer';

// Copy an ASCII string into a `Buffer` one byte at a time.
// (This only works for ASCII-only strings. In general, one should use
// `Buffer.from()` to perform this conversion.)

const str = 'Node.js';
const buf = Buffer.allocUnsafe(str.length);

for (let i = 0; i < str.length; i++) {
  buf[i] = str.charCodeAt(i);
}

console.log(buf.toString('utf8'));
// Prints: Node.jsconst { Buffer } = require('buffer');

// Copy an ASCII string into a `Buffer` one byte at a time.
// (This only works for ASCII-only strings. In general, one should use
// `Buffer.from()` to perform this conversion.)

const str = 'Node.js';
const buf = Buffer.allocUnsafe(str.length);

for (let i = 0; i < str.length; i++) {
  buf[i] = str.charCodeAt(i);
}

console.log(buf.toString('utf8'));
// Prints: Node.js
```

buf.buffer

- `<ArrayBuffer>` The underlying `ArrayBuffer` object based on which this `Buffer` object is created.

This `ArrayBuffer` is not guaranteed to correspond exactly to the original `Buffer`. See the notes on `buf.byteOffset` for details.

```
import { Buffer } from 'buffer';

const arrayBuffer = new ArrayBuffer(16);
const buffer = Buffer.from(arrayBuffer);

console.log(buffer.buffer === arrayBuffer);
// Prints: trueconst { Buffer } = require('buffer');
```

```
const arrayBuffer = new ArrayBuffer(16);
const buffer = Buffer.from(arrayBuffer);

console.log(buffer.buffer === arrayBuffer);
// Prints: true
```

buf.byteOffset

- `<integer>` The `byteOffset` of the `Buffer`'s underlying `ArrayBuffer` object.

When setting `byteOffset` in `Buffer.from(ArrayBuffer, byteOffset, length)`, or sometimes when allocating a `Buffer` smaller than `Buffer.poolSize`, the buffer does not start from a zero offset on the underlying `ArrayBuffer`.

This can cause problems when accessing the underlying `ArrayBuffer` directly using `buf.buffer`, as other parts of the `ArrayBuffer` may be unrelated to the `Buffer` object itself.

A common issue when creating a `TypedArray` object that shares its memory with a `Buffer` is that in this case one needs to specify the `byteOffset` correctly:

```
import { Buffer } from 'buffer';

// Create a buffer smaller than `Buffer.poolSize`.
const nodeBuffer = new Buffer.from([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]);

// When casting the Node.js Buffer to an Int8Array, use the byteOffset
// to refer only to the part of `nodeBuffer.buffer` that contains the memory
// for `nodeBuffer`.
new Int8Array(nodeBuffer.buffer, nodeBuffer.byteOffset, nodeBuffer.length);const { Buffer } = require('buffer');

// Create a buffer smaller than `Buffer.poolSize`.
const nodeBuffer = new Buffer.from([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]);

// When casting the Node.js Buffer to an Int8Array, use the byteOffset
// to refer only to the part of `nodeBuffer.buffer` that contains the memory
// for `nodeBuffer`.
new Int8Array(nodeBuffer.buffer, nodeBuffer.byteOffset, nodeBuffer.length);
```

buf.compare(target[, targetStart[, targetEnd[, sourceStart[, sourceEnd]]]])

- `target` `<Buffer> | <Uint8Array>` A `Buffer` or `Uint8Array` with which to compare `buf`.
- `targetStart` `<integer>` The offset within `target` at which to begin comparison. **Default:** `0`.
- `targetEnd` `<integer>` The offset within `target` at which to end comparison (not inclusive). **Default:** `target.length`.
- `sourceStart` `<integer>` The offset within `buf` at which to begin comparison. **Default:** `0`.
- `sourceEnd` `<integer>` The offset within `buf` at which to end comparison (not inclusive). **Default:** `buf.length`.
- Returns: `<integer>`

Compares `buf` with `target` and returns a number indicating whether `buf` comes before, after, or is the same as `target` in sort order. Comparison is based on the actual sequence of bytes in each `Buffer`.

- `0` is returned if `target` is the same as `buf`
- `1` is returned if `target` should come *before* `buf` when sorted.

- `-1` is returned if `target` should come *after* `buf` when sorted.

```
import { Buffer } from 'buffer';

const buf1 = Buffer.from('ABC');
const buf2 = Buffer.from('BCD');
const buf3 = Buffer.from('ABCD');

console.log(buf1.compare(buf1));
// Prints: 0
console.log(buf1.compare(buf2));
// Prints: -1
console.log(buf1.compare(buf3));
// Prints: -1
console.log(buf2.compare(buf1));
// Prints: 1
console.log(buf2.compare(buf3));
// Prints: 1
console.log([buf1, buf2, buf3].sort(Buffer.compare));
// Prints: [ <Buffer 41 42 43>, <Buffer 41 42 43 44>, <Buffer 42 43 44> ]
// (This result is equal to: [buf1, buf3, buf2].)const { Buffer } = require('buffer');

const buf1 = Buffer.from('ABC');
const buf2 = Buffer.from('BCD');
const buf3 = Buffer.from('ABCD');

console.log(buf1.compare(buf1));
// Prints: 0
console.log(buf1.compare(buf2));
// Prints: -1
console.log(buf1.compare(buf3));
// Prints: -1
console.log(buf2.compare(buf1));
// Prints: 1
console.log(buf2.compare(buf3));
// Prints: 1
console.log([buf1, buf2, buf3].sort(Buffer.compare));
// Prints: [ <Buffer 41 42 43>, <Buffer 41 42 43 44>, <Buffer 42 43 44> ]
// (This result is equal to: [buf1, buf3, buf2].)
```

The optional `targetStart`, `targetEnd`, `sourceStart`, and `sourceEnd` arguments can be used to limit the comparison to specific ranges within `target` and `buf` respectively.

```
import { Buffer } from 'buffer';

const buf1 = Buffer.from([1, 2, 3, 4, 5, 6, 7, 8, 9]);
const buf2 = Buffer.from([5, 6, 7, 8, 9, 1, 2, 3, 4]);

console.log(buf1.compare(buf2, 5, 9, 0, 4));
// Prints: 0
```

```

console.log(buf1.compare(buf2, 0, 6, 4));
// Prints: -1
console.log(buf1.compare(buf2, 5, 6, 5));
// Prints: 1const { Buffer } = require('buffer');

const buf1 = Buffer.from([1, 2, 3, 4, 5, 6, 7, 8, 9]);
const buf2 = Buffer.from([5, 6, 7, 8, 9, 1, 2, 3, 4]);

console.log(buf1.compare(buf2, 5, 9, 0, 4));
// Prints: 0
console.log(buf1.compare(buf2, 0, 6, 4));
// Prints: -1
console.log(buf1.compare(buf2, 5, 6, 5));
// Prints: 1

```

`ERR_OUT_OF_RANGE` is thrown if `targetStart < 0`, `sourceStart < 0`, `targetEnd > target.byteLength`, or `sourceEnd > source.byteLength`.

buf.copy(target[, targetStart[, sourceStart[, sourceEnd]]])

- `target` `<Buffer>` | `<Uint8Array>` A `Buffer` or `Uint8Array` to copy into.
- `targetStart` `<integer>` The offset within `target` at which to begin writing. **Default:** `0`.
- `sourceStart` `<integer>` The offset within `buf` from which to begin copying. **Default:** `0`.
- `sourceEnd` `<integer>` The offset within `buf` at which to stop copying (not inclusive). **Default:** `buf.length`.
- Returns: `<integer>` The number of bytes copied.

Copies data from a region of `buf` to a region in `target`, even if the `target` memory region overlaps with `buf`.

`TypedArray.prototype.set()` performs the same operation, and is available for all TypedArrays, including Node.js `Buffer`s, although it takes different function arguments.

```

import { Buffer } from 'buffer';

// Create two `Buffer` instances.
const buf1 = Buffer.allocUnsafe(26);
const buf2 = Buffer.allocUnsafe(26).fill('!');

for (let i = 0; i < 26; i++) {
  // 97 is the decimal ASCII value for 'a'.
  buf1[i] = i + 97;
}

// Copy `buf1` bytes 16 through 19 into `buf2` starting at byte 8 of `buf2`.
buf1.copy(buf2, 8, 16, 20);
// This is equivalent to:
// buf2.set(buf1.subarray(16, 20), 8);

console.log(buf2.toString('ascii', 0, 25));
// Prints: !!!!!!!!!qrst!!!!!!!!!!!!const { Buffer } = require('buffer');

// Create two `Buffer` instances.

```

```

const buf1 = Buffer.allocUnsafe(26);
const buf2 = Buffer.allocUnsafe(26).fill('!');

for (let i = 0; i < 26; i++) {
  // 97 is the decimal ASCII value for 'a'.
  buf1[i] = i + 97;
}

// Copy `buf1` bytes 16 through 19 into `buf2` starting at byte 8 of `buf2`.
buf1.copy(buf2, 8, 16, 20);
// This is equivalent to:
// buf2.set(buf1.subarray(16, 20), 8);

console.log(buf2.toString('ascii', 0, 25));
// Prints: !!!!!!!!!qrst!!!!!!!!!!!!

```

```

import { Buffer } from 'buffer';

// Create a `Buffer` and copy data from one region to an overlapping region
// within the same `Buffer`.

const buf = Buffer.allocUnsafe(26);

for (let i = 0; i < 26; i++) {
  // 97 is the decimal ASCII value for 'a'.
  buf[i] = i + 97;
}

buf.copy(buf, 0, 4, 10);

console.log(buf.toString());
// Prints: efghijghijklmnopqrstuvwxyz

// Create a `Buffer` and copy data from one region to an overlapping region
// within the same `Buffer`.

const buf = Buffer.allocUnsafe(26);

for (let i = 0; i < 26; i++) {
  // 97 is the decimal ASCII value for 'a'.
  buf[i] = i + 97;
}

buf.copy(buf, 0, 4, 10);

console.log(buf.toString());
// Prints: efghijghijklmnopqrstuvwxyz

```

buf.entries()

- Returns: `<Iterator>`

Creates and returns an `iterator` of `[index, byte]` pairs from the contents of `buf`.

```
import { Buffer } from 'buffer';

// Log the entire contents of a `Buffer`.

const buf = Buffer.from('buffer');

for (const pair of buf.entries()) {
  console.log(pair);
}

// Prints:
//   [0, 98]
//   [1, 117]
//   [2, 102]
//   [3, 102]
//   [4, 101]
//   [5, 114]const { Buffer } = require('buffer');

// Log the entire contents of a `Buffer`.

const buf = Buffer.from('buffer');

for (const pair of buf.entries()) {
  console.log(pair);
}

// Prints:
//   [0, 98]
//   [1, 117]
//   [2, 102]
//   [3, 102]
//   [4, 101]
//   [5, 114]
```

buf.equals(otherBuffer)

- `otherBuffer` `<Buffer>` | `<Uint8Array>` A `Buffer` or `Uint8Array` with which to compare `buf`.
- Returns: `<boolean>`

Returns `true` if both `buf` and `otherBuffer` have exactly the same bytes, `false` otherwise. Equivalent to `buf.compare(otherBuffer) === 0`.

```
import { Buffer } from 'buffer';

const buf1 = Buffer.from('ABC');
const buf2 = Buffer.from('414243', 'hex');
const buf3 = Buffer.from('ABCD');

console.log(buf1.equals(buf2));
```

```
// Prints: true
console.log(buf1.equals(buf3));

// Prints: false
const { Buffer } = require('buffer');

const buf1 = Buffer.from('ABC');
const buf2 = Buffer.from('414243', 'hex');
const buf3 = Buffer.from('ABCD');

console.log(buf1.equals(buf2));

// Prints: true
console.log(buf1.equals(buf3));

// Prints: false
```

```
buf.fill(value[, offset[, end]][, encoding])
```

- `value` `<string>` | `<Buffer>` | `<Uint8Array>` | `<integer>` The value with which to fill `buf`.
- `offset` `<integer>` Number of bytes to skip before starting to fill `buf`. **Default:** `0`.
- `end` `<integer>` Where to stop filling `buf` (not inclusive). **Default:** `buf.length`.
- `encoding` `<string>` The encoding for `value` if `value` is a string. **Default:** `'utf8'`.
- Returns: `<Buffer>` A reference to `buf`.

Fills `buf` with the specified `value`. If the `offset` and `end` are not given, the entire `buf` will be filled:

```
import { Buffer } from 'buffer';

// Fill a `Buffer` with the ASCII character 'h'.

const b = Buffer.allocUnsafe(50).fill('h');

console.log(b.toString());

// Prints: hhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhconst { Buffer } = require('buffer');

// Fill a `Buffer` with the ASCII character 'h'.

const b = Buffer.allocUnsafe(50).fill('h');

console.log(b.toString());

// Prints: hhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhh
```

`value` is coerced to a `uint32` value if it is not a string, `Buffer`, or integer. If the resulting integer is greater than `255` (decimal), `buf` will be filled with `value & 255`.

If the final write of a `fill()` operation falls on a multi-byte character, then only the bytes of that character that fit into `buf` are written:

```
import { Buffer } from 'buffer';

// Fill a `Buffer` with character that takes up two bytes in UTF-8.

console.log(Buffer.allocUnsafe(5).fill('\u0222'));

// Prints: <Buffer c8 a2 c8 a2 c8>const { Buffer } = require('buffer');
```

```
// Fill a `Buffer` with character that takes up two bytes in UTF-8.

console.log(Buffer.allocUnsafe(5).fill('\u0222'));
// Prints: <Buffer c8 a2 c8 a2 c8>
```

If `value` contains invalid characters, it is truncated; if no valid fill data remains, an exception is thrown:

```
import { Buffer } from 'buffer';

const buf = Buffer.allocUnsafe(5);

console.log(buf.fill('a'));
// Prints: <Buffer 61 61 61 61 61>
console.log(buf.fill('aazz', 'hex'));
// Prints: <Buffer aa aa aa aa aa>
console.log(buf.fill('zz', 'hex'));
// Throws an exception.

const { Buffer } = require('buffer');

const buf = Buffer.allocUnsafe(5);

console.log(buf.fill('a'));
// Prints: <Buffer 61 61 61 61 61>
console.log(buf.fill('aazz', 'hex'));
// Prints: <Buffer aa aa aa aa aa>
console.log(buf.fill('zz', 'hex'));
// Throws an exception.
```

`buf.includes(value[, byteOffset][, encoding])`

- `value` `<string>` | `<Buffer>` | `<Uint8Array>` | `<integer>` What to search for.
- `byteOffset` `<integer>` Where to begin searching in `buf`. If negative, then offset is calculated from the end of `buf`. **Default:** `0`.
- `encoding` `<string>` If `value` is a string, this is its encoding. **Default:** `'utf8'`.
- Returns: `<boolean>` `true` if `value` was found in `buf`, `false` otherwise.

Equivalent to `buf.indexOf() !== -1`.

```
import { Buffer } from 'buffer';

const buf = Buffer.from('this is a buffer');

console.log(buf.includes('this'));
// Prints: true
console.log(buf.includes('is'));
// Prints: true
console.log(buf.includes(Buffer.from('a buffer')));
// Prints: true
console.log(buf.includes(97));
// Prints: true (97 is the decimal ASCII value for 'a')
console.log(buf.includes(Buffer.from('a buffer example')));
```



```
// Prints: false
console.log(buf.includes(Buffer.from('a buffer example').slice(0, 8)));
// Prints: true
console.log(buf.includes('this', 4));
// Prints: falseconst { Buffer } = require('buffer');

const buf = Buffer.from('this is a buffer');

console.log(buf.includes('this'));
// Prints: true
console.log(buf.includes('is'));
// Prints: true
console.log(buf.includes(Buffer.from('a buffer')));
// Prints: true
console.log(buf.includes(97));
// Prints: true (97 is the decimal ASCII value for 'a')
console.log(buf.includes(Buffer.from('a buffer example')));
// Prints: false
console.log(buf.includes(Buffer.from('a buffer example').slice(0, 8)));
// Prints: true
console.log(buf.includes('this', 4));
// Prints: false
```

buf.indexOf(value[, byteOffset][, encoding])

- **value** `<string>` | `<Buffer>` | `<Uint8Array>` | `<integer>` What to search for.
- **byteOffset** `<integer>` Where to begin searching in `buf`. If negative, then offset is calculated from the end of `buf`. **Default:** `0`.
- **encoding** `<string>` If `value` is a string, this is the encoding used to determine the binary representation of the string that will be searched for in `buf`. **Default:** `'utf8'`.
- **Returns:** `<integer>` The index of the first occurrence of `value` in `buf`, or `-1` if `buf` does not contain `value`.

If `value` is:

- a string, `value` is interpreted according to the character encoding in `encoding`.
- a `Buffer` or `Uint8Array`, `value` will be used in its entirety. To compare a partial `Buffer`, use `buf.slice()`.
- a number, `value` will be interpreted as an unsigned 8-bit integer value between `0` and `255`.

```
import { Buffer } from 'buffer';

const buf = Buffer.from('this is a buffer');

console.log(buf.indexOf('this'));
// Prints: 0
console.log(buf.indexOf('is'));
// Prints: 2
console.log(buf.indexOf(Buffer.from('a buffer')));
// Prints: 8
console.log(buf.indexOf(97));
// Prints: 8 (97 is the decimal ASCII value for 'a')
console.log(buf.indexOf(Buffer.from('a buffer example')));
```

```

// Prints: -1
console.log(buf.indexOf(Buffer.from('a buffer example').slice(0, 8)));
// Prints: 8

const utf16Buffer = Buffer.from('\u039a\u0391\u03a3\u03a3\u0395', 'utf16le');

console.log(utf16Buffer.indexOf('\u03a3', 0, 'utf16le'));
// Prints: 4
console.log(utf16Buffer.indexOf('\u03a3', -4, 'utf16le'));
// Prints: 6const { Buffer } = require('buffer');

const buf = Buffer.from('this is a buffer');

console.log(buf.indexOf('this'));
// Prints: 0
console.log(buf.indexOf('is'));
// Prints: 2
console.log(buf.indexOf(Buffer.from('a buffer')));
// Prints: 8
console.log(buf.indexOf(97));
// Prints: 8 (97 is the decimal ASCII value for 'a')
console.log(buf.indexOf(Buffer.from('a buffer example')));
// Prints: -1
console.log(buf.indexOf(Buffer.from('a buffer example').slice(0, 8)));
// Prints: 8

const utf16Buffer = Buffer.from('\u039a\u0391\u03a3\u03a3\u0395', 'utf16le');

console.log(utf16Buffer.indexOf('\u03a3', 0, 'utf16le'));
// Prints: 4
console.log(utf16Buffer.indexOf('\u03a3', -4, 'utf16le'));
// Prints: 6

```

If `value` is not a string, number, or `Buffer`, this method will throw a `TypeError`. If `value` is a number, it will be coerced to a valid byte value, an integer between 0 and 255.

If `byteOffset` is not a number, it will be coerced to a number. If the result of coercion is `NaN` or `0`, then the entire buffer will be searched. This behavior matches `String.prototype.indexOf()`.

```

import { Buffer } from 'buffer';

const b = Buffer.from('abcdef');

// Passing a value that's a number, but not a valid byte.
// Prints: 2, equivalent to searching for 99 or 'c'.
console.log(b.indexOf(99.9));
console.log(b.indexOf(256 + 99));

// Passing a byteOffset that coerces to NaN or 0.
// Prints: 1, searching the whole buffer.
console.log(b.indexOf('b', undefined));

```

```

console.log(b.indexOf('b', {}));
console.log(b.indexOf('b', null));
console.log(b.indexOf('b', []));const { Buffer } = require('buffer');

const b = Buffer.from('abcdef');

// Passing a value that's a number, but not a valid byte.
// Prints: 2, equivalent to searching for 99 or 'c'.
console.log(b.indexOf(99.9));
console.log(b.indexOf(256 + 99));

// Passing a byteOffset that coerces to NaN or 0.
// Prints: 1, searching the whole buffer.
console.log(b.indexOf('b', undefined));
console.log(b.indexOf('b', {}));
console.log(b.indexOf('b', null));
console.log(b.indexOf('b', []));

```

If `value` is an empty string or empty `Buffer` and `byteOffset` is less than `buf.length`, `byteOffset` will be returned. If `value` is empty and `byteOffset` is at least `buf.length`, `buf.length` will be returned.

buf.keys()

- Returns: `<Iterator>`

Creates and returns an `iterator` of `buf` keys (indices).

```

import { Buffer } from 'buffer';

const buf = Buffer.from('buffer');

for (const key of buf.keys()) {
  console.log(key);
}
// Prints:
//  0
//  1
//  2
//  3
//  4
//  5const { Buffer } = require('buffer');

const buf = Buffer.from('buffer');

for (const key of buf.keys()) {
  console.log(key);
}
// Prints:
//  0
//  1
//  2
//  3

```

```
// 4
// 5
```

buf.lastIndexOf(value[, byteOffset][, encoding])

- **value** `<string>` | `<Buffer>` | `<Uint8Array>` | `<integer>` What to search for.
- **byteOffset** `<integer>` Where to begin searching in `buf`. If negative, then offset is calculated from the end of `buf`. **Default:** `buf.length - 1`.
- **encoding** `<string>` If `value` is a string, this is the encoding used to determine the binary representation of the string that will be searched for in `buf`. **Default:** `'utf8'`.
- **Returns:** `<integer>` The index of the last occurrence of `value` in `buf`, or `-1` if `buf` does not contain `value`.

Identical to `buf.indexOf()`, except the last occurrence of `value` is found rather than the first occurrence.

```
import { Buffer } from 'buffer';

const buf = Buffer.from('this buffer is a buffer');

console.log(buf.lastIndexOf('this'));
// Prints: 0
console.log(buf.lastIndexOf('buffer'));
// Prints: 17
console.log(buf.lastIndexOf(Buffer.from('buffer')));
// Prints: 17
console.log(buf.lastIndexOf(97));
// Prints: 15 (97 is the decimal ASCII value for 'a')
console.log(buf.lastIndexOf(Buffer.from('yolo')));
// Prints: -1
console.log(buf.lastIndexOf('buffer', 5));
// Prints: 5
console.log(buf.lastIndexOf('buffer', 4));
// Prints: -1

const utf16Buffer = Buffer.from('\u039a\u0391\u03a3\u03a5', 'utf16le');

console.log(utf16Buffer.lastIndexOf('\u03a3', undefined, 'utf16le'));
// Prints: 6
console.log(utf16Buffer.lastIndexOf('\u03a3', -5, 'utf16le'));
// Prints: 4const { Buffer } = require('buffer');

const buf = Buffer.from('this buffer is a buffer');

console.log(buf.lastIndexOf('this'));
// Prints: 0
console.log(buf.lastIndexOf('buffer'));
// Prints: 17
console.log(buf.lastIndexOf(Buffer.from('buffer')));
// Prints: 17
console.log(buf.lastIndexOf(97));
// Prints: 15 (97 is the decimal ASCII value for 'a')
console.log(buf.lastIndexOf(Buffer.from('yolo')));
```

```
// Prints: -1
console.log(buf.lastIndexOf('buffer', 5));
// Prints: 5
console.log(buf.lastIndexOf('buffer', 4));
// Prints: -1

const utf16Buffer = Buffer.from('\u039a\u0391\u0391\u0391\u0395', 'utf16le');

console.log(utf16Buffer.lastIndexOf('\u03a3', undefined, 'utf16le'));
// Prints: 6
console.log(utf16Buffer.lastIndexOf('\u03a3', -5, 'utf16le'));
// Prints: 4
```

If `value` is not a string, number, or `Buffer`, this method will throw a `TypeError`. If `value` is a number, it will be coerced to a valid byte value, an integer between 0 and 255.

If `byteOffset` is not a number, it will be coerced to a number. Any arguments that coerce to `NaN`, like `{}` or `undefined`, will search the whole buffer. This behavior matches `String.prototype.lastIndexOf()`.

```
import { Buffer } from 'buffer';

const b = Buffer.from('abcdef');

// Passing a value that's a number, but not a valid byte.
// Prints: 2, equivalent to searching for 99 or 'c'.
console.log(b.lastIndexOf(99.9));
console.log(b.lastIndexOf(256 + 99));

// Passing a byteOffset that coerces to NaN.
// Prints: 1, searching the whole buffer.
console.log(b.lastIndexOf('b', undefined));
console.log(b.lastIndexOf('b', {}));

// Passing a byteOffset that coerces to 0.
// Prints: -1, equivalent to passing 0.
console.log(b.lastIndexOf('b', null));
console.log(b.lastIndexOf('b', []));const { Buffer } = require('buffer');

const b = Buffer.from('abcdef');

// Passing a value that's a number, but not a valid byte.
// Prints: 2, equivalent to searching for 99 or 'c'.
console.log(b.lastIndexOf(99.9));
console.log(b.lastIndexOf(256 + 99));

// Passing a byteOffset that coerces to NaN.
// Prints: 1, searching the whole buffer.
console.log(b.lastIndexOf('b', undefined));
console.log(b.lastIndexOf('b', {}));

// Passing a byteOffset that coerces to 0.
```

```
// Prints: -1, equivalent to passing 0.
console.log(b.lastIndexOf('b', null));
console.log(b.lastIndexOf('b', []));
```

If `value` is an empty string or empty `Buffer`, `byteOffset` will be returned.

buf.length

- `<integer>`

Returns the number of bytes in `buf`.

```
import { Buffer } from 'buffer';

// Create a `Buffer` and write a shorter string to it using UTF-8.

const buf = Buffer.alloc(1234);

console.log(buf.length);
// Prints: 1234

buf.write('some string', 0, 'utf8');

console.log(buf.length);
// Prints: 1234const { Buffer } = require('buffer');

// Create a `Buffer` and write a shorter string to it using UTF-8.

const buf = Buffer.alloc(1234);

console.log(buf.length);
// Prints: 1234

buf.write('some string', 0, 'utf8');

console.log(buf.length);
// Prints: 1234
```

buf.parent

Stability: 0 - Deprecated: Use `buf.buffer` instead.

The `buf.parent` property is a deprecated alias for `buf.buffer`.

buf.readBigInt64BE([offset])

- `offset` `<integer>` Number of bytes to skip before starting to read. Must satisfy: `0 <= offset <= buf.length - 8`. Default: `0`.
- Returns: `<bigint>`

Reads a signed, big-endian 64-bit integer from `buf` at the specified `offset`.

Integers read from a `Buffer` are interpreted as two's complement signed values.

`buf.readInt64LE([offset])`

- `offset` `<integer>` Number of bytes to skip before starting to read. Must satisfy: `0 <= offset <= buf.length - 8`. Default: `0`.
- Returns: `<bigint>`

Reads a signed, little-endian 64-bit integer from `buf` at the specified `offset`.

Integers read from a `Buffer` are interpreted as two's complement signed values.

`buf.readBigUint64BE([offset])`

- `offset` `<integer>` Number of bytes to skip before starting to read. Must satisfy: `0 <= offset <= buf.length - 8`. Default: `0`.
- Returns: `<bigint>`

Reads an unsigned, big-endian 64-bit integer from `buf` at the specified `offset`.

This function is also available under the `readBigUint64BE` alias.

```
import { Buffer } from 'buffer';

const buf = Buffer.from([0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff]);

console.log(buf.readBigUint64BE(0));
// Prints: 4294967295nconst { Buffer } = require('buffer');

const buf = Buffer.from([0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff]);

console.log(buf.readBigUint64BE(0));
// Prints: 4294967295n
```

`buf.readBigUint64LE([offset])`

- `offset` `<integer>` Number of bytes to skip before starting to read. Must satisfy: `0 <= offset <= buf.length - 8`. Default: `0`.
- Returns: `<bigint>`

Reads an unsigned, little-endian 64-bit integer from `buf` at the specified `offset`.

This function is also available under the `readBigUint64LE` alias.

```
import { Buffer } from 'buffer';

const buf = Buffer.from([0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff]);

console.log(buf.readBigUint64LE(0));
// Prints: 18446744069414584320nconst { Buffer } = require('buffer');

const buf = Buffer.from([0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff]);

console.log(buf.readBigUint64LE(0));
// Prints: 18446744069414584320n
```

buf.readDoubleBE([offset])

- **offset** `<integer>` Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - 8`. Default: `0`.
- Returns: `<number>`

Reads a 64-bit, big-endian double from `buf` at the specified `offset`.

```
import { Buffer } from 'buffer';

const buf = Buffer.from([1, 2, 3, 4, 5, 6, 7, 8]);

console.log(buf.readDoubleBE(0));
// Prints: 8.20788039913184e-304const { Buffer } = require('buffer');

const buf = Buffer.from([1, 2, 3, 4, 5, 6, 7, 8]);

console.log(buf.readDoubleBE(0));
// Prints: 8.20788039913184e-304
```

buf.readDoubleLE([offset])

- **offset** `<integer>` Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - 8`. Default: `0`.
- Returns: `<number>`

Reads a 64-bit, little-endian double from `buf` at the specified `offset`.

```
import { Buffer } from 'buffer';

const buf = Buffer.from([1, 2, 3, 4, 5, 6, 7, 8]);

console.log(buf.readDoubleLE(0));
// Prints: 5.447603722011605e-270
console.log(buf.readDoubleLE(1));
// Throws ERR_OUT_OF_RANGE.const { Buffer } = require('buffer');

const buf = Buffer.from([1, 2, 3, 4, 5, 6, 7, 8]);

console.log(buf.readDoubleLE(0));
// Prints: 5.447603722011605e-270
console.log(buf.readDoubleLE(1));
// Throws ERR_OUT_OF_RANGE.
```

buf.readFloatBE([offset])

- **offset** `<integer>` Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - 4`. Default: `0`.
- Returns: `<number>`

Reads a 32-bit, big-endian float from `buf` at the specified `offset`.

```
import { Buffer } from 'buffer';
```



```
const buf = Buffer.from([1, 2, 3, 4]);

console.log(buf.readFloatBE(0));
// Prints: 2.387939260590663e-38const { Buffer } = require('buffer');

const buf = Buffer.from([1, 2, 3, 4]);

console.log(buf.readFloatBE(0));
// Prints: 2.387939260590663e-38
```

buf.readFloatLE([offset])

- **offset** `<integer>` Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - 4`. **Default:** `0`.
- **Returns:** `<number>`

Reads a 32-bit, little-endian float from `buf` at the specified `offset`.

```
import { Buffer } from 'buffer';

const buf = Buffer.from([1, 2, 3, 4]);

console.log(buf.readFloatLE(0));
// Prints: 1.539989614439558e-36
console.log(buf.readFloatLE(1));
// Throws ERR_OUT_OF_RANGE.const { Buffer } = require('buffer');

const buf = Buffer.from([1, 2, 3, 4]);

console.log(buf.readFloatLE(0));
// Prints: 1.539989614439558e-36
console.log(buf.readFloatLE(1));
// Throws ERR_OUT_OF_RANGE.
```

buf.readInt8([offset])

- **offset** `<integer>` Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - 1`. **Default:** `0`.
- **Returns:** `<integer>`

Reads a signed 8-bit integer from `buf` at the specified `offset`.

Integers read from a `Buffer` are interpreted as two's complement signed values.

```
import { Buffer } from 'buffer';

const buf = Buffer.from([-1, 5]);

console.log(buf.readInt8(0));
// Prints: -1
console.log(buf.readInt8(1));
// Prints: 5
console.log(buf.readInt8(2));
```

```
// Throws ERR_OUT_OF_RANGE.const { Buffer } = require('buffer');

const buf = Buffer.from([-1, 5]);

console.log(buf.readInt8(0));
// Prints: -1
console.log(buf.readInt8(1));
// Prints: 5
console.log(buf.readInt8(2));
// Throws ERR_OUT_OF_RANGE.
```

buf.readInt16BE([offset])

- **offset** `<integer>` Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - 2`. Default: `0`.
- Returns: `<integer>`

Reads a signed, big-endian 16-bit integer from `buf` at the specified `offset`.

Integers read from a `Buffer` are interpreted as two's complement signed values.

```
import { Buffer } from 'buffer';

const buf = Buffer.from([0, 5]);

console.log(buf.readInt16BE(0));
// Prints: 5const { Buffer } = require('buffer');

const buf = Buffer.from([0, 5]);

console.log(buf.readInt16BE(0));
// Prints: 5
```

buf.readInt16LE([offset])

- **offset** `<integer>` Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - 2`. Default: `0`.
- Returns: `<integer>`

Reads a signed, little-endian 16-bit integer from `buf` at the specified `offset`.

Integers read from a `Buffer` are interpreted as two's complement signed values.

```
import { Buffer } from 'buffer';

const buf = Buffer.from([0, 5]);

console.log(buf.readInt16LE(0));
// Prints: 1280
console.log(buf.readInt16LE(1));
// Throws ERR_OUT_OF_RANGE.const { Buffer } = require('buffer');

const buf = Buffer.from([0, 5]);
```

```
console.log(buf.readInt16LE(0));  
// Prints: 1280  
console.log(buf.readInt16LE(1));  
// Throws ERR_OUT_OF_RANGE.
```

buf.readInt32BE([offset])

- **offset** `<integer>` Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - 4`. Default: `0`.
- Returns: `<integer>`

Reads a signed, big-endian 32-bit integer from `buf` at the specified `offset`.

Integers read from a `Buffer` are interpreted as two's complement signed values.

```
import { Buffer } from 'buffer';  
  
const buf = Buffer.from([0, 0, 0, 5]);  
  
console.log(buf.readInt32BE(0));  
// Prints: 5  
const { Buffer } = require('buffer');  
  
const buf = Buffer.from([0, 0, 0, 5]);  
  
console.log(buf.readInt32BE(0));  
// Prints: 5
```

buf.readInt32LE([offset])

- **offset** `<integer>` Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - 4`. Default: `0`.
- Returns: `<integer>`

Reads a signed, little-endian 32-bit integer from `buf` at the specified `offset`.

Integers read from a `Buffer` are interpreted as two's complement signed values.

```
import { Buffer } from 'buffer';  
  
const buf = Buffer.from([0, 0, 0, 5]);  
  
console.log(buf.readInt32LE(0));  
// Prints: 83886080  
console.log(buf.readInt32LE(1));  
// Throws ERR_OUT_OF_RANGE.  
const { Buffer } = require('buffer');  
  
const buf = Buffer.from([0, 0, 0, 5]);  
  
console.log(buf.readInt32LE(0));  
// Prints: 83886080  
console.log(buf.readInt32LE(1));  
// Throws ERR_OUT_OF_RANGE.
```

buf.readIntBE(offset, byteLength)

- `offset` `<integer>` Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - byteLength`.
- `byteLength` `<integer>` Number of bytes to read. Must satisfy `0 < byteLength <= 6`.
- Returns: `<integer>`

Reads `byteLength` number of bytes from `buf` at the specified `offset` and interprets the result as a big-endian, two's complement signed value supporting up to 48 bits of accuracy.

```
import { Buffer } from 'buffer';

const buf = Buffer.from([0x12, 0x34, 0x56, 0x78, 0x90, 0xab]);

console.log(buf.readIntBE(0, 6).toString(16));
// Prints: 1234567890ab
console.log(buf.readIntBE(1, 6).toString(16));
// Throws ERR_OUT_OF_RANGE.
console.log(buf.readIntBE(1, 0).toString(16));
// Throws ERR_OUT_OF_RANGE.const { Buffer } = require('buffer');

const buf = Buffer.from([0x12, 0x34, 0x56, 0x78, 0x90, 0xab]);

console.log(buf.readIntBE(0, 6).toString(16));
// Prints: 1234567890ab
console.log(buf.readIntBE(1, 6).toString(16));
// Throws ERR_OUT_OF_RANGE.
console.log(buf.readIntBE(1, 0).toString(16));
// Throws ERR_OUT_OF_RANGE.
```

buf.readIntLE(offset, byteLength)

- `offset` `<integer>` Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - byteLength`.
- `byteLength` `<integer>` Number of bytes to read. Must satisfy `0 < byteLength <= 6`.
- Returns: `<integer>`

Reads `byteLength` number of bytes from `buf` at the specified `offset` and interprets the result as a little-endian, two's complement signed value supporting up to 48 bits of accuracy.

```
import { Buffer } from 'buffer';

const buf = Buffer.from([0x12, 0x34, 0x56, 0x78, 0x90, 0xab]);

console.log(buf.readIntLE(0, 6).toString(16));
// Prints: -546f87a9cbeeconst { Buffer } = require('buffer');

const buf = Buffer.from([0x12, 0x34, 0x56, 0x78, 0x90, 0xab]);

console.log(buf.readIntLE(0, 6).toString(16));
// Prints: -546f87a9cbee
```

buf.readUInt8([offset])

- **offset** `<integer>` Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - 1`. Default: `0`.
- Returns: `<integer>`

Reads an unsigned 8-bit integer from `buf` at the specified `offset`.

This function is also available under the `readUInt8` alias.

```
import { Buffer } from 'buffer';

const buf = Buffer.from([1, -2]);

console.log(buf.readUInt8(0));
// Prints: 1
console.log(buf.readUInt8(1));
// Prints: 254
console.log(buf.readUInt8(2));
// Throws ERR_OUT_OF_RANGE.const { Buffer } = require('buffer');

const buf = Buffer.from([1, -2]);

console.log(buf.readUInt8(0));
// Prints: 1
console.log(buf.readUInt8(1));
// Prints: 254
console.log(buf.readUInt8(2));
// Throws ERR_OUT_OF_RANGE.
```

buf.readUInt16BE([offset])

- **offset** `<integer>` Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - 2`. Default: `0`.
- Returns: `<integer>`

Reads an unsigned, big-endian 16-bit integer from `buf` at the specified `offset`.

This function is also available under the `readUInt16BE` alias.

```
import { Buffer } from 'buffer';

const buf = Buffer.from([0x12, 0x34, 0x56]);

console.log(buf.readUInt16BE(0).toString(16));
// Prints: 1234
console.log(buf.readUInt16BE(1).toString(16));
// Prints: 3456const { Buffer } = require('buffer');

const buf = Buffer.from([0x12, 0x34, 0x56]);

console.log(buf.readUInt16BE(0).toString(16));
// Prints: 1234
```

```
console.log(buf.readUInt16BE(1).toString(16));  
// Prints: 3456
```

buf.readUInt16LE([offset])

- **offset** `<integer>` Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - 2`. Default: `0`.
- Returns: `<integer>`

Reads an unsigned, little-endian 16-bit integer from `buf` at the specified `offset`.

This function is also available under the `readUInt16LE` alias.

```
import { Buffer } from 'buffer';  
  
const buf = Buffer.from([0x12, 0x34, 0x56]);  
  
console.log(buf.readUInt16LE(0).toString(16));  
// Prints: 3412  
console.log(buf.readUInt16LE(1).toString(16));  
// Prints: 5634  
console.log(buf.readUInt16LE(2).toString(16));  
// Throws ERR_OUT_OF_RANGE.  
const { Buffer } = require('buffer');  
  
const buf = Buffer.from([0x12, 0x34, 0x56]);  
  
console.log(buf.readUInt16LE(0).toString(16));  
// Prints: 3412  
console.log(buf.readUInt16LE(1).toString(16));  
// Prints: 5634  
console.log(buf.readUInt16LE(2).toString(16));  
// Throws ERR_OUT_OF_RANGE.
```

buf.readUInt32BE([offset])

- **offset** `<integer>` Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - 4`. Default: `0`.
- Returns: `<integer>`

Reads an unsigned, big-endian 32-bit integer from `buf` at the specified `offset`.

This function is also available under the `readUInt32BE` alias.

```
import { Buffer } from 'buffer';  
  
const buf = Buffer.from([0x12, 0x34, 0x56, 0x78]);  
  
console.log(buf.readUInt32BE(0).toString(16));  
// Prints: 12345678  
const { Buffer } = require('buffer');  
  
const buf = Buffer.from([0x12, 0x34, 0x56, 0x78]);
```

```
console.log(buf.readUInt32BE(0).toString(16));  
// Prints: 12345678
```

buf.readUInt32LE([offset])

- **offset** `<integer>` Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - 4`. Default: `0`.
- Returns: `<integer>`

Reads an unsigned, little-endian 32-bit integer from `buf` at the specified `offset`.

This function is also available under the `readUInt32LE` alias.

```
import { Buffer } from 'buffer';  
  
const buf = Buffer.from([0x12, 0x34, 0x56, 0x78]);  
  
console.log(buf.readUInt32LE(0).toString(16));  
// Prints: 78563412  
console.log(buf.readUInt32LE(1).toString(16));  
// Throws ERR_OUT_OF_RANGE.  
const { Buffer } = require('buffer');  
  
const buf = Buffer.from([0x12, 0x34, 0x56, 0x78]);  
  
console.log(buf.readUInt32LE(0).toString(16));  
// Prints: 78563412  
console.log(buf.readUInt32LE(1).toString(16));  
// Throws ERR_OUT_OF_RANGE.
```

buf.readUIntBE(offset, byteLength)

- **offset** `<integer>` Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - byteLength`.
- **byteLength** `<integer>` Number of bytes to read. Must satisfy `0 < byteLength <= 6`.
- Returns: `<integer>`

Reads `byteLength` number of bytes from `buf` at the specified `offset` and interprets the result as an unsigned big-endian integer supporting up to 48 bits of accuracy.

This function is also available under the `readUIntBE` alias.

```
import { Buffer } from 'buffer';  
  
const buf = Buffer.from([0x12, 0x34, 0x56, 0x78, 0x90, 0xab]);  
  
console.log(buf.readUIntBE(0, 6).toString(16));  
// Prints: 1234567890ab  
console.log(buf.readUIntBE(1, 6).toString(16));  
// Throws ERR_OUT_OF_RANGE.  
const { Buffer } = require('buffer');  
  
const buf = Buffer.from([0x12, 0x34, 0x56, 0x78, 0x90, 0xab]);  
  
console.log(buf.readUIntBE(0, 6).toString(16));
```

```
// Prints: 1234567890ab
console.log(buf.readUIntBE(1, 6).toString(16));
// Throws ERR_OUT_OF_RANGE.
```

buf.readUIntLE(offset, byteLength)

- **offset** `<integer>` Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - byteLength`.
- **byteLength** `<integer>` Number of bytes to read. Must satisfy `0 < byteLength <= 6`.
- Returns: `<integer>`

Reads `byteLength` number of bytes from `buf` at the specified `offset` and interprets the result as an unsigned, little-endian integer supporting up to 48 bits of accuracy.

This function is also available under the `readUIntLE` alias.

```
import { Buffer } from 'buffer';

const buf = Buffer.from([0x12, 0x34, 0x56, 0x78, 0x90, 0xab]);

console.log(buf.readUIntLE(0, 6).toString(16));
// Prints: ab9078563412const { Buffer } = require('buffer');

const buf = Buffer.from([0x12, 0x34, 0x56, 0x78, 0x90, 0xab]);

console.log(buf.readUIntLE(0, 6).toString(16));
// Prints: ab9078563412
```

buf.subarray([start[, end]])

- **start** `<integer>` Where the new `Buffer` will start. **Default:** `0`.
- **end** `<integer>` Where the new `Buffer` will end (not inclusive). **Default:** `buf.length`.
- Returns: `<Buffer>`

Returns a new `Buffer` that references the same memory as the original, but offset and cropped by the `start` and `end` indices.

Specifying `end` greater than `buf.length` will return the same result as that of `end` equal to `buf.length`.

This method is inherited from `TypedArray.prototype.subarray()`.

Modifying the new `Buffer` slice will modify the memory in the original `Buffer` because the allocated memory of the two objects overlap.

```
import { Buffer } from 'buffer';

// Create a `Buffer` with the ASCII alphabet, take a slice, and modify one byte
// from the original `Buffer`.

const buf1 = Buffer.allocUnsafe(26);

for (let i = 0; i < 26; i++) {
  // 97 is the decimal ASCII value for 'a'.
  buf1[i] = i + 97;
```



```

}

const buf2 = buf1.subarray(0, 3);

console.log(buf2.toString('ascii', 0, buf2.length));
// Prints: abc

buf1[0] = 33;

console.log(buf2.toString('ascii', 0, buf2.length));
// Prints: !bc

const { Buffer } = require('buffer');

// Create a `Buffer` with the ASCII alphabet, take a slice, and modify one byte
// from the original `Buffer`.

const buf1 = Buffer.allocUnsafe(26);

for (let i = 0; i < 26; i++) {
  // 97 is the decimal ASCII value for 'a'.
  buf1[i] = i + 97;
}

const buf2 = buf1.subarray(0, 3);

console.log(buf2.toString('ascii', 0, buf2.length));
// Prints: abc

buf1[0] = 33;

console.log(buf2.toString('ascii', 0, buf2.length));
// Prints: !bc

```

Specifying negative indexes causes the slice to be generated relative to the end of `buf` rather than the beginning.

```

import { Buffer } from 'buffer';

const buf = Buffer.from('buffer');

console.log(buf.subarray(-6, -1).toString());
// Prints: buffe
// (Equivalent to buf.subarray(0, 5).)

console.log(buf.subarray(-6, -2).toString());
// Prints: buff
// (Equivalent to buf.subarray(0, 4).)

console.log(buf.subarray(-5, -2).toString());
// Prints: uff
// (Equivalent to buf.subarray(1, 4).)

const { Buffer } = require('buffer');

```

```

const buf = Buffer.from('buffer');

console.log(buf.subarray(-6, -1).toString());
// Prints: buffe
// (Equivalent to buf.subarray(0, 5).)

console.log(buf.subarray(-6, -2).toString());
// Prints: buff
// (Equivalent to buf.subarray(0, 4).)

console.log(buf.subarray(-5, -2).toString());
// Prints: uff
// (Equivalent to buf.subarray(1, 4).)

```

buf.slice([start[, end]])

- **start** `<integer>` Where the new `Buffer` will start. **Default:** `0`.
- **end** `<integer>` Where the new `Buffer` will end (not inclusive). **Default:** `buf.length`.
- Returns: `<Buffer>`

Returns a new `Buffer` that references the same memory as the original, but offset and cropped by the `start` and `end` indices.

This is the same behavior as `buf.subarray()`.

This method is not compatible with the `Uint8Array.prototype.slice()`, which is a superclass of `Buffer`. To copy the slice, use `Uint8Array.prototype.slice()`.

```

import { Buffer } from 'buffer';

const buf = Buffer.from('buffer');

const copiedBuf = Uint8Array.prototype.slice.call(buf);
copiedBuf[0]++;
console.log(copiedBuf.toString());
// Prints: cuffer

console.log(buf.toString());
// Prints: bufferconst { Buffer } = require('buffer');

const buf = Buffer.from('buffer');

const copiedBuf = Uint8Array.prototype.slice.call(buf);
copiedBuf[0]++;
console.log(copiedBuf.toString());
// Prints: cuffer

console.log(buf.toString());
// Prints: buffer

```

buf.swap16()

- Returns: `<Buffer>` A reference to `buf`.

Interprets `buf` as an array of unsigned 16-bit integers and swaps the byte order *in-place*. Throws `ERR_INVALID_BUFFER_SIZE` if `buf.length` is not a multiple of 2.

```
import { Buffer } from 'buffer';

const buf1 = Buffer.from([0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8]);

console.log(buf1);
// Prints: <Buffer 01 02 03 04 05 06 07 08>

buf1.swap16();

console.log(buf1);
// Prints: <Buffer 02 01 04 03 06 05 08 07>

const buf2 = Buffer.from([0x1, 0x2, 0x3]);

buf2.swap16();
// Throws ERR_INVALID_BUFFER_SIZE.

const { Buffer } = require('buffer');

const buf1 = Buffer.from([0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8]);

console.log(buf1);
// Prints: <Buffer 01 02 03 04 05 06 07 08>

buf1.swap16();

console.log(buf1);
// Prints: <Buffer 02 01 04 03 06 05 08 07>

const buf2 = Buffer.from([0x1, 0x2, 0x3]);

buf2.swap16();
// Throws ERR_INVALID_BUFFER_SIZE.
```

One convenient use of `buf.swap16()` is to perform a fast in-place conversion between UTF-16 little-endian and UTF-16 big-endian:

```
import { Buffer } from 'buffer';

const buf = Buffer.from('This is little-endian UTF-16', 'utf16le');
buf.swap16(); // Convert to big-endian UTF-16 text.

const { Buffer } = require('buffer');

const buf = Buffer.from('This is little-endian UTF-16', 'utf16le');
buf.swap16(); // Convert to big-endian UTF-16 text.
```

buf.swap32()

- Returns: `<Buffer>` A reference to `buf`.

Interprets `buf` as an array of unsigned 32-bit integers and swaps the byte order *in-place*. Throws `ERR_INVALID_BUFFER_SIZE` if `buf.length` is not a multiple of 4.

```
import { Buffer } from 'buffer';

const buf1 = Buffer.from([0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8]);

console.log(buf1);
// Prints: <Buffer 01 02 03 04 05 06 07 08>

buf1.swap32();

console.log(buf1);
// Prints: <Buffer 04 03 02 01 08 07 06 05>

const buf2 = Buffer.from([0x1, 0x2, 0x3]);

buf2.swap32();
// Throws ERR_INVALID_BUFFER_SIZE.

const buf1 = Buffer.from([0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8]);

console.log(buf1);
// Prints: <Buffer 01 02 03 04 05 06 07 08>

buf1.swap32();

console.log(buf1);
// Prints: <Buffer 04 03 02 01 08 07 06 05>

const buf2 = Buffer.from([0x1, 0x2, 0x3]);

buf2.swap32();
// Throws ERR_INVALID_BUFFER_SIZE.
```

buf.swap64()

- Returns: `<Buffer>` A reference to `buf`.

Interprets `buf` as an array of 64-bit numbers and swaps byte order *in-place*. Throws `ERR_INVALID_BUFFER_SIZE` if `buf.length` is not a multiple of 8.

```
import { Buffer } from 'buffer';

const buf1 = Buffer.from([0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8]);

console.log(buf1);
// Prints: <Buffer 01 02 03 04 05 06 07 08>

buf1.swap64();
```

```

console.log(buf1);
// Prints: <Buffer 08 07 06 05 04 03 02 01>

const buf2 = Buffer.from([0x1, 0x2, 0x3]);

buf2.swap64();
// Throws ERR_INVALID_BUFFER_SIZE.

const buf1 = Buffer.from([0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8]);

console.log(buf1);
// Prints: <Buffer 01 02 03 04 05 06 07 08>

buf1.swap64();

console.log(buf1);
// Prints: <Buffer 08 07 06 05 04 03 02 01>

const buf2 = Buffer.from([0x1, 0x2, 0x3]);

buf2.swap64();
// Throws ERR_INVALID_BUFFER_SIZE.

```

buf.toJSON()

- Returns: `<Object>`

Returns a JSON representation of `buf`. `JSON.stringify()` implicitly calls this function when stringifying a `Buffer` instance.

`Buffer.from()` accepts objects in the format returned from this method. In particular, `Buffer.from(buf.toJSON())` works like `Buffer.from(buf)`.

```

import { Buffer } from 'buffer';

const buf = Buffer.from([0x1, 0x2, 0x3, 0x4, 0x5]);
const json = JSON.stringify(buf);

console.log(json);
// Prints: {"type":"Buffer","data":[1,2,3,4,5]}

const copy = JSON.parse(json, (key, value) => {
  return value && value.type === 'Buffer' ?
    Buffer.from(value) :
    value;
});

console.log(copy);
// Prints: <Buffer 01 02 03 04 05>

const { Buffer } = require('buffer');

const buf = Buffer.from([0x1, 0x2, 0x3, 0x4, 0x5]);
const json = JSON.stringify(buf);

```

```

console.log(json);
// Prints: {"type":"Buffer","data":[1,2,3,4,5]}

const copy = JSON.parse(json, (key, value) => {
  return value && value.type === 'Buffer' ?
    Buffer.from(value) :
    value;
});

console.log(copy);
// Prints: <Buffer 01 02 03 04 05>

```

buf.toString([encoding[, start[, end]]])

- **encoding** `<string>` The character encoding to use. **Default:** `'utf8'`.
- **start** `<integer>` The byte offset to start decoding at. **Default:** `0`.
- **end** `<integer>` The byte offset to stop decoding at (not inclusive). **Default:** `buf.length`.
- Returns: `<string>`

Decodes `buf` to a string according to the specified character encoding in `encoding`. `start` and `end` may be passed to decode only a subset of `buf`.

If `encoding` is `'utf8'` and a byte sequence in the input is not valid UTF-8, then each invalid byte is replaced with the replacement character `U+FFFD`.

The maximum length of a string instance (in UTF-16 code units) is available as `buffer.constants.MAX_STRING_LENGTH`.

```

import { Buffer } from 'buffer';

const buf1 = Buffer.allocUnsafe(26);

for (let i = 0; i < 26; i++) {
  // 97 is the decimal ASCII value for 'a'.
  buf1[i] = i + 97;
}

console.log(buf1.toString('utf8'));
// Prints: abcdefghijklmnopqrstuvwxyz
console.log(buf1.toString('utf8', 0, 5));
// Prints: abcde

const buf2 = Buffer.from('tést');

console.log(buf2.toString('hex'));
// Prints: 74c3a97374
console.log(buf2.toString('utf8', 0, 3));
// Prints: té
console.log(buf2.toString(undefined, 0, 3));
// Prints: té
const { Buffer } = require('buffer');

```

```

const buf1 = Buffer.allocUnsafe(26);

for (let i = 0; i < 26; i++) {
  // 97 is the decimal ASCII value for 'a'.
  buf1[i] = i + 97;
}

console.log(buf1.toString('utf8'));
// Prints: abcdefghijklmnopqrstuvwxyz
console.log(buf1.toString('utf8', 0, 5));
// Prints: abcde

const buf2 = Buffer.from('tést');

console.log(buf2.toString('hex'));
// Prints: 74c3a97374
console.log(buf2.toString('utf8', 0, 3));
// Prints: té
console.log(buf2.toString(undefined, 0, 3));
// Prints: té

```

buf.values()

- Returns: `<Iterator>`

Creates and returns an `iterator` for `buf` values (bytes). This function is called automatically when a `Buffer` is used in a `for..of` statement.

```

import { Buffer } from 'buffer';

const buf = Buffer.from('buffer');

for (const value of buf.values()) {
  console.log(value);
}
// Prints:
//   98
//   117
//   102
//   102
//   101
//   114

for (const value of buf) {
  console.log(value);
}
// Prints:
//   98
//   117
//   102
//   102
//   101

```

```
// 114const { Buffer } = require('buffer');

const buf = Buffer.from('buffer');

for (const value of buf.values()) {
  console.log(value);
}
// Prints:
// 98
// 117
// 102
// 102
// 101
// 114

for (const value of buf) {
  console.log(value);
}
// Prints:
// 98
// 117
// 102
// 102
// 101
// 114
```

buf.write(string[, offset[, length]][, encoding])

- **string** `<string>` String to write to `buf`.
- **offset** `<integer>` Number of bytes to skip before starting to write `string`. **Default:** `0`.
- **length** `<integer>` Maximum number of bytes to write (written bytes will not exceed `buf.length - offset`). **Default:** `buf.length - offset`.
- **encoding** `<string>` The character encoding of `string`. **Default:** `'utf8'`.
- Returns: `<integer>` Number of bytes written.

Writes `string` to `buf` at `offset` according to the character encoding in `encoding`. The `length` parameter is the number of bytes to write. If `buf` did not contain enough space to fit the entire string, only part of `string` will be written. However, partially encoded characters will not be written.

```
import { Buffer } from 'buffer';

const buf = Buffer.alloc(256);

const len = buf.write('\u00bd + \u00bc = \u00be', 0);

console.log(`${len} bytes: ${buf.toString('utf8', 0, len)}`);
// Prints: 12 bytes: ½ + ¼ = ¾

const buffer = Buffer.alloc(10);
```



```

const length = buffer.write('abcd', 8);

console.log(`${length} bytes: ${buffer.toString('utf8', 8, 10)}`);
// Prints: 2 bytes : abconst { Buffer } = require('buffer');

const buf = Buffer.alloc(256);

const len = buf.write('\u00bd + \u00bc = \u00be', 0);

console.log(`${len} bytes: ${buf.toString('utf8', 0, len)}`);
// Prints: 12 bytes: ½ + ¼ = ¾

const buffer = Buffer.alloc(10);

const length = buffer.write('abcd', 8);

console.log(`${length} bytes: ${buffer.toString('utf8', 8, 10)}`);
// Prints: 2 bytes : ab

```

buf.writeBigInt64BE(value[, offset])

- `value` `<bigint>` Number to be written to `buf`.
- `offset` `<integer>` Number of bytes to skip before starting to write. Must satisfy: `0 <= offset <= buf.length - 8`. Default: `0`.
- Returns: `<integer>` `offset` plus the number of bytes written.

Writes `value` to `buf` at the specified `offset` as big-endian.

`value` is interpreted and written as a two's complement signed integer.

```

import { Buffer } from 'buffer';

const buf = Buffer.allocUnsafe(8);

buf.writeBigInt64BE(0x0102030405060708n, 0);

console.log(buf);
// Prints: <Buffer 01 02 03 04 05 06 07 08>const { Buffer } = require('buffer');

const buf = Buffer.allocUnsafe(8);

buf.writeBigInt64BE(0x0102030405060708n, 0);

console.log(buf);
// Prints: <Buffer 01 02 03 04 05 06 07 08>

```

buf.writeBigInt64LE(value[, offset])

- `value` `<bigint>` Number to be written to `buf`.
- `offset` `<integer>` Number of bytes to skip before starting to write. Must satisfy: `0 <= offset <= buf.length - 8`. Default: `0`.
- Returns: `<integer>` `offset` plus the number of bytes written.

Writes `value` to `buf` at the specified `offset` as little-endian.

`value` is interpreted and written as a two's complement signed integer.

```
import { Buffer } from 'buffer';

const buf = Buffer.allocUnsafe(8);

buf.writeBigInt64LE(0x0102030405060708n, 0);

console.log(buf);
// Prints: <Buffer 08 07 06 05 04 03 02 01>const { Buffer } = require('buffer');

const buf = Buffer.allocUnsafe(8);

buf.writeBigInt64LE(0x0102030405060708n, 0);

console.log(buf);
// Prints: <Buffer 08 07 06 05 04 03 02 01>
```

`buf.writeBigUInt64BE(value[, offset])`

- `value` `<bigint>` Number to be written to `buf`.
- `offset` `<integer>` Number of bytes to skip before starting to write. Must satisfy: `0 <= offset <= buf.length - 8`. Default: `0`.
- Returns: `<integer>` `offset` plus the number of bytes written.

Writes `value` to `buf` at the specified `offset` as big-endian.

This function is also available under the `writeBigUInt64BE` alias.

```
import { Buffer } from 'buffer';

const buf = Buffer.allocUnsafe(8);

buf.writeBigUInt64BE(0xdecafafecefa, 0);

console.log(buf);
// Prints: <Buffer de ca fa fe ca ce fa de>const { Buffer } = require('buffer');

const buf = Buffer.allocUnsafe(8);

buf.writeBigUInt64BE(0xdecafafecefa, 0);

console.log(buf);
// Prints: <Buffer de ca fa fe ca ce fa de>
```

`buf.writeBigUInt64LE(value[, offset])`

- `value` `<bigint>` Number to be written to `buf`.
- `offset` `<integer>` Number of bytes to skip before starting to write. Must satisfy: `0 <= offset <= buf.length - 8`. Default: `0`.

- Returns: `<integer>` `offset` plus the number of bytes written.

Writes `value` to `buf` at the specified `offset` as little-endian

```
import { Buffer } from 'buffer';

const buf = Buffer.allocUnsafe(8);

buf.writeBigUInt64LE(0xdecafafecefa, 0);

console.log(buf);
// Prints: <Buffer de fa ce ca fe fa ca de>const { Buffer } = require('buffer');

const buf = Buffer.allocUnsafe(8);

buf.writeBigUInt64LE(0xdecafafecefa, 0);

console.log(buf);
// Prints: <Buffer de fa ce ca fe fa ca de>
```

This function is also available under the `writeBigUInt64LE` alias.

buf.writeDoubleBE(value[, offset])

- `value` `<number>` Number to be written to `buf`.
- `offset` `<integer>` Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - 8`. Default: `0`.
- Returns: `<integer>` `offset` plus the number of bytes written.

Writes `value` to `buf` at the specified `offset` as big-endian. The `value` must be a JavaScript number. Behavior is undefined when `value` is anything other than a JavaScript number.

```
import { Buffer } from 'buffer';

const buf = Buffer.allocUnsafe(8);

buf.writeDoubleBE(123.456, 0);

console.log(buf);
// Prints: <Buffer 40 5e dd 2f 1a 9f be 77>const { Buffer } = require('buffer');

const buf = Buffer.allocUnsafe(8);

buf.writeDoubleBE(123.456, 0);

console.log(buf);
// Prints: <Buffer 40 5e dd 2f 1a 9f be 77>
```

buf.writeDoubleLE(value[, offset])

- `value` `<number>` Number to be written to `buf`.
- `offset` `<integer>` Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - 8`. Default: `0`.

- Returns: `<integer>` `offset` plus the number of bytes written.

Writes `value` to `buf` at the specified `offset` as little-endian. The `value` must be a JavaScript number. Behavior is undefined when `value` is anything other than a JavaScript number.

```
import { Buffer } from 'buffer';

const buf = Buffer.allocUnsafe(8);

buf.writeDoubleLE(123.456, 0);

console.log(buf);
// Prints: <Buffer 77 be 9f 1a 2f dd 5e 40>const { Buffer } = require('buffer');

const buf = Buffer.allocUnsafe(8);

buf.writeDoubleLE(123.456, 0);

console.log(buf);
// Prints: <Buffer 77 be 9f 1a 2f dd 5e 40>
```

buf.writeFloatBE(value[, offset])

- `value` `<number>` Number to be written to `buf`.
- `offset` `<integer>` Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - 4`. Default: `0`.
- Returns: `<integer>` `offset` plus the number of bytes written.

Writes `value` to `buf` at the specified `offset` as big-endian. Behavior is undefined when `value` is anything other than a JavaScript number.

```
import { Buffer } from 'buffer';

const buf = Buffer.allocUnsafe(4);

buf.writeFloatBE(0xcafebabe, 0);

console.log(buf);
// Prints: <Buffer 4f 4a fe bb>const { Buffer } = require('buffer');

const buf = Buffer.allocUnsafe(4);

buf.writeFloatBE(0xcafebabe, 0);

console.log(buf);
// Prints: <Buffer 4f 4a fe bb>
```

buf.writeFloatLE(value[, offset])

- `value` `<number>` Number to be written to `buf`.
- `offset` `<integer>` Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - 4`. Default: `0`.
- Returns: `<integer>` `offset` plus the number of bytes written.

Writes `value` to `buf` at the specified `offset` as little-endian. Behavior is undefined when `value` is anything other than a JavaScript number.

```
import { Buffer } from 'buffer';

const buf = Buffer.allocUnsafe(4);

buf.writeFloatLE(0xcafebabe, 0);

console.log(buf);
// Prints: <Buffer bb fe 4a 4f>const { Buffer } = require('buffer');

const buf = Buffer.allocUnsafe(4);

buf.writeFloatLE(0xcafebabe, 0);

console.log(buf);
// Prints: <Buffer bb fe 4a 4f>
```

`buf.writeInt8(value[, offset])`

- `value` `<integer>` Number to be written to `buf`.
- `offset` `<integer>` Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - 1`. Default: `0`.
- Returns: `<integer>` `offset` plus the number of bytes written.

Writes `value` to `buf` at the specified `offset`. `value` must be a valid signed 8-bit integer. Behavior is undefined when `value` is anything other than a signed 8-bit integer.

`value` is interpreted and written as a two's complement signed integer.

```
import { Buffer } from 'buffer';

const buf = Buffer.allocUnsafe(2);

buf.writeInt8(2, 0);
buf.writeInt8(-2, 1);

console.log(buf);
// Prints: <Buffer 02 fe>const { Buffer } = require('buffer');

const buf = Buffer.allocUnsafe(2);

buf.writeInt8(2, 0);
buf.writeInt8(-2, 1);

console.log(buf);
// Prints: <Buffer 02 fe>
```

`buf.writeInt16BE(value[, offset])`

- `value` `<integer>` Number to be written to `buf`.

- `offset` `<integer>` Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - 2`. **Default:** `0`.
- Returns: `<integer>` `offset` plus the number of bytes written.

Writes `value` to `buf` at the specified `offset` as big-endian. The `value` must be a valid signed 16-bit integer. Behavior is undefined when `value` is anything other than a signed 16-bit integer.

The `value` is interpreted and written as a two's complement signed integer.

```
import { Buffer } from 'buffer';

const buf = Buffer.allocUnsafe(2);

buf.writeInt16BE(0x0102, 0);

console.log(buf);
// Prints: <Buffer 01 02>const { Buffer } = require('buffer');

const buf = Buffer.allocUnsafe(2);

buf.writeInt16BE(0x0102, 0);

console.log(buf);
// Prints: <Buffer 01 02>
```

`buf.writeInt16LE(value[, offset])`

- `value` `<integer>` Number to be written to `buf`.
- `offset` `<integer>` Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - 2`. **Default:** `0`.
- Returns: `<integer>` `offset` plus the number of bytes written.

Writes `value` to `buf` at the specified `offset` as little-endian. The `value` must be a valid signed 16-bit integer. Behavior is undefined when `value` is anything other than a signed 16-bit integer.

The `value` is interpreted and written as a two's complement signed integer.

```
import { Buffer } from 'buffer';

const buf = Buffer.allocUnsafe(2);

buf.writeInt16LE(0x0304, 0);

console.log(buf);
// Prints: <Buffer 04 03>const { Buffer } = require('buffer');

const buf = Buffer.allocUnsafe(2);

buf.writeInt16LE(0x0304, 0);

console.log(buf);
// Prints: <Buffer 04 03>
```

buf.writeInt32BE(value[, offset])

- `value` `<integer>` Number to be written to `buf`.
- `offset` `<integer>` Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - 4`. Default: `0`.
- Returns: `<integer>` `offset` plus the number of bytes written.

Writes `value` to `buf` at the specified `offset` as big-endian. The `value` must be a valid signed 32-bit integer. Behavior is undefined when `value` is anything other than a signed 32-bit integer.

The `value` is interpreted and written as a two's complement signed integer.

```
import { Buffer } from 'buffer';

const buf = Buffer.allocUnsafe(4);

buf.writeInt32BE(0x01020304, 0);

console.log(buf);
// Prints: <Buffer 01 02 03 04>const { Buffer } = require('buffer');

const buf = Buffer.allocUnsafe(4);

buf.writeInt32BE(0x01020304, 0);

console.log(buf);
// Prints: <Buffer 01 02 03 04>
```

buf.writeInt32LE(value[, offset])

- `value` `<integer>` Number to be written to `buf`.
- `offset` `<integer>` Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - 4`. Default: `0`.
- Returns: `<integer>` `offset` plus the number of bytes written.

Writes `value` to `buf` at the specified `offset` as little-endian. The `value` must be a valid signed 32-bit integer. Behavior is undefined when `value` is anything other than a signed 32-bit integer.

The `value` is interpreted and written as a two's complement signed integer.

```
import { Buffer } from 'buffer';

const buf = Buffer.allocUnsafe(4);

buf.writeInt32LE(0x05060708, 0);

console.log(buf);
// Prints: <Buffer 08 07 06 05>const { Buffer } = require('buffer');

const buf = Buffer.allocUnsafe(4);

buf.writeInt32LE(0x05060708, 0);
```

```
console.log(buf);  
// Prints: <Buffer 08 07 06 05>
```

buf.writeIntBE(value, offset, byteLength)

- `value` `<integer>` Number to be written to `buf`.
- `offset` `<integer>` Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - byteLength`.
- `byteLength` `<integer>` Number of bytes to write. Must satisfy `0 < byteLength <= 6`.
- Returns: `<integer>` `offset` plus the number of bytes written.

Writes `byteLength` bytes of `value` to `buf` at the specified `offset` as big-endian. Supports up to 48 bits of accuracy. Behavior is undefined when `value` is anything other than a signed integer.

```
import { Buffer } from 'buffer';  
  
const buf = Buffer.allocUnsafe(6);  
  
buf.writeIntBE(0x1234567890ab, 0, 6);  
  
console.log(buf);  
// Prints: <Buffer 12 34 56 78 90 ab>  
const { Buffer } = require('buffer');  
  
const buf = Buffer.allocUnsafe(6);  
  
buf.writeIntBE(0x1234567890ab, 0, 6);  
  
console.log(buf);  
// Prints: <Buffer 12 34 56 78 90 ab>
```

buf.writeIntLE(value, offset, byteLength)

- `value` `<integer>` Number to be written to `buf`.
- `offset` `<integer>` Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - byteLength`.
- `byteLength` `<integer>` Number of bytes to write. Must satisfy `0 < byteLength <= 6`.
- Returns: `<integer>` `offset` plus the number of bytes written.

Writes `byteLength` bytes of `value` to `buf` at the specified `offset` as little-endian. Supports up to 48 bits of accuracy. Behavior is undefined when `value` is anything other than a signed integer.

```
import { Buffer } from 'buffer';  
  
const buf = Buffer.allocUnsafe(6);  
  
buf.writeIntLE(0x1234567890ab, 0, 6);  
  
console.log(buf);  
// Prints: <Buffer ab 90 78 56 34 12>  
const { Buffer } = require('buffer');  
  
const buf = Buffer.allocUnsafe(6);
```



```
buf.writeIntLE(0x1234567890ab, 0, 6);

console.log(buf);
// Prints: <Buffer ab 90 78 56 34 12>
```

buf.writeUInt8(value[, offset])

- **value** `<integer>` Number to be written to `buf`.
- **offset** `<integer>` Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - 1`. Default: `0`.
- Returns: `<integer>` `offset` plus the number of bytes written.

Writes `value` to `buf` at the specified `offset`. `value` must be a valid unsigned 8-bit integer. Behavior is undefined when `value` is anything other than an unsigned 8-bit integer.

This function is also available under the `writeUInt8` alias.

```
import { Buffer } from 'buffer';

const buf = Buffer.allocUnsafe(4);

buf.writeUInt8(0x3, 0);
buf.writeUInt8(0x4, 1);
buf.writeUInt8(0x23, 2);
buf.writeUInt8(0x42, 3);

console.log(buf);
// Prints: <Buffer 03 04 23 42>const { Buffer } = require('buffer');

const buf = Buffer.allocUnsafe(4);

buf.writeUInt8(0x3, 0);
buf.writeUInt8(0x4, 1);
buf.writeUInt8(0x23, 2);
buf.writeUInt8(0x42, 3);

console.log(buf);
// Prints: <Buffer 03 04 23 42>
```

buf.writeUInt16BE(value[, offset])

- **value** `<integer>` Number to be written to `buf`.
- **offset** `<integer>` Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - 2`. Default: `0`.
- Returns: `<integer>` `offset` plus the number of bytes written.

Writes `value` to `buf` at the specified `offset` as big-endian. The `value` must be a valid unsigned 16-bit integer. Behavior is undefined when `value` is anything other than an unsigned 16-bit integer.

This function is also available under the `writeUInt16BE` alias.

```
import { Buffer } from 'buffer';
```

```

const buf = Buffer.allocUnsafe(4);

buf.writeUInt16BE(0xdead, 0);
buf.writeUInt16BE(0xbeef, 2);

console.log(buf);
// Prints: <Buffer de ad be ef>const { Buffer } = require('buffer');

const buf = Buffer.allocUnsafe(4);

buf.writeUInt16BE(0xdead, 0);
buf.writeUInt16BE(0xbeef, 2);

console.log(buf);
// Prints: <Buffer de ad be ef>

```

buf.writeUInt16LE(value[, offset])

- **value** `<integer>` Number to be written to `buf`.
- **offset** `<integer>` Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - 2`. **Default:** `0`.
- **Returns:** `<integer>` `offset` plus the number of bytes written.

Writes `value` to `buf` at the specified `offset` as little-endian. The `value` must be a valid unsigned 16-bit integer. Behavior is undefined when `value` is anything other than an unsigned 16-bit integer.

This function is also available under the `writeUInt16LE` alias.

```

import { Buffer } from 'buffer';

const buf = Buffer.allocUnsafe(4);

buf.writeUInt16LE(0xdead, 0);
buf.writeUInt16LE(0xbeef, 2);

console.log(buf);
// Prints: <Buffer ad de ef be>const { Buffer } = require('buffer');

const buf = Buffer.allocUnsafe(4);

buf.writeUInt16LE(0xdead, 0);
buf.writeUInt16LE(0xbeef, 2);

console.log(buf);
// Prints: <Buffer ad de ef be>

```

buf.writeUInt32BE(value[, offset])

- **value** `<integer>` Number to be written to `buf`.
- **offset** `<integer>` Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - 4`. **Default:** `0`.
- **Returns:** `<integer>` `offset` plus the number of bytes written.

Writes `value` to `buf` at the specified `offset` as big-endian. The `value` must be a valid unsigned 32-bit integer. Behavior is undefined when `value` is anything other than an unsigned 32-bit integer.

This function is also available under the `writeUInt32BE` alias.

```
import { Buffer } from 'buffer';

const buf = Buffer.allocUnsafe(4);

buf.writeUInt32BE(0xfeedface, 0);

console.log(buf);
// Prints: <Buffer fe ed fa ce>const { Buffer } = require('buffer');

const buf = Buffer.allocUnsafe(4);

buf.writeUInt32BE(0xfeedface, 0);

console.log(buf);
// Prints: <Buffer fe ed fa ce>
```

`buf.writeUInt32LE(value[, offset])`

- `value` `<integer>` Number to be written to `buf`.
- `offset` `<integer>` Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - 4`. Default: `0`.
- Returns: `<integer>` `offset` plus the number of bytes written.

Writes `value` to `buf` at the specified `offset` as little-endian. The `value` must be a valid unsigned 32-bit integer. Behavior is undefined when `value` is anything other than an unsigned 32-bit integer.

This function is also available under the `writeUInt32LE` alias.

```
import { Buffer } from 'buffer';

const buf = Buffer.allocUnsafe(4);

buf.writeUInt32LE(0xfeedface, 0);

console.log(buf);
// Prints: <Buffer ce fa ed fe>const { Buffer } = require('buffer');

const buf = Buffer.allocUnsafe(4);

buf.writeUInt32LE(0xfeedface, 0);

console.log(buf);
// Prints: <Buffer ce fa ed fe>
```

`buf.writeUIntBE(value, offset, byteLength)`

- `value` `<integer>` Number to be written to `buf`.

- `offset` `<integer>` Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - byteLength`.
- `byteLength` `<integer>` Number of bytes to write. Must satisfy `0 < byteLength <= 6`.
- Returns: `<integer>` `offset` plus the number of bytes written.

Writes `byteLength` bytes of `value` to `buf` at the specified `offset` as big-endian. Supports up to 48 bits of accuracy. Behavior is undefined when `value` is anything other than an unsigned integer.

This function is also available under the `writeUIntBE` alias.

```
import { Buffer } from 'buffer';

const buf = Buffer.allocUnsafe(6);

buf.writeUIntBE(0x1234567890ab, 0, 6);

console.log(buf);
// Prints: <Buffer 12 34 56 78 90 ab>const { Buffer } = require('buffer');

const buf = Buffer.allocUnsafe(6);

buf.writeUIntBE(0x1234567890ab, 0, 6);

console.log(buf);
// Prints: <Buffer 12 34 56 78 90 ab>
```

`buf.writeUIntLE(value, offset, byteLength)`

- `value` `<integer>` Number to be written to `buf`.
- `offset` `<integer>` Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - byteLength`.
- `byteLength` `<integer>` Number of bytes to write. Must satisfy `0 < byteLength <= 6`.
- Returns: `<integer>` `offset` plus the number of bytes written.

Writes `byteLength` bytes of `value` to `buf` at the specified `offset` as little-endian. Supports up to 48 bits of accuracy. Behavior is undefined when `value` is anything other than an unsigned integer.

This function is also available under the `writeUIntLE` alias.

```
import { Buffer } from 'buffer';

const buf = Buffer.allocUnsafe(6);

buf.writeUIntLE(0x1234567890ab, 0, 6);

console.log(buf);
// Prints: <Buffer ab 90 78 56 34 12>const { Buffer } = require('buffer');

const buf = Buffer.allocUnsafe(6);

buf.writeUIntLE(0x1234567890ab, 0, 6);
```

```
console.log(buf);  
// Prints: <Buffer ab 90 78 56 34 12>
```

new Buffer(array)

Stability: 0 - Deprecated: Use `Buffer.from(array)` instead.

- `array` `<integer[]>` An array of bytes to copy from.

See `Buffer.from(array)`.

new Buffer(arrayBuffer[, byteOffset[, length]])

Stability: 0 - Deprecated: Use `Buffer.from(arrayBuffer[, byteOffset[, length]])` instead.

- `arrayBuffer` `<ArrayBuffer>` | `<SharedArrayBuffer>` An `ArrayBuffer`, `SharedArrayBuffer` or the `.buffer` property of a `TypedArray`.
- `byteOffset` `<integer>` Index of first byte to expose. **Default:** `0`.
- `length` `<integer>` Number of bytes to expose. **Default:** `arrayBuffer.byteLength - byteOffset`.

See `Buffer.from(arrayBuffer[, byteOffset[, length]])`.

new Buffer(buffer)

Stability: 0 - Deprecated: Use `Buffer.from(buffer)` instead.

- `buffer` `<Buffer>` | `<Uint8Array>` An existing `Buffer` or `Uint8Array` from which to copy data.

See `Buffer.from(buffer)`.

new Buffer(size)

Stability: 0 - Deprecated: Use `Buffer.alloc()` instead (also see `Buffer.allocUnsafe()`).

- `size` `<integer>` The desired length of the new `Buffer`.

See `Buffer.alloc()` and `Buffer.allocUnsafe()`. This variant of the constructor is equivalent to `Buffer.alloc()`.

new Buffer(string[, encoding])

Stability: 0 - Deprecated: Use `Buffer.from(string[, encoding])` instead.

- `string` `<string>` String to encode.
- `encoding` `<string>` The encoding of `string`. **Default:** `'utf8'`.

See `Buffer.from(string[, encoding])`.

buffer module APIs

While, the `Buffer` object is available as a global, there are additional `Buffer`-related APIs that are available only via the `buffer` module accessed using `require('buffer')`.

buffer.atob(data)

Stability: 3 - Legacy. Use `Buffer.from(data, 'base64')` instead.

- `data` *<any>* The Base64-encoded input string.

Decodes a string of Base64-encoded data into bytes, and encodes those bytes into a string using Latin-1 (ISO-8859-1).

The `data` may be any JavaScript-value that can be coerced into a string.

This function is only provided for compatibility with legacy web platform APIs and should never be used in new code, because they use strings to represent binary data and predate the introduction of typed arrays in JavaScript. For code running using Node.js APIs, converting between base64-encoded strings and binary data should be performed using `Buffer.from(str, 'base64')` and `buf.toString('base64')`.

buffer.btoa(data)

Stability: 3 - Legacy. Use `buf.toString('base64')` instead.

- `data` *<any>* An ASCII (Latin1) string.

Decodes a string into bytes using Latin-1 (ISO-8859), and encodes those bytes into a string using Base64.

The `data` may be any JavaScript-value that can be coerced into a string.

This function is only provided for compatibility with legacy web platform APIs and should never be used in new code, because they use strings to represent binary data and predate the introduction of typed arrays in JavaScript. For code running using Node.js APIs, converting between base64-encoded strings and binary data should be performed using `Buffer.from(str, 'base64')` and `buf.toString('base64')`.

buffer.INSPECT_MAX_BYTES

- *<integer>* Default: 50

Returns the maximum number of bytes that will be returned when `buf.inspect()` is called. This can be overridden by user modules. See `util.inspect()` for more details on `buf.inspect()` behavior.

buffer.kMaxLength

- *<integer>* The largest size allowed for a single `Buffer` instance.

An alias for `buffer.constants.MAX_LENGTH`.

buffer.kStringMaxLength

- *<integer>* The largest length allowed for a single `string` instance.

An alias for `buffer.constants.MAX_STRING_LENGTH`.

buffer.resolveObjectURL(id)

Stability: 1 - Experimental

- `id` `<string>` A `'blob:nodata:...'` URL string returned by a prior call to `URL.createObjectURL()`.
- Returns: `<Blob>`

Resolves a `'blob:nodata:...'` an associated `<Blob>` object registered using a prior call to `URL.createObjectURL()`.

buffer.transcode(source, fromEnc, toEnc)

- `source` `<Buffer>` | `<Uint8Array>` A `Buffer` or `Uint8Array` instance.
- `fromEnc` `<string>` The current encoding.
- `toEnc` `<string>` To target encoding.
- Returns: `<Buffer>`

Re-encodes the given `Buffer` or `Uint8Array` instance from one character encoding to another. Returns a new `Buffer` instance.

Throws if the `fromEnc` or `toEnc` specify invalid character encodings or if conversion from `fromEnc` to `toEnc` is not permitted.

Encodings supported by `buffer.transcode()` are: `'ascii'`, `'utf8'`, `'utf16le'`, `'ucs2'`, `'latin1'`, and `'binary'`.

The transcoding process will use substitution characters if a given byte sequence cannot be adequately represented in the target encoding. For instance:

```
import { Buffer, transcode } from 'buffer';

const newBuf = transcode(Buffer.from('€'), 'utf8', 'ascii');
console.log(newBuf.toString('ascii'));
// Prints: '?'

const { Buffer, transcode } = require('buffer');

const newBuf = transcode(Buffer.from('€'), 'utf8', 'ascii');
console.log(newBuf.toString('ascii'));
// Prints: '?'
```

Because the Euro (€) sign is not representable in US-ASCII, it is replaced with `?` in the transcoded `Buffer`.

Class: SlowBuffer

Stability: 0 - Deprecated: Use `Buffer.allocUnsafeSlow()` instead.

See `Buffer.allocUnsafeSlow()`. This was never a class in the sense that the constructor always returned a `Buffer` instance, rather than a `SlowBuffer` instance.

new SlowBuffer(size)

Stability: 0 - Deprecated: Use `Buffer.allocUnsafeSlow()` instead.

- `size` `<integer>` The desired length of the new `SlowBuffer`.

See `Buffer.allocUnsafeSlow()`.

Buffer constants

`buffer.constants.MAX_LENGTH`

- `<integer>` The largest size allowed for a single `Buffer` instance.

On 32-bit architectures, this value currently is $2^{30} - 1$ (about 1 GB).

On 64-bit architectures, this value currently is 2^{32} (about 4 GB).

It reflects `v8::TypedArray::kMaxLength` under the hood.

This value is also available as `buffer.kMaxLength`.

`buffer.constants.MAX_STRING_LENGTH`

- `<integer>` The largest length allowed for a single `string` instance.

Represents the largest `length` that a `string` primitive can have, counted in UTF-16 code units.

This value may depend on the JS engine that is being used.

`Buffer.from()`, `Buffer.alloc()`, and `Buffer.allocUnsafe()`

In versions of Node.js prior to 6.0.0, `Buffer` instances were created using the `Buffer` constructor function, which allocates the returned `Buffer` differently based on what arguments are provided:

- Passing a number as the first argument to `Buffer()` (e.g. `new Buffer(10)`) allocates a new `Buffer` object of the specified size. Prior to Node.js 8.0.0, the memory allocated for such `Buffer` instances is *not* initialized and *can contain sensitive data*. Such `Buffer` instances *must* be subsequently initialized by using either `buf.fill(0)` or by writing to the entire `Buffer` before reading data from the `Buffer`. While this behavior is *intentional* to improve performance, development experience has demonstrated that a more explicit distinction is required between creating a fast-but-uninitialized `Buffer` versus creating a slower-but-safer `Buffer`. Since Node.js 8.0.0, `Buffer(num)` and `new Buffer(num)` return a `Buffer` with initialized memory.
- Passing a string, array, or `Buffer` as the first argument copies the passed object's data into the `Buffer`.
- Passing an `ArrayBuffer` or a `SharedArrayBuffer` returns a `Buffer` that shares allocated memory with the given array buffer.

Because the behavior of `new Buffer()` is different depending on the type of the first argument, security and reliability issues can be inadvertently introduced into applications when argument validation or `Buffer` initialization is not performed.

For example, if an attacker can cause an application to receive a number where a string is expected, the application may call `new Buffer(100)` instead of `new Buffer("100")`, leading it to allocate a 100 byte buffer instead of allocating a 3 byte buffer with content `"100"`. This is commonly possible using JSON API calls. Since JSON distinguishes between numeric and string types, it allows injection of numbers where a naively written application that does not validate its input sufficiently might expect to always receive a string. Before Node.js 8.0.0, the 100 byte buffer might contain arbitrary pre-existing in-memory data, so may be used to expose in-memory secrets to a remote attacker. Since Node.js 8.0.0, exposure of memory cannot occur because the data is zero-filled. However, other attacks are still possible, such as causing very large buffers to be allocated by the server, leading to performance degradation or crashing on memory exhaustion.

To make the creation of `Buffer` instances more reliable and less error-prone, the various forms of the `new Buffer()` constructor have been **deprecated** and replaced by separate `Buffer.from()`, `Buffer.alloc()`, and `Buffer.allocUnsafe()` methods.

Developers should migrate all existing uses of the `new Buffer()` constructors to one of these new APIs.

- `Buffer.from(array)` returns a new `Buffer` that *contains a copy* of the provided octets.
- `Buffer.from(arrayBuffer[, byteOffset[, length]])` returns a new `Buffer` that *shares the same allocated memory* as the given `ArrayBuffer`.
- `Buffer.from(buffer)` returns a new `Buffer` that *contains a copy* of the contents of the given `Buffer`.

- `Buffer.from(string[, encoding])` returns a new `Buffer` that contains a copy of the provided string.
- `Buffer.alloc(size[, fill[, encoding]])` returns a new initialized `Buffer` of the specified size. This method is slower than `Buffer.allocUnsafe(size)` but guarantees that newly created `Buffer` instances never contain old data that is potentially sensitive. A `TypeError` will be thrown if `size` is not a number.
- `Buffer.allocUnsafe(size)` and `Buffer.allocUnsafeSlow(size)` each return a new uninitialized `Buffer` of the specified `size`. Because the `Buffer` is uninitialized, the allocated segment of memory might contain old data that is potentially sensitive.

`Buffer` instances returned by `Buffer.allocUnsafe()` and `Buffer.from(array)` may be allocated off a shared internal memory pool if `size` is less than or equal to half `Buffer.poolSize`. Instances returned by `Buffer.allocUnsafeSlow()` never use the shared internal memory pool.

The `--zero-fill-buffers` command-line option

Node.js can be started using the `--zero-fill-buffers` command-line option to cause all newly-allocated `Buffer` instances to be zero-filled upon creation by default. Without the option, buffers created with `Buffer.allocUnsafe()`, `Buffer.allocUnsafeSlow()`, and `new SlowBuffer(size)` are not zero-filled. Use of this flag can have a measurable negative impact on performance. Use the `--zero-fill-buffers` option only when necessary to enforce that newly allocated `Buffer` instances cannot contain old data that is potentially sensitive.

```
$ node --zero-fill-buffers
> Buffer.allocUnsafe(5);
<Buffer 00 00 00 00 00>
```

What makes `Buffer.allocUnsafe()` and `Buffer.allocUnsafeSlow()` "unsafe"?

When calling `Buffer.allocUnsafe()` and `Buffer.allocUnsafeSlow()`, the segment of allocated memory is *uninitialized* (it is not zeroed-out). While this design makes the allocation of memory quite fast, the allocated segment of memory might contain old data that is potentially sensitive. Using a `Buffer` created by `Buffer.allocUnsafe()` without *completely* overwriting the memory can allow this old data to be leaked when the `Buffer` memory is read.

While there are clear performance advantages to using `Buffer.allocUnsafe()`, extra care *must* be taken in order to avoid introducing security vulnerabilities into an application.

C++ addons

Addons are dynamically-linked shared objects written in C++. The `require()` function can load addons as ordinary Node.js modules. Addons provide an interface between JavaScript and C/C++ libraries.

There are three options for implementing addons: Node-API, nan, or direct use of internal V8, libuv and Node.js libraries. Unless there is a need for direct access to functionality which is not exposed by Node-API, use Node-API. Refer to [C/C++ addons with Node-API](#) for more information on Node-API.

When not using Node-API, implementing addons is complicated, involving knowledge of several components and APIs:

- **V8** : the C++ library Node.js uses to provide the JavaScript implementation. V8 provides the mechanisms for creating objects, calling functions, etc. V8's API is documented mostly in the `v8.h` header file (`deps/v8/include/v8.h` in the Node.js source tree), which is also available [online](#).
- **libuv** : The C library that implements the Node.js event loop, its worker threads and all of the asynchronous behaviors of the platform. It also serves as a cross-platform abstraction library, giving easy, POSIX-like access across all major operating systems to many common system tasks, such as interacting with the filesystem, sockets, timers, and system events. libuv also provides a threading abstraction similar to POSIX threads for more sophisticated asynchronous addons that need to move beyond the standard event loop. Addon authors should avoid blocking the event loop with I/O or other time-intensive tasks by offloading work via libuv to non-blocking system operations, worker threads, or a custom use of libuv threads.

- Internal Node.js libraries. Node.js itself exports C++ APIs that addons can use, the most important of which is the `node::ObjectWrap` class.
- Node.js includes other statically linked libraries including OpenSSL. These other libraries are located in the `deps/` directory in the Node.js source tree. Only the libuv, OpenSSL, V8 and zlib symbols are purposefully re-exported by Node.js and may be used to various extents by addons. See [Linking to libraries included with Node.js](#) for additional information.

All of the following examples are available for [download](#) and may be used as the starting-point for an addon.

Hello world

This "Hello world" example is a simple addon, written in C++, that is the equivalent of the following JavaScript code:

```
module.exports.hello = () => 'world';
```

First, create the file `hello.cc` :

```
// hello.cc
#include <node.h>

namespace demo {

using v8::FunctionCallbackInfo;
using v8::Isolate;
using v8::Local;
using v8::Object;
using v8::String;
using v8::Value;

void Method(const FunctionCallbackInfo<Value>& args) {
    Isolate* isolate = args.GetIsolate();
    args.GetReturnValue().Set(String::NewFromUtf8(
        isolate, "world").ToLocalChecked());
}

void Initialize(Local<Object> exports) {
    NODE_SET_METHOD(exports, "hello", Method);
}

NODE_MODULE(NODE_GYP_MODULE_NAME, Initialize)

} // namespace demo
```

All Node.js addons must export an initialization function following the pattern:

```
void Initialize(Local<Object> exports);
NODE_MODULE(NODE_GYP_MODULE_NAME, Initialize)
```

There is no semi-colon after `NODE_MODULE` as it's not a function (see `node.h`).

The `module_name` must match the filename of the final binary (excluding the `.node` suffix).

In the `hello.cc` example, then, the initialization function is `Initialize` and the addon module name is `addon`.

When building addons with `node-gyp`, using the macro `NODE_GYP_MODULE_NAME` as the first parameter of `NODE_MODULE()` will ensure that the name of the final binary will be passed to `NODE_MODULE()`.

Context-aware addons

There are environments in which Node.js addons may need to be loaded multiple times in multiple contexts. For example, the [Electron](#) runtime runs multiple instances of Node.js in a single process. Each instance will have its own `require()` cache, and thus each instance will need a native addon to behave correctly when loaded via `require()`. This means that the addon must support multiple initializations.

A context-aware addon can be constructed by using the macro `NODE_MODULE_INITIALIZER`, which expands to the name of a function which Node.js will expect to find when it loads an addon. An addon can thus be initialized as in the following example:

```
using namespace v8;

extern "C" NODE_MODULE_EXPORT void
NODE_MODULE_INITIALIZER(Local<Object> exports,
                       Local<Value> module,
                       Local<Context> context) {
    /* Perform addon initialization steps here. */
}
```

Another option is to use the macro `NODE_MODULE_INIT()`, which will also construct a context-aware addon. Unlike `NODE_MODULE()`, which is used to construct an addon around a given addon initializer function, `NODE_MODULE_INIT()` serves as the declaration of such an initializer to be followed by a function body.

The following three variables may be used inside the function body following an invocation of `NODE_MODULE_INIT()`:

- `Local<Object> exports`,
- `Local<Value> module`, and
- `Local<Context> context`

The choice to build a context-aware addon carries with it the responsibility of carefully managing global static data. Since the addon may be loaded multiple times, potentially even from different threads, any global static data stored in the addon must be properly protected, and must not contain any persistent references to JavaScript objects. The reason for this is that JavaScript objects are only valid in one context, and will likely cause a crash when accessed from the wrong context or from a different thread than the one on which they were created.

The context-aware addon can be structured to avoid global static data by performing the following steps:

- Define a class which will hold per-addon-instance data and which has a static member of the form

```
static void DeleteInstance(void* data) {
    // Cast `data` to an instance of the class and delete it.
}
```

- Heap-allocate an instance of this class in the addon initializer. This can be accomplished using the `new` keyword.
- Call `node::AddEnvironmentCleanupHook()`, passing it the above-created instance and a pointer to `DeleteInstance()`. This will ensure the instance is deleted when the environment is torn down.
- Store the instance of the class in a `v8::External`, and

- Pass the `v8::External` to all methods exposed to JavaScript by passing it to `v8::FunctionTemplate::New()` or `v8::Function::New()` which creates the native-backed JavaScript functions. The third parameter of `v8::FunctionTemplate::New()` or `v8::Function::New()` accepts the `v8::External` and makes it available in the native callback using the `v8::FunctionCallbackInfo::Data()` method.

This will ensure that the per-addon-instance data reaches each binding that can be called from JavaScript. The per-addon-instance data must also be passed into any asynchronous callbacks the addon may create.

The following example illustrates the implementation of a context-aware addon:

```
#include <node.h>

using namespace v8;

class AddonData {
public:
    explicit AddonData(Isolate* isolate):
        call_count(0) {
        // Ensure this per-addon-instance data is deleted at environment cleanup.
        node::AddEnvironmentCleanupHook(isolate, DeleteInstance, this);
    }

    // Per-addon data.
    int call_count;

    static void DeleteInstance(void* data) {
        delete static_cast<AddonData*>(data);
    }
};

static void Method(const v8::FunctionCallbackInfo<v8::Value>& info) {
    // Retrieve the per-addon-instance data.
    AddonData* data =
        reinterpret_cast<AddonData*>(info.Data().As<External>()->Value());
    data->call_count++;
    info.GetReturnValue().Set((double)data->call_count);
}

// Initialize this addon to be context-aware.
NODE_MODULE_INIT(/* exports, module, context */) {
    Isolate* isolate = context->GetIsolate();

    // Create a new instance of `AddonData` for this instance of the addon and
    // tie its life cycle to that of the Node.js environment.
    AddonData* data = new AddonData(isolate);

    // Wrap the data in a `v8::External` so we can pass it to the method we
    // expose.
    Local<External> external = External::New(isolate, data);

    // Expose the method `Method` to JavaScript, and make sure it receives the
    // per-addon-instance data we created above by passing `external` as the
```

```
// third parameter to the `FunctionTemplate` constructor.
exports->Set(context,
    String::NewFromUtf8(isolate, "method").ToLocalChecked(),
    FunctionTemplate::New(isolate, Method, external)
    ->GetFunction(context).ToLocalChecked()).FromJust();
}
```

Worker support

In order to be loaded from multiple Node.js environments, such as a main thread and a Worker thread, an add-on needs to either:

- Be an Node-API addon, or
- Be declared as context-aware using `NODE_MODULE_INIT()` as described above

In order to support **Worker** threads, addons need to clean up any resources they may have allocated when such a thread exists. This can be achieved through the usage of the `AddEnvironmentCleanupHook()` function:

```
void AddEnvironmentCleanupHook(v8::Isolate* isolate,
    void (*fun)(void* arg),
    void* arg);
```

This function adds a hook that will run before a given Node.js instance shuts down. If necessary, such hooks can be removed before they are run using `RemoveEnvironmentCleanupHook()`, which has the same signature. Callbacks are run in last-in first-out order.

If necessary, there is an additional pair of `AddEnvironmentCleanupHook()` and `RemoveEnvironmentCleanupHook()` overloads, where the cleanup hook takes a callback function. This can be used for shutting down asynchronous resources, such as any libuv handles registered by the addon.

The following `addon.cc` uses `AddEnvironmentCleanupHook`:

```
// addon.cc
#include <node.h>
#include <assert.h>
#include <stdlib.h>

using node::AddEnvironmentCleanupHook;
using v8::HandleScope;
using v8::Isolate;
using v8::Local;
using v8::Object;

// Note: In a real-world application, do not rely on static/global data.
static char cookie[] = "yum yum";
static int cleanup_cb1_called = 0;
static int cleanup_cb2_called = 0;

static void cleanup_cb1(void* arg) {
    Isolate* isolate = static_cast<Isolate*>(arg);
    HandleScope scope(isolate);
    Local<Object> obj = Object::New(isolate);
    assert(!obj.IsEmpty()); // assert VM is still alive
    assert(obj->IsObject());
```

```

cleanup_cb1_called++;
}

static void cleanup_cb2(void* arg) {
    assert(arg == static_cast<void*>(cookie));
    cleanup_cb2_called++;
}

static void sanity_check(void*) {
    assert(cleanup_cb1_called == 1);
    assert(cleanup_cb2_called == 1);
}

// Initialize this addon to be context-aware.
NODE_MODULE_INIT(/* exports, module, context */) {
    Isolate* isolate = context->GetIsolate();

    AddEnvironmentCleanupHook(isolate, sanity_check, nullptr);
    AddEnvironmentCleanupHook(isolate, cleanup_cb2, cookie);
    AddEnvironmentCleanupHook(isolate, cleanup_cb1, isolate);
}

```

Test in JavaScript by running:

```

// test.js
require('./build/Release/addon');

```

Building

Once the source code has been written, it must be compiled into the binary `addon.node` file. To do so, create a file called `binding.gyp` in the top-level of the project describing the build configuration of the module using a JSON-like format. This file is used by `node-gyp`, a tool written specifically to compile Node.js addons.

```

{
  "targets": [
    {
      "target_name": "addon",
      "sources": [ "hello.cc" ]
    }
  ]
}

```

A version of the `node-gyp` utility is bundled and distributed with Node.js as part of `npm`. This version is not made directly available for developers to use and is intended only to support the ability to use the `npm install` command to compile and install addons. Developers who wish to use `node-gyp` directly can install it using the command `npm install -g node-gyp`. See the `node-gyp` [installation instructions](#) for more information, including platform-specific requirements.

Once the `binding.gyp` file has been created, use `node-gyp configure` to generate the appropriate project build files for the current platform. This will generate either a `Makefile` (on Unix platforms) or a `vcxproj` file (on Windows) in the `build/` directory.

Next, invoke the `node-gyp build` command to generate the compiled `addon.node` file. This will be put into the `build/Release/` directory.

When using `npm install` to install a Node.js addon, npm uses its own bundled version of `node-gyp` to perform this same set of actions, generating a compiled version of the addon for the user's platform on demand.

Once built, the binary addon can be used from within Node.js by pointing `require()` to the built `addon.node` module:

```
// hello.js
const addon = require('./build/Release/addon');

console.log(addon.hello());
// Prints: 'world'
```

Because the exact path to the compiled addon binary can vary depending on how it is compiled (i.e. sometimes it may be in `./build/Debug/`), addons can use the `bindings` package to load the compiled module.

While the `bindings` package implementation is more sophisticated in how it locates addon modules, it is essentially using a `try...catch` pattern similar to:

```
try {
  return require('./build/Release/addon.node');
} catch (err) {
  return require('./build/Debug/addon.node');
}
```

Linking to libraries included with Node.js

Node.js uses statically linked libraries such as V8, libuv and OpenSSL. All addons are required to link to V8 and may link to any of the other dependencies as well. Typically, this is as simple as including the appropriate `#include <...>` statements (e.g. `#include <v8.h>`) and `node-gyp` will locate the appropriate headers automatically. However, there are a few caveats to be aware of:

- When `node-gyp` runs, it will detect the specific release version of Node.js and download either the full source tarball or just the headers. If the full source is downloaded, addons will have complete access to the full set of Node.js dependencies. However, if only the Node.js headers are downloaded, then only the symbols exported by Node.js will be available.
- `node-gyp` can be run using the `--nodedir` flag pointing at a local Node.js source image. Using this option, the addon will have access to the full set of dependencies.

Loading addons using `require()`

The filename extension of the compiled addon binary is `.node` (as opposed to `.dll` or `.so`). The `require()` function is written to look for files with the `.node` file extension and initialize those as dynamically-linked libraries.

When calling `require()`, the `.node` extension can usually be omitted and Node.js will still find and initialize the addon. One caveat, however, is that Node.js will first attempt to locate and load modules or JavaScript files that happen to share the same base name. For instance, if there is a file `addon.js` in the same directory as the binary `addon.node`, then `require('addon')` will give precedence to the `addon.js` file and load it instead.

Native abstractions for Node.js

Each of the examples illustrated in this document directly use the Node.js and V8 APIs for implementing addons. The V8 API can, and has, changed dramatically from one V8 release to the next (and one major Node.js release to the next). With each change, addons may need to be updated and recompiled in order to continue functioning. The Node.js release schedule is designed to minimize the frequency and impact of such changes but there is little that Node.js can do to ensure stability of the V8 APIs.

The [Native Abstractions for Node.js](#) (or `nan`) provide a set of tools that add-on developers are recommended to use to keep compatibility between past and future releases of V8 and Node.js. See the [nan examples](#) for an illustration of how it can be used.

Node-API

Stability: 2 - Stable

Node-API is an API for building native add-ons. It is independent from the underlying JavaScript runtime (e.g. V8) and is maintained as part of Node.js itself. This API will be Application Binary Interface (ABI) stable across versions of Node.js. It is intended to insulate add-ons from changes in the underlying JavaScript engine and allow modules compiled for one version to run on later versions of Node.js without recompilation. Add-ons are built/packaged with the same approach/tools outlined in this document (node-gyp, etc.). The only difference is the set of APIs that are used by the native code. Instead of using the V8 or [Native Abstractions for Node.js](#) APIs, the functions available in the Node-API are used.

Creating and maintaining an add-on that benefits from the ABI stability provided by Node-API carries with it certain [implementation considerations](#).

To use Node-API in the above "Hello world" example, replace the content of `hello.cc` with the following. All other instructions remain the same.

```
// hello.cc using Node-API
#include <node_api.h>

namespace demo {

napi_value Method(napi_env env, napi_callback_info args) {
    napi_value greeting;
    napi_status status;

    status = napi_create_string_utf8(env, "world", NAPI_AUTO_LENGTH, &greeting);
    if (status != napi_ok) return nullptr;
    return greeting;
}

napi_value init(napi_env env, napi_value exports) {
    napi_status status;
    napi_value fn;

    status = napi_create_function(env, nullptr, 0, Method, nullptr, &fn);
    if (status != napi_ok) return nullptr;

    status = napi_set_named_property(env, exports, "hello", fn);
    if (status != napi_ok) return nullptr;
    return exports;
}

NAPI_MODULE(NODE_GYP_MODULE_NAME, init)

} // namespace demo
```


The functions available and how to use them are documented in [C/C++ addons with Node-API](#).

Addon examples

Following are some example addons intended to help developers get started. The examples use the V8 APIs. Refer to the online [V8 reference](#) for help with the various V8 calls, and V8's [Embedder's Guide](#) for an explanation of several concepts used such as handles, scopes, function templates, etc.

Each of these examples using the following `binding.gyp` file:

```
{
  "targets": [
    {
      "target_name": "addon",
      "sources": [ "addon.cc" ]
    }
  ]
}
```

In cases where there is more than one `.cc` file, simply add the additional filename to the `sources` array:

```
"sources": [ "addon.cc", "myexample.cc" ]
```

Once the `binding.gyp` file is ready, the example addons can be configured and built using `node-gyp`:

```
$ node-gyp configure build
```

Function arguments

Addons will typically expose objects and functions that can be accessed from JavaScript running within Node.js. When functions are invoked from JavaScript, the input arguments and return value must be mapped to and from the C/C++ code.

The following example illustrates how to read function arguments passed from JavaScript and how to return a result:

```
// addon.cc
#include <node.h>

namespace demo {

using v8::Exception;
using v8::FunctionCallbackInfo;
using v8::Isolate;
using v8::Local;
using v8::Number;
using v8::Object;
using v8::String;
using v8::Value;

// This is the implementation of the "add" method
// Input arguments are passed using the
```

```

// const FunctionCallbackInfo<Value>& args struct
void Add(const FunctionCallbackInfo<Value>& args) {
    Isolate* isolate = args.GetIsolate();

    // Check the number of arguments passed.
    if (args.Length() < 2) {
        // Throw an Error that is passed back to JavaScript
        isolate->ThrowException(Exception::TypeError(
            String::NewFromUtf8(isolate,
                                "Wrong number of arguments").ToLocalChecked()));

        return;
    }

    // Check the argument types
    if (!args[0]->IsNumber() || !args[1]->IsNumber()) {
        isolate->ThrowException(Exception::TypeError(
            String::NewFromUtf8(isolate,
                                "Wrong arguments").ToLocalChecked()));

        return;
    }

    // Perform the operation
    double value =
        args[0].As<Number>()->Value() + args[1].As<Number>()->Value();
    Local<Number> num = Number::New(isolate, value);

    // Set the return value (using the passed in
    // FunctionCallbackInfo<Value>&)
    args.GetReturnValue().Set(num);
}

void Init(Local<Object> exports) {
    NODE_SET_METHOD(exports, "add", Add);
}

NODE_MODULE(NODE_GYP_MODULE_NAME, Init)

} // namespace demo

```

Once compiled, the example addon can be required and used from within Node.js:

```

// test.js
const addon = require('./build/Release/addon');

console.log('This should be eight:', addon.add(3, 5));

```

Callbacks

It is common practice within addons to pass JavaScript functions to a C++ function and execute them from there. The following example illustrates how to invoke such callbacks:

```

// addon.cc
#include <node.h>

namespace demo {

using v8::Context;
using v8::Function;
using v8::FunctionCallbackInfo;
using v8::Isolate;
using v8::Local;
using v8::Null;
using v8::Object;
using v8::String;
using v8::Value;

void RunCallback(const FunctionCallbackInfo<Value>& args) {
    Isolate* isolate = args.GetIsolate();
    Local<Context> context = isolate->GetCurrentContext();
    Local<Function> cb = Local<Function>::Cast(args[0]);
    const unsigned argc = 1;
    Local<Value> argv[argc] = {
        String::NewFromUtf8(isolate,
                             "hello world").ToLocalChecked() };
    cb->Call(context, Null(isolate), argc, argv).ToLocalChecked();
}

void Init(Local<Object> exports, Local<Object> module) {
    NODE_SET_METHOD(module, "exports", RunCallback);
}

NODE_MODULE(NODE_GYP_MODULE_NAME, Init)

} // namespace demo

```

This example uses a two-argument form of `Init()` that receives the full `module` object as the second argument. This allows the addon to completely overwrite `exports` with a single function instead of adding the function as a property of `exports`.

To test it, run the following JavaScript:

```

// test.js
const addon = require('./build/Release/addon');

addon((msg) => {
    console.log(msg);
// Prints: 'hello world'
});

```

In this example, the callback function is invoked synchronously.

Object factory

Addons can create and return new objects from within a C++ function as illustrated in the following example. An object is created and returned with a property `msg` that echoes the string passed to `createObject()` :

```
// addon.cc
#include <node.h>

namespace demo {

using v8::Context;
using v8::FunctionCallbackInfo;
using v8::Isolate;
using v8::Local;
using v8::Object;
using v8::String;
using v8::Value;

void CreateObject(const FunctionCallbackInfo<Value>& args) {
    Isolate* isolate = args.GetIsolate();
    Local<Context> context = isolate->GetCurrentContext();

    Local<Object> obj = Object::New(isolate);
    obj->Set(context,
              String::NewFromUtf8(isolate,
                                   "msg").ToLocalChecked(),
              args[0]->ToString(context).ToLocalChecked())
        .FromJust();

    args.GetReturnValue().Set(obj);
}

void Init(Local<Object> exports, Local<Object> module) {
    NODE_SET_METHOD(module, "exports", CreateObject);
}

NODE_MODULE(NODE_GYP_MODULE_NAME, Init)

} // namespace demo
```

To test it in JavaScript:

```
// test.js
const addon = require('./build/Release/addon');

const obj1 = addon('hello');
const obj2 = addon('world');
console.log(obj1.msg, obj2.msg);
// Prints: 'hello world'
```

Function factory

Another common scenario is creating JavaScript functions that wrap C++ functions and returning those back to JavaScript:

```
// addon.cc
#include <node.h>

namespace demo {

using v8::Context;
using v8::Function;
using v8::FunctionCallbackInfo;
using v8::FunctionTemplate;
using v8::Isolate;
using v8::Local;
using v8::Object;
using v8::String;
using v8::Value;

void MyFunction(const FunctionCallbackInfo<Value>& args) {
    Isolate* isolate = args.GetIsolate();
    args.GetReturnValue().Set(String::NewFromUtf8(
        isolate, "hello world").ToLocalChecked());
}

void CreateFunction(const FunctionCallbackInfo<Value>& args) {
    Isolate* isolate = args.GetIsolate();

    Local<Context> context = isolate->GetCurrentContext();
    Local<FunctionTemplate> tpl = FunctionTemplate::New(isolate, MyFunction);
    Local<Function> fn = tpl->GetFunction(context).ToLocalChecked();

    // omit this to make it anonymous
    fn->SetName(String::NewFromUtf8(
        isolate, "theFunction").ToLocalChecked());

    args.GetReturnValue().Set(fn);
}

void Init(Local<Object> exports, Local<Object> module) {
    NODE_SET_METHOD(module, "exports", CreateFunction);
}

NODE_MODULE(NODE_GYP_MODULE_NAME, Init)

} // namespace demo
```

To test:

```
// test.js
const addon = require('./build/Release/addon');
```

```
const fn = addon();
console.log(fn());
// Prints: 'hello world'
```

Wrapping C++ objects

It is also possible to wrap C++ objects/classes in a way that allows new instances to be created using the JavaScript `new` operator:

```
// addon.cc
#include <node.h>
#include "myobject.h"

namespace demo {

using v8::Local;
using v8::Object;

void InitAll(Local<Object> exports) {
    MyObject::Init(exports);
}

NODE_MODULE(NODE_GYP_MODULE_NAME, InitAll)

} // namespace demo
```

Then, in `myobject.h`, the wrapper class inherits from `node::ObjectWrap`:

```
// myobject.h
#ifndef MYOBJECT_H
#define MYOBJECT_H

#include <node.h>
#include <node_object_wrap.h>

namespace demo {

class MyObject : public node::ObjectWrap {
public:
    static void Init(v8::Local<v8::Object> exports);

private:
    explicit MyObject(double value = 0);
    ~MyObject();

    static void New(const v8::FunctionCallbackInfo<v8::Value>& args);
    static void PlusOne(const v8::FunctionCallbackInfo<v8::Value>& args);

    double value_;
};
```

```
} // namespace demo
```

```
#endif
```

In `myobject.cc`, implement the various methods that are to be exposed. Below, the method `plusOne()` is exposed by adding it to the constructor's prototype:

```
// myobject.cc
#include "myobject.h"

namespace demo {

using v8::Context;
using v8::Function;
using v8::FunctionCallbackInfo;
using v8::FunctionTemplate;
using v8::Isolate;
using v8::Local;
using v8::Number;
using v8::Object;
using v8::ObjectTemplate;
using v8::String;
using v8::Value;

MyObject::MyObject(double value) : value_(value) {
}

MyObject::~MyObject() {
}

void MyObject::Init(Local<Object> exports) {
    Isolate* isolate = exports->GetIsolate();
    Local<Context> context = isolate->GetCurrentContext();

    Local<ObjectTemplate> addon_data_tpl = ObjectTemplate::New(isolate);
    addon_data_tpl->SetInternalFieldCount(1); // 1 field for the MyObject::New()
    Local<Object> addon_data =
        addon_data_tpl->NewInstance(context).ToLocalChecked();

    // Prepare constructor template
    Local<FunctionTemplate> tpl = FunctionTemplate::New(isolate, New, addon_data);
    tpl->SetClassName(String::NewFromUtf8(isolate, "MyObject").ToLocalChecked());
    tpl->InstanceTemplate()->SetInternalFieldCount(1);

    // Prototype
    NODE_SET_PROTOTYPE_METHOD(tpl, "plusOne", PlusOne);

    Local<Function> constructor = tpl->GetFunction(context).ToLocalChecked();
    addon_data->SetInternalField(0, constructor);
    exports->Set(context, String::NewFromUtf8(
        isolate, "MyObject").ToLocalChecked(),
```

```

        constructor).FromJust();
    }

void MyObject::New(const FunctionCallbackInfo<Value>& args) {
    Isolate* isolate = args.GetIsolate();
    Local<Context> context = isolate->GetCurrentContext();

    if (args.IsConstructCall()) {
        // Invoked as constructor: `new MyObject(...)`
        double value = args[0]->IsUndefined() ?
            0 : args[0]->NumberValue(context).FromMaybe(0);
        MyObject* obj = new MyObject(value);
        obj->Wrap(args.This());
        args.GetReturnValue().Set(args.This());
    } else {
        // Invoked as plain function `MyObject(...)` , turn into construct call.
        const int argc = 1;
        Local<Value> argv[argc] = { args[0] };
        Local<Function> cons =
            args.Data().As<Object>()->GetInternalField(0).As<Function>();
        Local<Object> result =
            cons->NewInstance(context, argc, argv).ToLocalChecked();
        args.GetReturnValue().Set(result);
    }
}

void MyObject::PlusOne(const FunctionCallbackInfo<Value>& args) {
    Isolate* isolate = args.GetIsolate();

    MyObject* obj = ObjectWrap::Unwrap<MyObject>(args.Holder());
    obj->value_ += 1;

    args.GetReturnValue().Set(Number::New(isolate, obj->value_));
}

} // namespace demo

```

To build this example, the `myobject.cc` file must be added to the `binding.gyp`:

```

{
  "targets": [
    {
      "target_name": "addon",
      "sources": [
        "addon.cc",
        "myobject.cc"
      ]
    }
  ]
}

```


Test it with:

```
// test.js
const addon = require('./build/Release/addon');

const obj = new addon.MyObject(10);
console.log(obj.plusOne());
// Prints: 11
console.log(obj.plusOne());
// Prints: 12
console.log(obj.plusOne());
// Prints: 13
```

The destructor for a wrapper object will run when the object is garbage-collected. For destructor testing, there are command-line flags that can be used to make it possible to force garbage collection. These flags are provided by the underlying V8 JavaScript engine. They are subject to change or removal at any time. They are not documented by Node.js or V8, and they should never be used outside of testing.

Factory of wrapped objects

Alternatively, it is possible to use a factory pattern to avoid explicitly creating object instances using the JavaScript `new` operator:

```
const obj = addon.createObject();
// instead of:
// const obj = new addon.Object();
```

First, the `createObject()` method is implemented in `addon.cc`:

```
// addon.cc
#include <node.h>
#include "myobject.h"

namespace demo {

using v8::FunctionCallbackInfo;
using v8::Isolate;
using v8::Local;
using v8::Object;
using v8::String;
using v8::Value;

void CreateObject(const FunctionCallbackInfo<Value>& args) {
    MyObject::NewInstance(args);
}

void InitAll(Local<Object> exports, Local<Object> module) {
    MyObject::Init(exports->GetIsolate());

    NODE_SET_METHOD(module, "exports", CreateObject);
}
```

```

NODE_MODULE(NODE_GYP_MODULE_NAME, InitAll)

} // namespace demo

```

In `myobject.h`, the static method `NewInstance()` is added to handle instantiating the object. This method takes the place of using `new` in JavaScript:

```

// myobject.h
#ifndef MYOBJECT_H
#define MYOBJECT_H

#include <node.h>
#include <node_object_wrap.h>

namespace demo {

class MyObject : public node::ObjectWrap {
public:
    static void Init(v8::Isolate* isolate);
    static void NewInstance(const v8::FunctionCallbackInfo<v8::Value>& args);

private:
    explicit MyObject(double value = 0);
    ~MyObject();

    static void New(const v8::FunctionCallbackInfo<v8::Value>& args);
    static void PlusOne(const v8::FunctionCallbackInfo<v8::Value>& args);
    static v8::Global<v8::Function> constructor;
    double value_;
};

} // namespace demo

#endif

```

The implementation in `myobject.cc` is similar to the previous example:

```

// myobject.cc
#include <node.h>
#include "myobject.h"

namespace demo {

using node::AddEnvironmentCleanupHook;
using v8::Context;
using v8::Function;
using v8::FunctionCallbackInfo;
using v8::FunctionTemplate;
using v8::Global;
using v8::Isolate;

```

```

using v8::Local;
using v8::Number;
using v8::Object;
using v8::String;
using v8::Value;

// Warning! This is not thread-safe, this addon cannot be used for worker
// threads.
Global<Function> MyObject::constructor;

MyObject::MyObject(double value) : value_(value) {
}

MyObject::~MyObject() {
}

void MyObject::Init(Isolate* isolate) {
    // Prepare constructor template
    Local<FunctionTemplate> tpl = FunctionTemplate::New(isolate, New);
    tpl->SetClassName(String::NewFromUtf8(isolate, "MyObject").ToLocalChecked());
    tpl->InstanceTemplate()->SetInternalFieldCount(1);

    // Prototype
    NODE_SET_PROTOTYPE_METHOD(tpl, "plusOne", PlusOne);

    Local<Context> context = isolate->GetCurrentContext();
    constructor.Reset(isolate, tpl->GetFunction(context).ToLocalChecked());

    AddEnvironmentCleanupHook(isolate, [](void*) {
        constructor.Reset();
    }, nullptr);
}

void MyObject::New(const FunctionCallbackInfo<Value>& args) {
    Isolate* isolate = args.GetIsolate();
    Local<Context> context = isolate->GetCurrentContext();

    if (args.IsConstructCall()) {
        // Invoked as constructor: `new MyObject(...)`
        double value = args[0]->IsUndefined() ?
            0 : args[0]->NumberValue(context).FromMaybe(0);
        MyObject* obj = new MyObject(value);
        obj->Wrap(args.This());
        args.GetReturnValue().Set(args.This());
    } else {
        // Invoked as plain function `MyObject(...)` , turn into construct call.
        const int argc = 1;
        Local<Value> argv[argc] = { args[0] };
        Local<Function> cons = Local<Function>::New(isolate, constructor);
        Local<Object> instance =
            cons->NewInstance(context, argc, argv).ToLocalChecked();
    }
}

```

```

    args.GetReturnValue().Set(instance);
}
}

void MyObject::NewInstance(const FunctionCallbackInfo<Value>& args) {
    Isolate* isolate = args.GetIsolate();

    const unsigned argc = 1;
    Local<Value> argv[argc] = { args[0] };
    Local<Function> cons = Local<Function>::New(isolate, constructor);
    Local<Context> context = isolate->GetCurrentContext();
    Local<Object> instance =
        cons->NewInstance(context, argc, argv).ToLocalChecked();

    args.GetReturnValue().Set(instance);
}

void MyObject::PlusOne(const FunctionCallbackInfo<Value>& args) {
    Isolate* isolate = args.GetIsolate();

    MyObject* obj = ObjectWrap::Unwrap<MyObject>(args.Holder());
    obj->value_ += 1;

    args.GetReturnValue().Set(Number::New(isolate, obj->value_));
}

} // namespace demo

```

Once again, to build this example, the `myobject.cc` file must be added to the `binding.gyp`:

```

{
  "targets": [
    {
      "target_name": "addon",
      "sources": [
        "addon.cc",
        "myobject.cc"
      ]
    }
  ]
}

```

Test it with:

```

// test.js
const createObject = require('./build/Release/addon');

const obj = createObject(10);
console.log(obj.plusOne());
// Prints: 11

```

```

console.log(obj.plusOne());
// Prints: 12
console.log(obj.plusOne());
// Prints: 13

const obj2 = createObject(20);
console.log(obj2.plusOne());
// Prints: 21
console.log(obj2.plusOne());
// Prints: 22
console.log(obj2.plusOne());
// Prints: 23

```

Passing wrapped objects around

In addition to wrapping and returning C++ objects, it is possible to pass wrapped objects around by unwrapping them with the Node.js helper function `node::ObjectWrap::Unwrap`. The following examples shows a function `add()` that can take two `MyObject` objects as input arguments:

```

// addon.cc
#include <node.h>
#include <node_object_wrap.h>
#include "myobject.h"

namespace demo {

using v8::Context;
using v8::FunctionCallbackInfo;
using v8::Isolate;
using v8::Local;
using v8::Number;
using v8::Object;
using v8::String;
using v8::Value;

void CreateObject(const FunctionCallbackInfo<Value>& args) {
    MyObject::NewInstance(args);
}

void Add(const FunctionCallbackInfo<Value>& args) {
    Isolate* isolate = args.GetIsolate();
    Local<Context> context = isolate->GetCurrentContext();

    MyObject* obj1 = node::ObjectWrap::Unwrap<MyObject>(
        args[0]->ToObject(context).ToLocalChecked());
    MyObject* obj2 = node::ObjectWrap::Unwrap<MyObject>(
        args[1]->ToObject(context).ToLocalChecked());

    double sum = obj1->value() + obj2->value();
    args.GetReturnValue().Set(Number::New(isolate, sum));
}

```

```

void InitAll(Local<Object> exports) {
    MyObject::Init(exports->GetIsolate());

    NODE_SET_METHOD(exports, "createObject", CreateObject);
    NODE_SET_METHOD(exports, "add", Add);
}

NODE_MODULE(NODE_GYP_MODULE_NAME, InitAll)

} // namespace demo

```

In `myobject.h`, a new public method is added to allow access to private values after unwrapping the object.

```

// myobject.h
#ifndef MYOBJECT_H
#define MYOBJECT_H

#include <node.h>
#include <node_object_wrap.h>

namespace demo {

class MyObject : public node::ObjectWrap {
public:
    static void Init(v8::Isolate* isolate);
    static void NewInstance(const v8::FunctionCallbackInfo<v8::Value>& args);
    inline double value() const { return value_; }

private:
    explicit MyObject(double value = 0);
    ~MyObject();

    static void New(const v8::FunctionCallbackInfo<v8::Value>& args);
    static v8::Global<v8::Function> constructor;
    double value_;
};

} // namespace demo

#endif

```

The implementation of `myobject.cc` is similar to before:

```

// myobject.cc
#include <node.h>
#include "myobject.h"

namespace demo {

```

```

using node::AddEnvironmentCleanupHook;
using v8::Context;
using v8::Function;
using v8::FunctionCallbackInfo;
using v8::FunctionTemplate;
using v8::Global;
using v8::Isolate;
using v8::Local;
using v8::Object;
using v8::String;
using v8::Value;

// Warning! This is not thread-safe, this addon cannot be used for worker
// threads.
Global<Function> MyObject::constructor;

MyObject::MyObject(double value) : value_(value) {
}

MyObject::~MyObject() {
}

void MyObject::Init(Isolate* isolate) {
    // Prepare constructor template
    Local<FunctionTemplate> tpl = FunctionTemplate::New(isolate, New);
    tpl->SetClassName(String::NewFromUtf8(isolate, "MyObject").ToLocalChecked());
    tpl->InstanceTemplate()->SetInternalFieldCount(1);

    Local<Context> context = isolate->GetCurrentContext();
    constructor.Reset(isolate, tpl->GetFunction(context).ToLocalChecked());

    AddEnvironmentCleanupHook(isolate, [](void*) {
        constructor.Reset();
    }, nullptr);
}

void MyObject::New(const FunctionCallbackInfo<Value>& args) {
    Isolate* isolate = args.GetIsolate();
    Local<Context> context = isolate->GetCurrentContext();

    if (args.IsConstructCall()) {
        // Invoked as constructor: `new MyObject(...)`
        double value = args[0]->IsUndefined() ?
            0 : args[0]->NumberValue(context).FromMaybe(0);
        MyObject* obj = new MyObject(value);
        obj->Wrap(args.This());
        args.GetReturnValue().Set(args.This());
    } else {
        // Invoked as plain function `MyObject(...)` , turn into construct call.
        const int argc = 1;
        Local<Value> argv[argc] = { args[0] };

```

```

    Local<Function> cons = Local<Function>::New(isolate, constructor);
    Local<Object> instance =
        cons->NewInstance(context, argc, argv).ToLocalChecked();
    args.GetReturnValue().Set(instance);
}
}

void MyObject::NewInstance(const FunctionCallbackInfo<Value>& args) {
    Isolate* isolate = args.GetIsolate();

    const unsigned argc = 1;
    Local<Value> argv[argc] = { args[0] };
    Local<Function> cons = Local<Function>::New(isolate, constructor);
    Local<Context> context = isolate->GetCurrentContext();
    Local<Object> instance =
        cons->NewInstance(context, argc, argv).ToLocalChecked();

    args.GetReturnValue().Set(instance);
}

} // namespace demo

```

Test it with:

```

// test.js
const addon = require('./build/Release/addon');

const obj1 = addon.createObject(10);
const obj2 = addon.createObject(20);
const result = addon.add(obj1, obj2);

console.log(result);
// Prints: 30

```

Node-API

Stability: 2 - Stable

Node-API (formerly N-API) is an API for building native Addons. It is independent from the underlying JavaScript runtime (for example, V8) and is maintained as part of Node.js itself. This API will be Application Binary Interface (ABI) stable across versions of Node.js. It is intended to insulate addons from changes in the underlying JavaScript engine and allow modules compiled for one major version to run on later major versions of Node.js without recompilation. The [ABI Stability](#) guide provides a more in-depth explanation.

Addons are built/packaged with the same approach/tools outlined in the section titled [C++ Addons](#). The only difference is the set of APIs that are used by the native code. Instead of using the V8 or [Native Abstractions for Node.js](#) APIs, the functions available in Node-API are used.

APIs exposed by Node-API are generally used to create and manipulate JavaScript values. Concepts and operations generally map to ideas specified in the ECMA-262 Language Specification. The APIs have the following properties:

- All Node-API calls return a status code of type `napi_status`. This status indicates whether the API call succeeded or failed.
- The API's return value is passed via an out parameter.
- All JavaScript values are abstracted behind an opaque type named `napi_value`.
- In case of an error status code, additional information can be obtained using `napi_get_last_error_info`. More information can be found in the error handling section [Error handling](#).

Node-API is a C API that ensures ABI stability across Node.js versions and different compiler levels. A C++ API can be easier to use. To support using C++, the project maintains a C++ wrapper module called `node-addon-api`. This wrapper provides an inlineable C++ API. Binaries built with `node-addon-api` will depend on the symbols for the Node-API C-based functions exported by Node.js. `node-addon-api` is a more efficient way to write code that calls Node-API. Take, for example, the following `node-addon-api` code. The first section shows the `node-addon-api` code and the second section shows what actually gets used in the addon.

```
Object obj = Object::New(env);
obj["foo"] = String::New(env, "bar");
```

```
napi_status status;
napi_value object, string;
status = napi_create_object(env, &object);
if (status != napi_ok) {
    napi_throw_error(env, ...);
    return;
}

status = napi_create_string_utf8(env, "bar", NAPI_AUTO_LENGTH, &string);
if (status != napi_ok) {
    napi_throw_error(env, ...);
    return;
}

status = napi_set_named_property(env, object, "foo", string);
if (status != napi_ok) {
    napi_throw_error(env, ...);
    return;
}
```

The end result is that the addon only uses the exported C APIs. As a result, it still gets the benefits of the ABI stability provided by the C API.

When using `node-addon-api` instead of the C APIs, start with the API [docs](#) for `node-addon-api`.

The [Node-API Resource](#) offers an excellent orientation and tips for developers just getting started with Node-API and `node-addon-api`.

Implications of ABI stability

Although Node-API provides an ABI stability guarantee, other parts of Node.js do not, and any external libraries used from the addon may not. In particular, none of the following APIs provide an ABI stability guarantee across major versions:

- the Node.js C++ APIs available via any of

```
#include <node.h>
#include <node_buffer.h>
```

```
#include <node_version.h>
#include <node_object_wrap.h>
```

- the libuv APIs which are also included with Node.js and available via

```
#include <uv.h>
```

- the V8 API available via

```
#include <v8.h>
```

Thus, for an addon to remain ABI-compatible across Node.js major versions, it must use Node-API exclusively by restricting itself to using

```
#include <node_api.h>
```

and by checking, for all external libraries that it uses, that the external library makes ABI stability guarantees similar to Node-API.

Building

Unlike modules written in JavaScript, developing and deploying Node.js native addons using Node-API requires an additional set of tools. Besides the basic tools required to develop for Node.js, the native addon developer requires a toolchain that can compile C and C++ code into a binary. In addition, depending upon how the native addon is deployed, the *user* of the native addon will also need to have a C/C++ toolchain installed.

For Linux developers, the necessary C/C++ toolchain packages are readily available. **GCC** is widely used in the Node.js community to build and test across a variety of platforms. For many developers, the **LLVM** compiler infrastructure is also a good choice.

For Mac developers, **Xcode** offers all the required compiler tools. However, it is not necessary to install the entire Xcode IDE. The following command installs the necessary toolchain:

```
xcode-select --install
```

For Windows developers, **Visual Studio** offers all the required compiler tools. However, it is not necessary to install the entire Visual Studio IDE. The following command installs the necessary toolchain:

```
npm install --global windows-build-tools
```

The sections below describe the additional tools available for developing and deploying Node.js native addons.

Build tools

Both the tools listed here require that *users* of the native addon have a C/C++ toolchain installed in order to successfully install the native addon.

node-gyp

node-gyp is a build system based on the **gyp-next** fork of Google's **GYP** tool and comes bundled with npm. GYP, and therefore node-gyp, requires that Python be installed.

Historically, node-gyp has been the tool of choice for building native addons. It has widespread adoption and documentation. However, some developers have run into limitations in node-gyp.

CMake.js

CMake.js is an alternative build system based on CMake .

CMake.js is a good choice for projects that already use CMake or for developers affected by limitations in node-gyp.

Uploading precompiled binaries

The three tools listed here permit native addon developers and maintainers to create and upload binaries to public or private servers. These tools are typically integrated with CI/CD build systems like Travis CI and AppVeyor to build and upload binaries for a variety of platforms and architectures. These binaries are then available for download by users who do not need to have a C/C++ toolchain installed.

node-pre-gyp

node-pre-gyp is a tool based on node-gyp that adds the ability to upload binaries to a server of the developer's choice. node-pre-gyp has particularly good support for uploading binaries to Amazon S3.

prebuild

prebuild is a tool that supports builds using either node-gyp or CMake.js. Unlike node-pre-gyp which supports a variety of servers, prebuild uploads binaries only to GitHub releases . prebuild is a good choice for GitHub projects using CMake.js.

prebuildify

prebuildify is a tool based on node-gyp. The advantage of prebuildify is that the built binaries are bundled with the native module when it's uploaded to npm. The binaries are downloaded from npm and are immediately available to the module user when the native module is installed.

Usage

In order to use the Node-API functions, include the file `node_api.h` which is located in the `src` directory in the node development tree:

```
#include <node_api.h>
```

This will opt into the default `NAPI_VERSION` for the given release of Node.js. In order to ensure compatibility with specific versions of Node-API, the version can be specified explicitly when including the header:

```
#define NAPI_VERSION 3
#include <node_api.h>
```

This restricts the Node-API surface to just the functionality that was available in the specified (and earlier) versions.

Some of the Node-API surface is experimental and requires explicit opt-in:

```
#define NAPI_EXPERIMENTAL
#include <node_api.h>
```

In this case the entire API surface, including any experimental APIs, will be available to the module code.

Node-API version matrix

Node-API versions are additive and versioned independently from Node.js. Version 4 is an extension to version 3 in that it has all of the APIs from version 3 with some additions. This means that it is not necessary to recompile for new versions of Node.js which are listed as supporting a later version.

	1	2	3
v6.x			v6.14.2*
v8.x	v8.6.0**	v8.10.0*	v8.11.2
v9.x	v9.0.0*	v9.3.0*	v9.11.0*
≥ v10.x	all releases	all releases	all releases

	4	5	6	7	8
v10.x	v10.16.0	v10.17.0	v10.20.0	v10.23.0	
v11.x	v11.8.0				
v12.x	v12.0.0	v12.11.0	v12.17.0	v12.19.0	v12.22.0
v13.x	v13.0.0	v13.0.0			
v14.x	v14.0.0	v14.0.0	v14.0.0	v14.12.0	v14.17.0
v15.x	v15.0.0	v15.0.0	v15.0.0	v15.0.0	v15.12.0
v16.x	v16.0.0	v16.0.0	v16.0.0	v16.0.0	v16.0.0

* Node-API was experimental.

** Node.js 8.0.0 included Node-API as experimental. It was released as Node-API version 1 but continued to evolve until Node.js 8.6.0. The API is different in versions prior to Node.js 8.6.0. We recommend Node-API version 3 or later.

Each API documented for Node-API will have a header named `added_in:`, and APIs which are stable will have the additional header `Node-API version:`. APIs are directly usable when using a Node.js version which supports the Node-API version shown in `Node-API version:` or higher. When using a Node.js version that does not support the `Node-API version:` listed or if there is no `Node-API version:` listed, then the API will only be available if `#define NAPI_EXPERIMENTAL` precedes the inclusion of `node_api.h` or `js_native_api.h`. If an API appears not to be available on a version of Node.js which is later than the one shown in `added_in:` then this is most likely the reason for the apparent absence.

The Node-APIs associated strictly with accessing ECMAScript features from native code can be found separately in `js_native_api.h` and `js_native_api_types.h`. The APIs defined in these headers are included in `node_api.h` and `node_api_types.h`. The headers are structured in this way in order to allow implementations of Node-API outside of Node.js. For those implementations the Node.js specific APIs may not be applicable.

The Node.js-specific parts of an addon can be separated from the code that exposes the actual functionality to the JavaScript environment so that the latter may be used with multiple implementations of Node-API. In the example below, `addon.c` and `addon.h` refer only to `js_native_api.h`. This ensures that `addon.c` can be reused to compile against either the Node.js implementation of Node-API or any implementation of Node-API outside of Node.js.

`addon_node.c` is a separate file that contains the Node.js specific entry point to the addon and which instantiates the addon by calling into `addon.c` when the addon is loaded into a Node.js environment.


```
    return result;
}
```

```
// addon_node.c
#include <node_api.h>
#include "addon.h"

NAPI_MODULE_INIT() {
    // This function body is expected to return a `napi_value`.
    // The variables `napi_env env` and `napi_value exports` may be used within
    // the body, as they are provided by the definition of `NAPI_MODULE_INIT()`.
    return create_addon(env);
}
```

Environment life cycle APIs

Section 8.7 of the [ECMAScript Language Specification](#) defines the concept of an "Agent" as a self-contained environment in which JavaScript code runs. Multiple such Agents may be started and terminated either concurrently or in sequence by the process.

A Node.js environment corresponds to an ECMAScript Agent. In the main process, an environment is created at startup, and additional environments can be created on separate threads to serve as [worker threads](#). When Node.js is embedded in another application, the main thread of the application may also construct and destroy a Node.js environment multiple times during the life cycle of the application process such that each Node.js environment created by the application may, in turn, during its life cycle create and destroy additional environments as worker threads.

From the perspective of a native addon this means that the bindings it provides may be called multiple times, from multiple contexts, and even concurrently from multiple threads.

Native addons may need to allocate global state which they use during their entire life cycle such that the state must be unique to each instance of the addon.

To this end, Node-API provides a way to allocate data such that its life cycle is tied to the life cycle of the Agent.

napi_set_instance_data

```
napi_status napi_set_instance_data(napi_env env,
                                   void* data,
                                   napi_finalize finalize_cb,
                                   void* finalize_hint);
```

- `[in] env`: The environment that the Node-API call is invoked under.
- `[in] data`: The data item to make available to bindings of this instance.
- `[in] finalize_cb`: The function to call when the environment is being torn down. The function receives `data` so that it might free it. [napi_finalize](#) provides more details.
- `[in] finalize_hint`: Optional hint to pass to the finalize callback during collection.

Returns `napi_ok` if the API succeeded.

This API associates `data` with the currently running Agent. `data` can later be retrieved using `napi_get_instance_data()`. Any existing data associated with the currently running Agent which was set by means of a previous call to `napi_set_instance_data()` will be

overwritten. If a `finalize_cb` was provided by the previous call, it will not be called.

`napi_get_instance_data`

```
napi_status napi_get_instance_data(napi_env env,
                                   void** data);
```

- `[in] env`: The environment that the Node-API call is invoked under.
- `[out] data`: The data item that was previously associated with the currently running Agent by a call to `napi_set_instance_data()`.

Returns `napi_ok` if the API succeeded.

This API retrieves data that was previously associated with the currently running Agent via `napi_set_instance_data()`. If no data is set, the call will succeed and `data` will be set to `NULL`.

Basic Node-API data types

Node-API exposes the following fundamental datatypes as abstractions that are consumed by the various APIs. These APIs should be treated as opaque, introspectable only with other Node-API calls.

`napi_status`

Integral status code indicating the success or failure of a Node-API call. Currently, the following status codes are supported.

```
typedef enum {
    napi_ok,
    napi_invalid_arg,
    napi_object_expected,
    napi_string_expected,
    napi_name_expected,
    napi_function_expected,
    napi_number_expected,
    napi_boolean_expected,
    napi_array_expected,
    napi_generic_failure,
    napi_pending_exception,
    napi_cancelled,
    napi_escape_called_twice,
    napi_handle_scope_mismatch,
    napi_callback_scope_mismatch,
    napi_queue_full,
    napi_closing,
    napi_bigint_expected,
    napi_date_expected,
    napi_arraybuffer_expected,
    napi_detachable_arraybuffer_expected,
    napi_would_deadlock, /* unused */
} napi_status;
```

If additional information is required upon an API returning a failed status, it can be obtained by calling `napi_get_last_error_info`.

napi_extended_error_info

```
typedef struct {
    const char* error_message;
    void* engine_reserved;
    uint32_t engine_error_code;
    napi_status error_code;
} napi_extended_error_info;
```

- `error_message` : UTF8-encoded string containing a VM-neutral description of the error.
- `engine_reserved` : Reserved for VM-specific error details. This is currently not implemented for any VM.
- `engine_error_code` : VM-specific error code. This is currently not implemented for any VM.
- `error_code` : The Node-API status code that originated with the last error.

See the [Error handling](#) section for additional information.

napi_env

`napi_env` is used to represent a context that the underlying Node-API implementation can use to persist VM-specific state. This structure is passed to native functions when they're invoked, and it must be passed back when making Node-API calls. Specifically, the same `napi_env` that was passed in when the initial native function was called must be passed to any subsequent nested Node-API calls. Caching the `napi_env` for the purpose of general reuse, and passing the `napi_env` between instances of the same addon running on different [Worker](#) threads is not allowed. The `napi_env` becomes invalid when an instance of a native addon is unloaded. Notification of this event is delivered through the callbacks given to `napi_add_env_cleanup_hook` and `napi_set_instance_data`.

napi_value

This is an opaque pointer that is used to represent a JavaScript value.

napi_threadsafe_function

This is an opaque pointer that represents a JavaScript function which can be called asynchronously from multiple threads via `napi_call_threadsafe_function()`.

napi_threadsafe_function_release_mode

A value to be given to `napi_release_threadsafe_function()` to indicate whether the thread-safe function is to be closed immediately (`napi_tsf_abort`) or merely released (`napi_tsf_release`) and thus available for subsequent use via `napi_acquire_threadsafe_function()` and `napi_call_threadsafe_function()`.

```
typedef enum {
    napi_tsf_release,
    napi_tsf_abort
} napi_threadsafe_function_release_mode;
```

napi_threadsafe_function_call_mode

A value to be given to `napi_call_threadsafe_function()` to indicate whether the call should block whenever the queue associated with the thread-safe function is full.

```
typedef enum {
    napi_tsf_nonblocking,
```



```
napi_tsfn_blocking
} napi_threadsafe_function_call_mode;
```

Node-API memory management types

napi_handle_scope

This is an abstraction used to control and modify the lifetime of objects created within a particular scope. In general, Node-API values are created within the context of a handle scope. When a native method is called from JavaScript, a default handle scope will exist. If the user does not explicitly create a new handle scope, Node-API values will be created in the default handle scope. For any invocations of code outside the execution of a native method (for instance, during a libuv callback invocation), the module is required to create a scope before invoking any functions that can result in the creation of JavaScript values.

Handle scopes are created using `napi_open_handle_scope` and are destroyed using `napi_close_handle_scope`. Closing the scope can indicate to the GC that all `napi_value`s created during the lifetime of the handle scope are no longer referenced from the current stack frame.

For more details, review the [Object lifetime management](#).

napi_escapable_handle_scope

Escapable handle scopes are a special type of handle scope to return values created within a particular handle scope to a parent scope.

napi_ref

This is the abstraction to use to reference a `napi_value`. This allows for users to manage the lifetimes of JavaScript values, including defining their minimum lifetimes explicitly.

For more details, review the [Object lifetime management](#).

napi_type_tag

A 128-bit value stored as two unsigned 64-bit integers. It serves as a UUID with which JavaScript objects can be "tagged" in order to ensure that they are of a certain type. This is a stronger check than `napi_instanceof`, because the latter can report a false positive if the object's prototype has been manipulated. Type-tagging is most useful in conjunction with `napi_wrap` because it ensures that the pointer retrieved from a wrapped object can be safely cast to the native type corresponding to the type tag that had been previously applied to the JavaScript object.

```
typedef struct {
    uint64_t lower;
    uint64_t upper;
} napi_type_tag;
```

napi_async_cleanup_hook_handle

An opaque value returned by `napi_add_async_cleanup_hook`. It must be passed to `napi_remove_async_cleanup_hook` when the chain of asynchronous cleanup events completes.

Node-API callback types

napi_callback_info

Opaque datatype that is passed to a callback function. It can be used for getting additional information about the context in which the callback was invoked.

napi_callback

Function pointer type for user-provided native functions which are to be exposed to JavaScript via Node-API. Callback functions should satisfy the following signature:

```
typedef napi_value (*napi_callback)(napi_env, napi_callback_info);
```

Unless for reasons discussed in [Object Lifetime Management](#), creating a handle and/or callback scope inside a `napi_callback` is not necessary.

napi_finalize

Function pointer type for add-on provided functions that allow the user to be notified when externally-owned data is ready to be cleaned up because the object with which it was associated with, has been garbage-collected. The user must provide a function satisfying the following signature which would get called upon the object's collection. Currently, `napi_finalize` can be used for finding out when objects that have external data are collected.

```
typedef void (*napi_finalize)(napi_env env,
                             void* finalize_data,
                             void* finalize_hint);
```

Unless for reasons discussed in [Object Lifetime Management](#), creating a handle and/or callback scope inside the function body is not necessary.

napi_async_execute_callback

Function pointer used with functions that support asynchronous operations. Callback functions must satisfy the following signature:

```
typedef void (*napi_async_execute_callback)(napi_env env, void* data);
```

Implementations of this function must avoid making Node-API calls that execute JavaScript or interact with JavaScript objects. Node-API calls should be in the `napi_async_complete_callback` instead. Do not use the `napi_env` parameter as it will likely result in execution of JavaScript.

napi_async_complete_callback

Function pointer used with functions that support asynchronous operations. Callback functions must satisfy the following signature:

```
typedef void (*napi_async_complete_callback)(napi_env env,
                                             napi_status status,
                                             void* data);
```

Unless for reasons discussed in [Object Lifetime Management](#), creating a handle and/or callback scope inside the function body is not necessary.

napi_threadsafe_function_call_js

Function pointer used with asynchronous thread-safe function calls. The callback will be called on the main thread. Its purpose is to use a data item arriving via the queue from one of the secondary threads to construct the parameters necessary for a call into JavaScript, usually via `napi_call_function`, and then make the call into JavaScript.

The data arriving from the secondary thread via the queue is given in the `data` parameter and the JavaScript function to call is given in the `js_callback` parameter.

Node-API sets up the environment prior to calling this callback, so it is sufficient to call the JavaScript function via `napi_call_function` rather than via `napi_make_callback`.

Callback functions must satisfy the following signature:

```
typedef void (*napi_threadsafe_function_call_js)(napi_env env,
                                                napi_value js_callback,
                                                void* context,
                                                void* data);
```

- `[in] env` : The environment to use for API calls, or `NULL` if the thread-safe function is being torn down and `data` may need to be freed.
- `[in] js_callback` : The JavaScript function to call, or `NULL` if the thread-safe function is being torn down and `data` may need to be freed. It may also be `NULL` if the thread-safe function was created without `js_callback`.
- `[in] context` : The optional data with which the thread-safe function was created.
- `[in] data` : Data created by the secondary thread. It is the responsibility of the callback to convert this native data to JavaScript values (with Node-API functions) that can be passed as parameters when `js_callback` is invoked. This pointer is managed entirely by the threads and this callback. Thus this callback should free the data.

Unless for reasons discussed in [Object Lifetime Management](#), creating a handle and/or callback scope inside the function body is not necessary.

`napi_async_cleanup_hook`

Function pointer used with `napi_add_async_cleanup_hook`. It will be called when the environment is being torn down.

Callback functions must satisfy the following signature:

```
typedef void (*napi_async_cleanup_hook)(napi_async_cleanup_hook_handle handle,
                                       void* data);
```

- `[in] handle` : The handle that must be passed to `napi_remove_async_cleanup_hook` after completion of the asynchronous cleanup.
- `[in] data` : The data that was passed to `napi_add_async_cleanup_hook`.

The body of the function should initiate the asynchronous cleanup actions at the end of which `handle` must be passed in a call to `napi_remove_async_cleanup_hook`.

Error handling

Node-API uses both return values and JavaScript exceptions for error handling. The following sections explain the approach for each case.

Return values

All of the Node-API functions share the same error handling pattern. The return type of all API functions is `napi_status`.

The return value will be `napi_ok` if the request was successful and no uncaught JavaScript exception was thrown. If an error occurred AND an exception was thrown, the `napi_status` value for the error will be returned. If an exception was thrown, and no error occurred, `napi_pending_exception` will be returned.

In cases where a return value other than `napi_ok` or `napi_pending_exception` is returned, `napi_is_exception_pending` must be called to check if an exception is pending. See the section on exceptions for more details.

The full set of possible `napi_status` values is defined in `napi_api_types.h`.

The `napi_status` return value provides a VM-independent representation of the error which occurred. In some cases it is useful to be able to get more detailed information, including a string representing the error as well as VM (engine)-specific information.

In order to retrieve this information `napi_get_last_error_info` is provided which returns a `napi_extended_error_info` structure. The format of the `napi_extended_error_info` structure is as follows:

```
typedef struct napi_extended_error_info {
    const char* error_message;
    void* engine_reserved;
    uint32_t engine_error_code;
    napi_status error_code;
};
```

- `error_message`: Textual representation of the error that occurred.
- `engine_reserved`: Opaque handle reserved for engine use only.
- `engine_error_code`: VM specific error code.
- `error_code`: Node-API status code for the last error.

`napi_get_last_error_info` returns the information for the last Node-API call that was made.

Do not rely on the content or format of any of the extended information as it is not subject to SemVer and may change at any time. It is intended only for logging purposes.

`napi_get_last_error_info`

```
napi_status
napi_get_last_error_info(napi_env env,
                        const napi_extended_error_info** result);
```

- `[in] env`: The environment that the API is invoked under.
- `[out] result`: The `napi_extended_error_info` structure with more information about the error.

Returns `napi_ok` if the API succeeded.

This API retrieves a `napi_extended_error_info` structure with information about the last error that occurred.

The content of the `napi_extended_error_info` returned is only valid up until a Node-API function is called on the same `env`.

Do not rely on the content or format of any of the extended information as it is not subject to SemVer and may change at any time. It is intended only for logging purposes.

This API can be called even if there is a pending JavaScript exception.

Exceptions

Any Node-API function call may result in a pending JavaScript exception. This is the case for any of the API functions, even those that may not cause the execution of JavaScript.

If the `napi_status` returned by a function is `napi_ok` then no exception is pending and no additional action is required. If the `napi_status` returned is anything other than `napi_ok` or `napi_pending_exception`, in order to try to recover and continue instead of simply returning immediately, `napi_is_exception_pending` must be called in order to determine if an exception is pending or not.

In many cases when a Node-API function is called and an exception is already pending, the function will return immediately with a `napi_status` of `napi_pending_exception`. However, this is not the case for all functions. Node-API allows a subset of the functions to be called to allow for some minimal cleanup before returning to JavaScript. In that case, `napi_status` will reflect the status for the function. It will not reflect previous pending exceptions. To avoid confusion, check the error status after every function call.

When an exception is pending one of two approaches can be employed.

The first approach is to do any appropriate cleanup and then return so that execution will return to JavaScript. As part of the transition back to JavaScript, the exception will be thrown at the point in the JavaScript code where the native method was invoked. The behavior of most Node-API calls is unspecified while an exception is pending, and many will simply return `napi_pending_exception`, so do as little as possible and then return to JavaScript where the exception can be handled.

The second approach is to try to handle the exception. There will be cases where the native code can catch the exception, take the appropriate action, and then continue. This is only recommended in specific cases where it is known that the exception can be safely handled. In these cases `napi_get_and_clear_last_exception` can be used to get and clear the exception. On success, result will contain the handle to the last JavaScript `Object` thrown. If it is determined, after retrieving the exception, the exception cannot be handled after all it can be re-thrown it with `napi_throw` where error is the JavaScript value to be thrown.

The following utility functions are also available in case native code needs to throw an exception or determine if a `napi_value` is an instance of a JavaScript `Error` object: `napi_throw_error`, `napi_throw_type_error`, `napi_throw_range_error` and `napi_is_error`.

The following utility functions are also available in case native code needs to create an `Error` object: `napi_create_error`, `napi_create_type_error`, and `napi_create_range_error`, where result is the `napi_value` that refers to the newly created JavaScript `Error` object.

The Node.js project is adding error codes to all of the errors generated internally. The goal is for applications to use these error codes for all error checking. The associated error messages will remain, but will only be meant to be used for logging and display with the expectation that the message can change without SemVer applying. In order to support this model with Node-API, both in internal functionality and for module specific functionality (as its good practice), the `throw_` and `create_` functions take an optional code parameter which is the string for the code to be added to the error object. If the optional parameter is `NULL` then no code will be associated with the error. If a code is provided, the name associated with the error is also updated to be:

```
originalName [code]
```

where `originalName` is the original name associated with the error and `code` is the code that was provided. For example, if the code is `'ERR_ERROR_1'` and a `TypeError` is being created the name will be:

```
TypeError [ERR_ERROR_1]
```

`napi_throw`

```
NAPI_EXTERN napi_status napi_throw(napi_env env, napi_value error);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] error` : The JavaScript value to be thrown.

Returns `napi_ok` if the API succeeded.

This API throws the JavaScript value provided.

`napi_throw_error`

```
NAPI_EXTERN napi_status napi_throw_error(napi_env env,
                                         const char* code,
                                         const char* msg);
```

- `[in] env`: The environment that the API is invoked under.
- `[in] code`: Optional error code to be set on the error.
- `[in] msg`: C string representing the text to be associated with the error.

Returns `napi_ok` if the API succeeded.

This API throws a JavaScript `Error` with the text provided.

`napi_throw_type_error`

```
NAPI_EXTERN napi_status napi_throw_type_error(napi_env env,
                                              const char* code,
                                              const char* msg);
```

- `[in] env`: The environment that the API is invoked under.
- `[in] code`: Optional error code to be set on the error.
- `[in] msg`: C string representing the text to be associated with the error.

Returns `napi_ok` if the API succeeded.

This API throws a JavaScript `TypeError` with the text provided.

`napi_throw_range_error`

```
NAPI_EXTERN napi_status napi_throw_range_error(napi_env env,
                                              const char* code,
                                              const char* msg);
```

- `[in] env`: The environment that the API is invoked under.
- `[in] code`: Optional error code to be set on the error.
- `[in] msg`: C string representing the text to be associated with the error.

Returns `napi_ok` if the API succeeded.

This API throws a JavaScript `RangeError` with the text provided.

`napi_is_error`

```
NAPI_EXTERN napi_status napi_is_error(napi_env env,
                                      napi_value value,
                                      bool* result);
```

- `[in] env`: The environment that the API is invoked under.
- `[in] value`: The `napi_value` to be checked.
- `[out] result`: Boolean value that is set to true if `napi_value` represents an error, false otherwise.

Returns `napi_ok` if the API succeeded.

This API queries a `napi_value` to check if it represents an error object.

`napi_create_error`

```
NAPI_EXTERN napi_status napi_create_error(napi_env env,
                                          napi_value code,
                                          napi_value msg,
                                          napi_value* result);
```

- `[in] env`: The environment that the API is invoked under.
- `[in] code`: Optional `napi_value` with the string for the error code to be associated with the error.
- `[in] msg`: `napi_value` that references a JavaScript `string` to be used as the message for the `Error`.
- `[out] result`: `napi_value` representing the error created.

Returns `napi_ok` if the API succeeded.

This API returns a JavaScript `Error` with the text provided.

`napi_create_type_error`

```
NAPI_EXTERN napi_status napi_create_type_error(napi_env env,
                                                napi_value code,
                                                napi_value msg,
                                                napi_value* result);
```

- `[in] env`: The environment that the API is invoked under.
- `[in] code`: Optional `napi_value` with the string for the error code to be associated with the error.
- `[in] msg`: `napi_value` that references a JavaScript `string` to be used as the message for the `Error`.
- `[out] result`: `napi_value` representing the error created.

Returns `napi_ok` if the API succeeded.

This API returns a JavaScript `TypeError` with the text provided.

`napi_create_range_error`

```
NAPI_EXTERN napi_status napi_create_range_error(napi_env env,
                                                 napi_value code,
                                                 napi_value msg,
                                                 napi_value* result);
```

- `[in] env`: The environment that the API is invoked under.
- `[in] code`: Optional `napi_value` with the string for the error code to be associated with the error.
- `[in] msg`: `napi_value` that references a JavaScript `string` to be used as the message for the `Error`.
- `[out] result`: `napi_value` representing the error created.

Returns `napi_ok` if the API succeeded.

This API returns a JavaScript `RangeError` with the text provided.

`napi_get_and_clear_last_exception`

```
napi_status napi_get_and_clear_last_exception(napi_env env,
                                              napi_value* result);
```

- `[in] env`: The environment that the API is invoked under.
- `[out] result`: The exception if one is pending, `NULL` otherwise.

Returns `napi_ok` if the API succeeded.

This API can be called even if there is a pending JavaScript exception.

`napi_is_exception_pending`

```
napi_status napi_is_exception_pending(napi_env env, bool* result);
```

- `[in] env`: The environment that the API is invoked under.
- `[out] result`: Boolean value that is set to true if an exception is pending.

Returns `napi_ok` if the API succeeded.

This API can be called even if there is a pending JavaScript exception.

`napi_fatal_exception`

```
napi_status napi_fatal_exception(napi_env env, napi_value err);
```

- `[in] env`: The environment that the API is invoked under.
- `[in] err`: The error that is passed to `'uncaughtException'`.

Trigger an `'uncaughtException'` in JavaScript. Useful if an async callback throws an exception with no way to recover.

Fatal errors

In the event of an unrecoverable error in a native module, a fatal error can be thrown to immediately terminate the process.

`napi_fatal_error`

```
NAPI_NO_RETURN void napi_fatal_error(const char* location,
                                     size_t location_len,
                                     const char* message,
                                     size_t message_len);
```

- `[in] location`: Optional location at which the error occurred.
- `[in] location_len`: The length of the location in bytes, or `NAPI_AUTO_LENGTH` if it is null-terminated.
- `[in] message`: The message associated with the error.
- `[in] message_len`: The length of the message in bytes, or `NAPI_AUTO_LENGTH` if it is null-terminated.

The function call does not return, the process will be terminated.

This API can be called even if there is a pending JavaScript exception.

Object lifetime management

As Node-API calls are made, handles to objects in the heap for the underlying VM may be returned as `napi_values`. These handles must hold the objects 'live' until they are no longer required by the native code, otherwise the objects could be collected before the native code was finished using them.

As object handles are returned they are associated with a 'scope'. The lifespan for the default scope is tied to the lifespan of the native method call. The result is that, by default, handles remain valid and the objects associated with these handles will be held live for the lifespan of the native method call.

In many cases, however, it is necessary that the handles remain valid for either a shorter or longer lifespan than that of the native method. The sections which follow describe the Node-API functions that can be used to change the handle lifespan from the default.

Making handle lifespan shorter than that of the native method

It is often necessary to make the lifespan of handles shorter than the lifespan of a native method. For example, consider a native method that has a loop which iterates through the elements in a large array:

```
for (int i = 0; i < 1000000; i++) {
    napi_value result;
    napi_status status = napi_get_element(env, object, i, &result);
    if (status != napi_ok) {
        break;
    }
    // do something with element
}
```

This would result in a large number of handles being created, consuming substantial resources. In addition, even though the native code could only use the most recent handle, all of the associated objects would also be kept alive since they all share the same scope.

To handle this case, Node-API provides the ability to establish a new 'scope' to which newly created handles will be associated. Once those handles are no longer required, the scope can be 'closed' and any handles associated with the scope are invalidated. The methods available to open/close scopes are `napi_open_handle_scope` and `napi_close_handle_scope`.

Node-API only supports a single nested hierarchy of scopes. There is only one active scope at any time, and all new handles will be associated with that scope while it is active. Scopes must be closed in the reverse order from which they are opened. In addition, all scopes created within a native method must be closed before returning from that method.

Taking the earlier example, adding calls to `napi_open_handle_scope` and `napi_close_handle_scope` would ensure that at most a single handle is valid throughout the execution of the loop:

```
for (int i = 0; i < 1000000; i++) {
    napi_handle_scope scope;
    napi_status status = napi_open_handle_scope(env, &scope);
    if (status != napi_ok) {
        break;
    }
    napi_value result;
    status = napi_get_element(env, object, i, &result);
    if (status != napi_ok) {
        break;
    }
    // do something with element
    status = napi_close_handle_scope(env, scope);
    if (status != napi_ok) {
```

```
    break;
  }
}
```

When nesting scopes, there are cases where a handle from an inner scope needs to live beyond the lifespan of that scope. Node-API supports an 'escapable scope' in order to support this case. An escapable scope allows one handle to be 'promoted' so that it 'escapes' the current scope and the lifespan of the handle changes from the current scope to that of the outer scope.

The methods available to open/close escapable scopes are `napi_open_escapable_handle_scope` and `napi_close_escapable_handle_scope`.

The request to promote a handle is made through `napi_escape_handle` which can only be called once.

`napi_open_handle_scope`

```
NAPI_EXTERN napi_status napi_open_handle_scope(napi_env env,
                                              napi_handle_scope* result);
```

- `[in] env`: The environment that the API is invoked under.
- `[out] result`: `napi_value` representing the new scope.

Returns `napi_ok` if the API succeeded.

This API opens a new scope.

`napi_close_handle_scope`

```
NAPI_EXTERN napi_status napi_close_handle_scope(napi_env env,
                                              napi_handle_scope scope);
```

- `[in] env`: The environment that the API is invoked under.
- `[in] scope`: `napi_value` representing the scope to be closed.

Returns `napi_ok` if the API succeeded.

This API closes the scope passed in. Scopes must be closed in the reverse order from which they were created.

This API can be called even if there is a pending JavaScript exception.

`napi_open_escapable_handle_scope`

```
NAPI_EXTERN napi_status
napi_open_escapable_handle_scope(napi_env env,
                                napi_handle_scope* result);
```

- `[in] env`: The environment that the API is invoked under.
- `[out] result`: `napi_value` representing the new scope.

Returns `napi_ok` if the API succeeded.

This API opens a new scope from which one object can be promoted to the outer scope.

`napi_close_escapable_handle_scope`

```

NAPI_EXTERN napi_status
    napi_close_escapable_handle_scope(napi_env env,
                                      napi_handle_scope scope);

```

- **[in] env**: The environment that the API is invoked under.
- **[in] scope**: **napi_value** representing the scope to be closed.

Returns **napi_ok** if the API succeeded.

This API closes the scope passed in. Scopes must be closed in the reverse order from which they were created.

This API can be called even if there is a pending JavaScript exception.

napi_escape_handle

```

napi_status napi_escape_handle(napi_env env,
                              napi_escapable_handle_scope scope,
                              napi_value escapee,
                              napi_value* result);

```

- **[in] env**: The environment that the API is invoked under.
- **[in] scope**: **napi_value** representing the current scope.
- **[in] escapee**: **napi_value** representing the JavaScript **Object** to be escaped.
- **[out] result**: **napi_value** representing the handle to the escaped **Object** in the outer scope.

Returns **napi_ok** if the API succeeded.

This API promotes the handle to the JavaScript object so that it is valid for the lifetime of the outer scope. It can only be called once per scope. If it is called more than once an error will be returned.

This API can be called even if there is a pending JavaScript exception.

References to objects with a lifespan longer than that of the native method

In some cases an addon will need to be able to create and reference objects with a lifespan longer than that of a single native method invocation. For example, to create a constructor and later use that constructor in a request to create instances, it must be possible to reference the constructor object across many different instance creation requests. This would not be possible with a normal handle returned as a **napi_value** as described in the earlier section. The lifespan of a normal handle is managed by scopes and all scopes must be closed before the end of a native method.

Node-API provides methods to create persistent references to an object. Each persistent reference has an associated count with a value of 0 or higher. The count determines if the reference will keep the corresponding object live. References with a count of 0 do not prevent the object from being collected and are often called 'weak' references. Any count greater than 0 will prevent the object from being collected.

References can be created with an initial reference count. The count can then be modified through **napi_reference_ref** and **napi_reference_unref**. If an object is collected while the count for a reference is 0, all subsequent calls to get the object associated with the reference **napi_get_reference_value** will return **NULL** for the returned **napi_value**. An attempt to call **napi_reference_ref** for a reference whose object has been collected results in an error.

References must be deleted once they are no longer required by the addon. When a reference is deleted, it will no longer prevent the corresponding object from being collected. Failure to delete a persistent reference results in a 'memory leak' with both the native memory for the persistent reference and the corresponding object on the heap being retained forever.

There can be multiple persistent references created which refer to the same object, each of which will either keep the object live or not based on its individual count.

napi_create_reference

[illegible]

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : `napi_value` representing the `Object` to which we want a reference.
- `[in] initial_refcount` : Initial reference count for the new reference.
- `[out] result` : `napi_ref` pointing to the new reference.

Returns `napi_ok` if the API succeeded.

This API create a new reference with the specified reference count to the `Object` passed in.

napi_delete_reference

```
NAPI_EXTERN napi_status napi_delete_reference(napi_env env, napi_ref ref);
```

- `[in] env`: The environment that the API is invoked under.
- `[in] ref`: `napi_ref` to be deleted.

Returns `napi_ok` if the API succeeded.

This API deletes the reference passed in.

This API can be called even if there is a pending JavaScript exception.

napi_reference_ref

[illegible]

- `[in] env` : The environment that the API is invoked under.
- `[in] ref` : `napi_ref` for which the reference count will be incremented.
- `[out] result` : The new reference count.

Returns `napi_ok` if the API succeeded.

This API increments the reference count for the reference passed in and returns the resulting reference count.

napi_reference_unref[illegible]

- `[in] env` : The environment that the API is invoked under.
- `[in] ref` : `napi_ref` for which the reference count will be decremented.
- `[out] result` : The new reference count.

Returns `napi_ok` if the API succeeded.

This API decrements the reference count for the reference passed in and returns the resulting reference count.

`napi_get_reference_value`

```

NAPI_EXTERN napi_status napi_get_reference_value(napi_env env,
                                                napi_ref ref,
                                                napi_value* result);

```

the `napi_value` passed in or out of these methods is a handle to the object to which the reference is related.

- `[in] env` : The environment that the API is invoked under.
- `[in] ref` : `napi_ref` for which we requesting the corresponding `Object` .
- `[out] result` : The `napi_value` for the `Object` referenced by the `napi_ref` .

Returns `napi_ok` if the API succeeded.

If still valid, this API returns the `napi_value` representing the JavaScript `Object` associated with the `napi_ref` . Otherwise, result will be `NULL` .

Cleanup on exit of the current Node.js instance

While a Node.js process typically releases all its resources when exiting, embedders of Node.js, or future Worker support, may require addons to register clean-up hooks that will be run once the current Node.js instance exits.

Node-API provides functions for registering and un-registering such callbacks. When those callbacks are run, all resources that are being held by the addon should be freed up.

`napi_add_env_cleanup_hook`

```

NODE_EXTERN napi_status napi_add_env_cleanup_hook(napi_env env,
                                                  void (*fun)(void* arg),
                                                  void* arg);

```

Registers `fun` as a function to be run with the `arg` parameter once the current Node.js environment exits.

A function can safely be specified multiple times with different `arg` values. In that case, it will be called multiple times as well. Providing the same `fun` and `arg` values multiple times is not allowed and will lead the process to abort.

The hooks will be called in reverse order, i.e. the most recently added one will be called first.

Removing this hook can be done by using `napi_remove_env_cleanup_hook` . Typically, that happens when the resource for which this hook was added is being torn down anyway.

For asynchronous cleanup, `napi_add_async_cleanup_hook` is available.

`napi_remove_env_cleanup_hook`

```

NAPI_EXTERN napi_status napi_remove_env_cleanup_hook(napi_env env,
                                                    void (*fun)(void* arg),
                                                    void* arg);

```

Unregisters `fun` as a function to be run with the `arg` parameter once the current Node.js environment exits. Both the argument and the function value need to be exact matches.

The function must have originally been registered with `napi_add_env_cleanup_hook`, otherwise the process will abort.

napi_add_async_cleanup_hook

```

NAPI_EXTERN napi_status napi_add_async_cleanup_hook(
    napi_env env,
    napi_async_cleanup_hook hook,
    void* arg,
    napi_async_cleanup_hook_handle* remove_handle);

```

- `[in] env`: The environment that the API is invoked under.
- `[in] hook`: The function pointer to call at environment teardown.
- `[in] arg`: The pointer to pass to `hook` when it gets called.
- `[out] remove_handle`: Optional handle that refers to the asynchronous cleanup hook.

Registers `hook`, which is a function of type `napi_async_cleanup_hook`, as a function to be run with the `remove_handle` and `arg` parameters once the current Node.js environment exits.

Unlike `napi_add_env_cleanup_hook`, the hook is allowed to be asynchronous.

Otherwise, behavior generally matches that of `napi_add_env_cleanup_hook`.

If `remove_handle` is not `NULL`, an opaque value will be stored in it that must later be passed to `napi_remove_async_cleanup_hook`, regardless of whether the hook has already been invoked. Typically, that happens when the resource for which this hook was added is being torn down anyway.

napi_remove_async_cleanup_hook

```

NAPI_EXTERN napi_status napi_remove_async_cleanup_hook(
    napi_async_cleanup_hook_handle remove_handle);

```

- `[in] remove_handle`: The handle to an asynchronous cleanup hook that was created with `napi_add_async_cleanup_hook`.

Unregisters the cleanup hook corresponding to `remove_handle`. This will prevent the hook from being executed, unless it has already started executing. This must be called on any `napi_async_cleanup_hook_handle` value obtained from `napi_add_async_cleanup_hook`.

Module registration

Node-API modules are registered in a manner similar to other modules except that instead of using the `NODE_MODULE` macro the following is used:

```

NAPI_MODULE(NODE_GYP_MODULE_NAME, Init)

```

The next difference is the signature for the `Init` method. For a Node-API module it is as follows:

```
napi_value Init(napi_env env, napi_value exports);
```

The return value from `Init` is treated as the `exports` object for the module. The `Init` method is passed an empty object via the `exports` parameter as a convenience. If `Init` returns `NULL`, the parameter passed as `exports` is exported by the module. Node-API modules cannot modify the `module` object but can specify anything as the `exports` property of the module.

To add the method `hello` as a function so that it can be called as a method provided by the addon:

```
napi_value Init(napi_env env, napi_value exports) {
    napi_status status;
    napi_property_descriptor desc = {
        "hello",
        NULL,
        Method,
        NULL,
        NULL,
        NULL,
        napi_writable | napi_enumerable | napi_configurable,
        NULL
    };
    status = napi_define_properties(env, exports, 1, &desc);
    if (status != napi_ok) return NULL;
    return exports;
}
```

To set a function to be returned by the `require()` for the addon:

```
napi_value Init(napi_env env, napi_value exports) {
    napi_value method;
    napi_status status;
    status = napi_create_function(env, "exports", NAPI_AUTO_LENGTH, Method, NULL, &method);
    if (status != napi_ok) return NULL;
    return method;
}
```

To define a class so that new instances can be created (often used with `Object wrap`):

```
// NOTE: partial example, not all referenced code is included
napi_value Init(napi_env env, napi_value exports) {
    napi_status status;
    napi_property_descriptor properties[] = {
        { "value", NULL, NULL, GetValue, SetValue, NULL, napi_writable | napi_configurable, NULL },
        DECLARE_NAPI_METHOD("plusOne", PlusOne),
        DECLARE_NAPI_METHOD("multiply", Multiply),
    };

    napi_value cons;
```

```

status =
    napi_define_class(env, "MyObject", New, NULL, 3, properties, &cons);
if (status != napi_ok) return NULL;

status = napi_create_reference(env, cons, 1, &constructor);
if (status != napi_ok) return NULL;

status = napi_set_named_property(env, exports, "MyObject", cons);
if (status != napi_ok) return NULL;

return exports;
}

```

You can also use the `NAPI_MODULE_INIT` macro, which acts as a shorthand for `NAPI_MODULE` and defining an `Init` function:

```

NAPI_MODULE_INIT() {
    napi_value answer;
    napi_status result;

    status = napi_create_int64(env, 42, &answer);
    if (status != napi_ok) return NULL;

    status = napi_set_named_property(env, exports, "answer", answer);
    if (status != napi_ok) return NULL;

    return exports;
}

```

All Node-API addons are context-aware, meaning they may be loaded multiple times. There are a few design considerations when declaring such a module. The documentation on [context-aware addons](#) provides more details.

The variables `env` and `exports` will be available inside the function body following the macro invocation.

For more details on setting properties on objects, see the section on [Working with JavaScript properties](#).

For more details on building addon modules in general, refer to the existing API.

Working with JavaScript values

Node-API exposes a set of APIs to create all types of JavaScript values. Some of these types are documented under [Section 6](#) of the [ECMAScript Language Specification](#).

Fundamentally, these APIs are used to do one of the following:

1. Create a new JavaScript object
2. Convert from a primitive C type to a Node-API value
3. Convert from Node-API value to a primitive C type
4. Get global instances including `undefined` and `null`

Node-API values are represented by the type `napi_value`. Any Node-API call that requires a JavaScript value takes in a `napi_value`. In some cases, the API does check the type of the `napi_value` up-front. However, for better performance, it's better for the caller to make sure that the `napi_value` in question is of the JavaScript type expected by the API.

Enum types

napi_key_collection_mode

```
typedef enum {  
    napi_key_include_prototypes,  
    napi_key_own_only  
} napi_key_collection_mode;
```

Describes the `Keys/Properties` filter enums:

`napi_key_collection_mode` limits the range of collected properties.

`napi_key_own_only` limits the collected properties to the given object only. `napi_key_include_prototypes` will include all keys of the objects's prototype chain as well.

napi_key_filter

```
typedef enum {  
    napi_key_all_properties = 0,  
    napi_key_writable = 1,  
    napi_key_enumerable = 1 << 1,  
    napi_key_configurable = 1 << 2,  
    napi_key_skip_strings = 1 << 3,  
    napi_key_skip_symbols = 1 << 4  
} napi_key_filter;
```

Property filter bits. They can be or'ed to build a composite filter.

napi_key_conversion

```
typedef enum {  
    napi_key_keep_numbers,  
    napi_key_numbers_to_strings  
} napi_key_conversion;
```

`napi_key_numbers_to_strings` will convert integer indices to strings. `napi_key_keep_numbers` will return numbers for integer indices.

napi_valuetype

```
typedef enum {  
    // ES6 types (corresponds to typeof)  
    napi_undefined,  
    napi_null,  
    napi_boolean,  
    napi_number,  
    napi_string,  
    napi_symbol,  
    napi_object,  
    napi_function,  
    napi_external,
```

```
napi_bigint,  
} napi_valuetype;
```

Describes the type of a `napi_value`. This generally corresponds to the types described in [Section 6.1](#) of the ECMAScript Language Specification. In addition to types in that section, `napi_valuetype` can also represent `Function`s and `Object`s with external data.

A JavaScript value of type `napi_external` appears in JavaScript as a plain object such that no properties can be set on it, and no prototype.

`napi_typedarray_type`

```
typedef enum {  
    napi_int8_array,  
    napi_uint8_array,  
    napi_uint8_clamped_array,  
    napi_int16_array,  
    napi_uint16_array,  
    napi_int32_array,  
    napi_uint32_array,  
    napi_float32_array,  
    napi_float64_array,  
    napi_bigint64_array,  
    napi_biguint64_array,  
} napi_typedarray_type;
```

This represents the underlying binary scalar datatype of the `TypedArray`. Elements of this enum correspond to [Section 22.2](#) of the ECMAScript Language Specification.

Object creation functions

`napi_create_array`

```
napi_status napi_create_array(napi_env env, napi_value* result)
```

- `[in] env`: The environment that the Node-API call is invoked under.
- `[out] result`: A `napi_value` representing a JavaScript `Array`.

Returns `napi_ok` if the API succeeded.

This API returns a Node-API value corresponding to a JavaScript `Array` type. JavaScript arrays are described in [Section 22.1](#) of the ECMAScript Language Specification.

`napi_create_array_with_length`

```
napi_status napi_create_array_with_length(napi_env env,  
                                          size_t length,  
                                          napi_value* result)
```

- `[in] env`: The environment that the API is invoked under.
- `[in] length`: The initial length of the `Array`.
- `[out] result`: A `napi_value` representing a JavaScript `Array`.

Returns `napi_ok` if the API succeeded.

This API returns a Node-API value corresponding to a JavaScript `Array` type. The `Array`'s `length` property is set to the passed-in `length` parameter. However, the underlying buffer is not guaranteed to be pre-allocated by the VM when the array is created. That behavior is left to the underlying VM implementation. If the buffer must be a contiguous block of memory that can be directly read and/or written via C, consider using `napi_create_external_arraybuffer`.

JavaScript arrays are described in [Section 22.1](#) of the ECMAScript Language Specification.

`napi_create_arraybuffer`

```
napi_status napi_create_arraybuffer(napi_env env,
                                   size_t byte_length,
                                   void** data,
                                   napi_value* result)
```

- `[in] env`: The environment that the API is invoked under.
- `[in] length`: The length in bytes of the array buffer to create.
- `[out] data`: Pointer to the underlying byte buffer of the `ArrayBuffer`.
- `[out] result`: A `napi_value` representing a JavaScript `ArrayBuffer`.

Returns `napi_ok` if the API succeeded.

This API returns a Node-API value corresponding to a JavaScript `ArrayBuffer`. `ArrayBuffer`s are used to represent fixed-length binary data buffers. They are normally used as a backing-buffer for `TypedArray` objects. The `ArrayBuffer` allocated will have an underlying byte buffer whose size is determined by the `length` parameter that's passed in. The underlying buffer is optionally returned back to the caller in case the caller wants to directly manipulate the buffer. This buffer can only be written to directly from native code. To write to this buffer from JavaScript, a typed array or `DataView` object would need to be created.

JavaScript `ArrayBuffer` objects are described in [Section 24.1](#) of the ECMAScript Language Specification.

`napi_create_buffer`

```
napi_status napi_create_buffer(napi_env env,
                              size_t size,
                              void** data,
                              napi_value* result)
```

- `[in] env`: The environment that the API is invoked under.
- `[in] size`: Size in bytes of the underlying buffer.
- `[out] data`: Raw pointer to the underlying buffer.
- `[out] result`: A `napi_value` representing a `node::Buffer`.

Returns `napi_ok` if the API succeeded.

This API allocates a `node::Buffer` object. While this is still a fully-supported data structure, in most cases using a `TypedArray` will suffice.

`napi_create_buffer_copy`

```
napi_status napi_create_buffer_copy(napi_env env,
                                   size_t length,
                                   const void* data,
```

```
void** result_data,  
napi_value* result)
```

- `[in] env`: The environment that the API is invoked under.
- `[in] size`: Size in bytes of the input buffer (should be the same as the size of the new buffer).
- `[in] data`: Raw pointer to the underlying buffer to copy from.
- `[out] result_data`: Pointer to the new `Buffer`'s underlying data buffer.
- `[out] result`: A `napi_value` representing a `node::Buffer`.

Returns `napi_ok` if the API succeeded.

This API allocates a `node::Buffer` object and initializes it with data copied from the passed-in buffer. While this is still a fully-supported data structure, in most cases using a `TypedArray` will suffice.

`napi_create_date`

```
napi_status napi_create_date(napi_env env,  
                             double time,  
                             napi_value* result);
```

- `[in] env`: The environment that the API is invoked under.
- `[in] time`: ECMAScript time value in milliseconds since 01 January, 1970 UTC.
- `[out] result`: A `napi_value` representing a JavaScript `Date`.

Returns `napi_ok` if the API succeeded.

This API does not observe leap seconds; they are ignored, as ECMAScript aligns with POSIX time specification.

This API allocates a JavaScript `Date` object.

JavaScript `Date` objects are described in [Section 20.3](#) of the ECMAScript Language Specification.

`napi_create_external`

```
napi_status napi_create_external(napi_env env,  
                                void* data,  
                                napi_finalize finalize_cb,  
                                void* finalize_hint,  
                                napi_value* result)
```

- `[in] env`: The environment that the API is invoked under.
- `[in] data`: Raw pointer to the external data.
- `[in] finalize_cb`: Optional callback to call when the external value is being collected. `napi_finalize` provides more details.
- `[in] finalize_hint`: Optional hint to pass to the finalize callback during collection.
- `[out] result`: A `napi_value` representing an external value.

Returns `napi_ok` if the API succeeded.

This API allocates a JavaScript value with external data attached to it. This is used to pass external data through JavaScript code, so it can be retrieved later by native code using `napi_get_value_external`.

The API adds a `napi_finalize` callback which will be called when the JavaScript object just created is ready for garbage collection. It is similar to `napi_wrap()` except that:

- the native data cannot be retrieved later using `napi_unwrap()`,
- nor can it be removed later using `napi_remove_wrap()`, and
- the object created by the API can be used with `napi_wrap()`.

The created value is not an object, and therefore does not support additional properties. It is considered a distinct value type: calling `napi_typeof()` with an external value yields `napi_external`.

`napi_create_external_arraybuffer`

```
napi_status  
napi_create_external_arraybuffer(napi_env env,  
                                void* external_data,  
                                size_t byte_length,  
                                napi_finalize finalize_cb,  
                                void* finalize_hint,  
                                napi_value* result)
```

- `[in] env`: The environment that the API is invoked under.
- `[in] external_data`: Pointer to the underlying byte buffer of the `ArrayBuffer`.
- `[in] byte_length`: The length in bytes of the underlying buffer.
- `[in] finalize_cb`: Optional callback to call when the `ArrayBuffer` is being collected. `napi_finalize` provides more details.
- `[in] finalize_hint`: Optional hint to pass to the finalize callback during collection.
- `[out] result`: A `napi_value` representing a JavaScript `ArrayBuffer`.

Returns `napi_ok` if the API succeeded.

This API returns a Node-API value corresponding to a JavaScript `ArrayBuffer`. The underlying byte buffer of the `ArrayBuffer` is externally allocated and managed. The caller must ensure that the byte buffer remains valid until the finalize callback is called.

The API adds a `napi_finalize` callback which will be called when the JavaScript object just created is ready for garbage collection. It is similar to `napi_wrap()` except that:

- the native data cannot be retrieved later using `napi_unwrap()`,
- nor can it be removed later using `napi_remove_wrap()`, and
- the object created by the API can be used with `napi_wrap()`.

JavaScript `ArrayBuffer`s are described in [Section 24.1](#) of the ECMAScript Language Specification.

`napi_create_external_buffer`

```
napi_status napi_create_external_buffer(napi_env env,  
                                       size_t length,  
                                       void* data,  
                                       napi_finalize finalize_cb,  
                                       void* finalize_hint,  
                                       napi_value* result)
```

- `[in] env`: The environment that the API is invoked under.

- Returns `napi_ok` if the API succeeded.

The API adds a `napi_finalize` callback which will be called when the JavaScript object just created is ready for garbage collection. It is similar to `napi_wrap()` except that:

- For Node.js >=4 Buffers are Uint8Array s.

```
napi_status napi_create_object(napi_env env, napi_value* result)
```

- Returns `napi_ok` if the API succeeded.

This API allocates a default JavaScript `Object`. It is the equivalent of doing `new Object()` in JavaScript.

The JavaScript `Object` type is described in [Section 6.1.7](#) of the ECMAScript Language Specification.

```
napi_status napi_create_symbol(napi_env env,
                               napi_value description,
                               napi_value* result)
```

- Returns `napi_ok` if the API succeeded.

This API creates a JavaScript `symbol` value from a UTF8-encoded C string.

The JavaScript `symbol` type is described in [Section 19.4](#) of the ECMAScript Language Specification.

[illegible]

```
size_t length,  
napi_value arraybuffer,  
size_t byte_offset,  
napi_value* result)
```

- [in] `env`: The environment that the API is invoked under.
- [in] `type`: Scalar datatype of the elements within the `TypedArray`.
- [in] `length`: Number of elements in the `TypedArray`.
- [in] `arraybuffer`: `ArrayBuffer` underlying the typed array.
- [in] `byte_offset`: The byte offset within the `ArrayBuffer` from which to start projecting the `TypedArray`.
- [out] `result`: A `napi_value` representing a JavaScript `TypedArray`.

Returns `napi_ok` if the API succeeded.

This API creates a JavaScript `TypedArray` object over an existing `ArrayBuffer`. `TypedArray` objects provide an array-like view over an underlying data buffer where each element has the same underlying binary scalar datatype.

It's required that $(\text{length} * \text{size_of_element}) + \text{byte_offset}$ should be \leq the size in bytes of the array passed in. If not, a `RangeError` exception is raised.

JavaScript `TypedArray` objects are described in [Section 22.2](#) of the ECMAScript Language Specification.

`napi_create_dataview`

```
napi_status napi_create_dataview(napi_env env,  
                                size_t byte_length,  
                                napi_value arraybuffer,  
                                size_t byte_offset,  
                                napi_value* result)
```

- [in] `env`: The environment that the API is invoked under.
- [in] `length`: Number of elements in the `DataView`.
- [in] `arraybuffer`: `ArrayBuffer` underlying the `DataView`.
- [in] `byte_offset`: The byte offset within the `ArrayBuffer` from which to start projecting the `DataView`.
- [out] `result`: A `napi_value` representing a JavaScript `DataView`.

Returns `napi_ok` if the API succeeded.

This API creates a JavaScript `DataView` object over an existing `ArrayBuffer`. `DataView` objects provide an array-like view over an underlying data buffer, but one which allows items of different size and type in the `ArrayBuffer`.

It is required that $\text{byte_length} + \text{byte_offset}$ is less than or equal to the size in bytes of the array passed in. If not, a `RangeError` exception is raised.

JavaScript `DataView` objects are described in [Section 24.3](#) of the ECMAScript Language Specification.

Functions to convert from C types to Node-API

`napi_create_int32`

```
napi_status napi_create_int32(napi_env env, int32_t value, napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : Integer value to be represented in JavaScript.
- `[out] result` : A `napi_value` representing a JavaScript `number`.

Returns `napi_ok` if the API succeeded.

This API is used to convert from the C `int32_t` type to the JavaScript `number` type.

The JavaScript `number` type is described in [Section 6.1.6](#) of the ECMAScript Language Specification.

`napi_create_uint32`

```
napi_status napi_create_uint32(napi_env env, uint32_t value, napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : Unsigned integer value to be represented in JavaScript.
- `[out] result` : A `napi_value` representing a JavaScript `number`.

Returns `napi_ok` if the API succeeded.

This API is used to convert from the C `uint32_t` type to the JavaScript `number` type.

The JavaScript `number` type is described in [Section 6.1.6](#) of the ECMAScript Language Specification.

`napi_create_int64`

```
napi_status napi_create_int64(napi_env env, int64_t value, napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : Integer value to be represented in JavaScript.
- `[out] result` : A `napi_value` representing a JavaScript `number`.

Returns `napi_ok` if the API succeeded.

This API is used to convert from the C `int64_t` type to the JavaScript `number` type.

The JavaScript `number` type is described in [Section 6.1.6](#) of the ECMAScript Language Specification. Note the complete range of `int64_t` cannot be represented with full precision in JavaScript. Integer values outside the range of `Number.MIN_SAFE_INTEGER` $-(2^{53} - 1)$ - `Number.MAX_SAFE_INTEGER` $(2^{53} - 1)$ will lose precision.

`napi_create_double`

```
napi_status napi_create_double(napi_env env, double value, napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : Double-precision value to be represented in JavaScript.
- `[out] result` : A `napi_value` representing a JavaScript `number`.

Returns `napi_ok` if the API succeeded.

This API is used to convert from the C `double` type to the JavaScript `number` type.

The JavaScript `number` type is described in [Section 6.1.6](#) of the ECMAScript Language Specification.

napi_create_bigint_int64

```
napi_status napi_create_bigint_int64(napi_env env,
                                     int64_t value,
                                     napi_value* result);
```

- [in] `env`: The environment that the API is invoked under.
- [in] `value`: Integer value to be represented in JavaScript.
- [out] `result`: A `napi_value` representing a JavaScript `BigInt`.

Returns `napi_ok` if the API succeeded.

This API converts the C `int64_t` type to the JavaScript `BigInt` type.

napi_create_bigint_uint64

```
napi_status napi_create_bigint_uint64(napi_env env,
                                       uint64_t value,
                                       napi_value* result);
```

- [in] `env`: The environment that the API is invoked under.
- [in] `value`: Unsigned integer value to be represented in JavaScript.
- [out] `result`: A `napi_value` representing a JavaScript `BigInt`.

Returns `napi_ok` if the API succeeded.

This API converts the C `uint64_t` type to the JavaScript `BigInt` type.

napi_create_bigint_words

```
napi_status napi_create_bigint_words(napi_env env,
                                     int sign_bit,
                                     size_t word_count,
                                     const uint64_t* words,
                                     napi_value* result);
```

- [in] `env`: The environment that the API is invoked under.
- [in] `sign_bit`: Determines if the resulting `BigInt` will be positive or negative.
- [in] `word_count`: The length of the `words` array.
- [in] `words`: An array of `uint64_t` little-endian 64-bit words.
- [out] `result`: A `napi_value` representing a JavaScript `BigInt`.

Returns `napi_ok` if the API succeeded.

This API converts an array of unsigned 64-bit words into a single `BigInt` value.

The resulting `BigInt` is calculated as: $(-1)^{\text{sign_bit}} (\text{words}[0] \times (2^{64})^0 + \text{words}[1] \times (2^{64})^1 + \dots)$

napi_create_string_latin1

```
napi_status napi_create_string_latin1(napi_env env,
                                      const char* str,
                                      size_t length,
                                      napi_value* result);
```

- `[in] env`: The environment that the API is invoked under.
- `[in] str`: Character buffer representing an ISO-8859-1-encoded string.
- `[in] length`: The length of the string in bytes, or `NAPI_AUTO_LENGTH` if it is null-terminated.
- `[out] result`: A `napi_value` representing a JavaScript `string`.

Returns `napi_ok` if the API succeeded.

This API creates a JavaScript `string` value from an ISO-8859-1-encoded C string. The native string is copied.

The JavaScript `string` type is described in [Section 6.1.4](#) of the ECMAScript Language Specification.

`napi_create_string_utf16`

```
napi_status napi_create_string_utf16(napi_env env,
                                     const char16_t* str,
                                     size_t length,
                                     napi_value* result)
```

- `[in] env`: The environment that the API is invoked under.
- `[in] str`: Character buffer representing a UTF16-LE-encoded string.
- `[in] length`: The length of the string in two-byte code units, or `NAPI_AUTO_LENGTH` if it is null-terminated.
- `[out] result`: A `napi_value` representing a JavaScript `string`.

Returns `napi_ok` if the API succeeded.

This API creates a JavaScript `string` value from a UTF16-LE-encoded C string. The native string is copied.

The JavaScript `string` type is described in [Section 6.1.4](#) of the ECMAScript Language Specification.

`napi_create_string_utf8`

```
napi_status napi_create_string_utf8(napi_env env,
                                    const char* str,
                                    size_t length,
                                    napi_value* result)
```

- `[in] env`: The environment that the API is invoked under.
- `[in] str`: Character buffer representing a UTF8-encoded string.
- `[in] length`: The length of the string in bytes, or `NAPI_AUTO_LENGTH` if it is null-terminated.
- `[out] result`: A `napi_value` representing a JavaScript `string`.

Returns `napi_ok` if the API succeeded.

This API creates a JavaScript `string` value from a UTF8-encoded C string. The native string is copied.

The JavaScript `string` type is described in [Section 6.1.4](#) of the ECMAScript Language Specification.

Functions to convert from Node-API to C types

napi_get_array_length

```
napi_status napi_get_array_length(napi_env env,
                                  napi_value value,
                                  uint32_t* result)
```

- [in] `env`: The environment that the API is invoked under.
- [in] `value`: `napi_value` representing the JavaScript `Array` whose length is being queried.
- [out] `result`: `uint32_t` representing length of the array.

Returns `napi_ok` if the API succeeded.

This API returns the length of an array.

`Array` length is described in [Section 22.1.4.1](#) of the ECMAScript Language Specification.

napi_get_arraybuffer_info

```
napi_status napi_get_arraybuffer_info(napi_env env,
                                       napi_value arraybuffer,
                                       void** data,
                                       size_t* byte_length)
```

- [in] `env`: The environment that the API is invoked under.
- [in] `arraybuffer`: `napi_value` representing the `ArrayBuffer` being queried.
- [out] `data`: The underlying data buffer of the `ArrayBuffer`. If `byte_length` is `0`, this may be `NULL` or any other pointer value.
- [out] `byte_length`: Length in bytes of the underlying data buffer.

Returns `napi_ok` if the API succeeded.

This API is used to retrieve the underlying data buffer of an `ArrayBuffer` and its length.

WARNING: Use caution while using this API. The lifetime of the underlying data buffer is managed by the `ArrayBuffer` even after it's returned. A possible safe way to use this API is in conjunction with `napi_create_reference`, which can be used to guarantee control over the lifetime of the `ArrayBuffer`. It's also safe to use the returned data buffer within the same callback as long as there are no calls to other APIs that might trigger a GC.

napi_get_buffer_info

```
napi_status napi_get_buffer_info(napi_env env,
                                  napi_value value,
                                  void** data,
                                  size_t* length)
```

- [in] `env`: The environment that the API is invoked under.
- [in] `value`: `napi_value` representing the `node::Buffer` being queried.
- [out] `data`: The underlying data buffer of the `node::Buffer`. If length is `0`, this may be `NULL` or any other pointer value.
- [out] `length`: Length in bytes of the underlying data buffer.


```
void** data,  
napi_value* arraybuffer,  
size_t* byte_offset)
```

- [in] `env` : The environment that the API is invoked under.
- [in] `dataview` : `napi_value` representing the `DataView` whose properties to query.
- [out] `byte_length` : Number of bytes in the `DataView`.
- [out] `data` : The data buffer underlying the `DataView`. If `byte_length` is 0, this may be `NULL` or any other pointer value.
- [out] `arraybuffer` : `ArrayBuffer` underlying the `DataView`.
- [out] `byte_offset` : The byte offset within the data buffer from which to start projecting the `DataView`.

Returns `napi_ok` if the API succeeded.

This API returns various properties of a `DataView`.

`napi_get_date_value`

```
napi_status napi_get_date_value(napi_env env,  
                                napi_value value,  
                                double* result)
```

- [in] `env` : The environment that the API is invoked under.
- [in] `value` : `napi_value` representing a JavaScript `Date`.
- [out] `result` : Time value as a `double` represented as milliseconds since midnight at the beginning of 01 January, 1970 UTC.

This API does not observe leap seconds; they are ignored, as ECMAScript aligns with POSIX time specification.

Returns `napi_ok` if the API succeeded. If a non-date `napi_value` is passed in it returns `napi_date_expected`.

This API returns the C double primitive of time value for the given JavaScript `Date`.

`napi_get_value_bool`

```
napi_status napi_get_value_bool(napi_env env, napi_value value, bool* result)
```

- [in] `env` : The environment that the API is invoked under.
- [in] `value` : `napi_value` representing JavaScript `Boolean`.
- [out] `result` : C boolean primitive equivalent of the given JavaScript `Boolean`.

Returns `napi_ok` if the API succeeded. If a non-boolean `napi_value` is passed in it returns `napi_boolean_expected`.

This API returns the C boolean primitive equivalent of the given JavaScript `Boolean`.

`napi_get_value_double`

```
napi_status napi_get_value_double(napi_env env,  
                                  napi_value value,  
                                  double* result)
```

- [in] `env` : The environment that the API is invoked under.

- [in] value: napi_value representing JavaScript number .
- [out] result : C double primitive equivalent of the given JavaScript number .

Returns napi_ok if the API succeeded. If a non-number napi_value is passed in it returns napi_number_expected .

This API returns the C double primitive equivalent of the given JavaScript number .

napi_get_value_bigint_int64

```
napi_status napi_get_value_bigint_int64(napi_env env,
                                         napi_value value,
                                         int64_t* result,
                                         bool* lossless);
```

- [in] env : The environment that the API is invoked under
- [in] value : napi_value representing JavaScript BigInt .
- [out] result : C int64_t primitive equivalent of the given JavaScript BigInt .
- [out] lossless : Indicates whether the BigInt value was converted losslessly.

Returns napi_ok if the API succeeded. If a non- BigInt is passed in it returns napi_bigint_expected .

This API returns the C int64_t primitive equivalent of the given JavaScript BigInt . If needed it will truncate the value, setting lossless to false .

napi_get_value_bigint_uint64

```
napi_status napi_get_value_bigint_uint64(napi_env env,
                                         napi_value value,
                                         uint64_t* result,
                                         bool* lossless);
```

- [in] env : The environment that the API is invoked under.
- [in] value : napi_value representing JavaScript BigInt .
- [out] result : C uint64_t primitive equivalent of the given JavaScript BigInt .
- [out] lossless : Indicates whether the BigInt value was converted losslessly.

Returns napi_ok if the API succeeded. If a non- BigInt is passed in it returns napi_bigint_expected .

This API returns the C uint64_t primitive equivalent of the given JavaScript BigInt . If needed it will truncate the value, setting lossless to false .

napi_get_value_bigint_words

```
napi_status napi_get_value_bigint_words(napi_env env,
                                         napi_value value,
                                         int* sign_bit,
                                         size_t* word_count,
                                         uint64_t* words);
```

- [in] env : The environment that the API is invoked under.
- [in] value : napi_value representing JavaScript BigInt .

- [out] `sign_bit` : Integer representing if the JavaScript `BigInt` is positive or negative.
- [in/out] `word_count` : Must be initialized to the length of the `words` array. Upon return, it will be set to the actual number of words that would be needed to store this `BigInt` .
- [out] `words` : Pointer to a pre-allocated 64-bit word array.

Returns `napi_ok` if the API succeeded.

This API converts a single `BigInt` value into a sign bit, 64-bit little-endian array, and the number of elements in the array. `sign_bit` and `words` may be both set to `NULL` , in order to get only `word_count` .

napi_get_value_external

```
napi_status napi_get_value_external(napi_env env,
                                   napi_value value,
                                   void** result)
```

- [in] `env` : The environment that the API is invoked under.
- [in] `value` : `napi_value` representing JavaScript external value.
- [out] `result` : Pointer to the data wrapped by the JavaScript external value.

Returns `napi_ok` if the API succeeded. If a non-external `napi_value` is passed in it returns `napi_invalid_arg` .

This API retrieves the external data pointer that was previously passed to `napi_create_external()` .

napi_get_value_int32

```
napi_status napi_get_value_int32(napi_env env,
                                 napi_value value,
                                 int32_t* result)
```

- [in] `env` : The environment that the API is invoked under.
- [in] `value` : `napi_value` representing JavaScript `number` .
- [out] `result` : C `int32` primitive equivalent of the given JavaScript `number` .

Returns `napi_ok` if the API succeeded. If a non-number `napi_value` is passed in `napi_number_expected` .

This API returns the C `int32` primitive equivalent of the given JavaScript `number` .

If the number exceeds the range of the 32 bit integer, then the result is truncated to the equivalent of the bottom 32 bits. This can result in a large positive number becoming a negative number if the value is $> 2^{31} - 1$.

Non-finite number values (`NaN` , `+Infinity` , or `-Infinity`) set the result to zero.

napi_get_value_int64

```
napi_status napi_get_value_int64(napi_env env,
                                 napi_value value,
                                 int64_t* result)
```

- [in] `env` : The environment that the API is invoked under.
- [in] `value` : `napi_value` representing JavaScript `number` .

- [out] result : C `int64` primitive equivalent of the given JavaScript `number` .

Returns `napi_ok` if the API succeeded. If a non-number `napi_value` is passed in it returns `napi_number_expected` .

This API returns the C `int64` primitive equivalent of the given JavaScript `number` .

`number` values outside the range of `Number.MIN_SAFE_INTEGER` $-(2^{53} - 1)$ - `Number.MAX_SAFE_INTEGER` $(2^{53} - 1)$ will lose precision.

Non-finite number values (`NaN` , `+Infinity` , or `-Infinity`) set the result to zero.

napi_get_value_string_latin1

```
napi_status napi_get_value_string_latin1(napi_env env,
                                         napi_value value,
                                         char* buf,
                                         size_t bufsize,
                                         size_t* result)
```

- [in] env : The environment that the API is invoked under.
- [in] value : `napi_value` representing JavaScript string.
- [in] buf : Buffer to write the ISO-8859-1-encoded string into. If `NULL` is passed in, the length of the string in bytes and excluding the null terminator is returned in `result` .
- [in] bufsize : Size of the destination buffer. When this value is insufficient, the returned string is truncated and null-terminated.
- [out] result : Number of bytes copied into the buffer, excluding the null terminator.

Returns `napi_ok` if the API succeeded. If a non- string `napi_value` is passed in it returns `napi_string_expected` .

This API returns the ISO-8859-1-encoded string corresponding the value passed in.

napi_get_value_string_utf8

```
napi_status napi_get_value_string_utf8(napi_env env,
                                       napi_value value,
                                       char* buf,
                                       size_t bufsize,
                                       size_t* result)
```

- [in] env : The environment that the API is invoked under.
- [in] value : `napi_value` representing JavaScript string.
- [in] buf : Buffer to write the UTF8-encoded string into. If `NULL` is passed in, the length of the string in bytes and excluding the null terminator is returned in `result` .
- [in] bufsize : Size of the destination buffer. When this value is insufficient, the returned string is truncated and null-terminated.
- [out] result : Number of bytes copied into the buffer, excluding the null terminator.

Returns `napi_ok` if the API succeeded. If a non- string `napi_value` is passed in it returns `napi_string_expected` .

This API returns the UTF8-encoded string corresponding the value passed in.

napi_get_value_string_utf16


```
napi_status napi_get_value_string_utf16(napi_env env,
                                       napi_value value,
                                       char16_t* buf,
                                       size_t bufsize,
                                       size_t* result)
```

- **[in] env** : The environment that the API is invoked under.
- **[in] value** : `napi_value` representing JavaScript string.
- **[in] buf** : Buffer to write the UTF16-LE-encoded string into. If `NULL` is passed in, the length of the string in 2-byte code units and excluding the null terminator is returned.
- **[in] bufsize** : Size of the destination buffer. When this value is insufficient, the returned string is truncated and null-terminated.
- **[out] result** : Number of 2-byte code units copied into the buffer, excluding the null terminator.

Returns `napi_ok` if the API succeeded. If a non-`string` `napi_value` is passed in it returns `napi_string_expected`.

This API returns the UTF16-encoded string corresponding the value passed in.

napi_get_value_uint32

```
napi_status napi_get_value_uint32(napi_env env,
                                  napi_value value,
                                  uint32_t* result)
```

- **[in] env** : The environment that the API is invoked under.
- **[in] value** : `napi_value` representing JavaScript `number`.
- **[out] result** : C primitive equivalent of the given `napi_value` as a `uint32_t`.

Returns `napi_ok` if the API succeeded. If a non-number `napi_value` is passed in it returns `napi_number_expected`.

This API returns the C primitive equivalent of the given `napi_value` as a `uint32_t`.

Functions to get global instances

napi_get_boolean

```
napi_status napi_get_boolean(napi_env env, bool value, napi_value* result)
```

- **[in] env** : The environment that the API is invoked under.
- **[in] value** : The value of the boolean to retrieve.
- **[out] result** : `napi_value` representing JavaScript `Boolean` singleton to retrieve.

Returns `napi_ok` if the API succeeded.

This API is used to return the JavaScript singleton object that is used to represent the given boolean value.

napi_get_global

```
napi_status napi_get_global(napi_env env, napi_value* result)
```

- **[in] env** : The environment that the API is invoked under.

- `[out] result`: `napi_value` representing JavaScript `global` object.

Returns `napi_ok` if the API succeeded.

This API returns the `global` object.

`napi_get_null`

```
napi_status napi_get_null(napi_env env, napi_value* result)
```

- `[in] env`: The environment that the API is invoked under.
- `[out] result`: `napi_value` representing JavaScript `null` object.

Returns `napi_ok` if the API succeeded.

This API returns the `null` object.

`napi_get_undefined`

```
napi_status napi_get_undefined(napi_env env, napi_value* result)
```

- `[in] env`: The environment that the API is invoked under.
- `[out] result`: `napi_value` representing JavaScript Undefined value.

Returns `napi_ok` if the API succeeded.

This API returns the Undefined object.

Working with JavaScript values and abstract operations

Node-API exposes a set of APIs to perform some abstract operations on JavaScript values. Some of these operations are documented under [Section 7](#) of the [ECMAScript Language Specification](#).

These APIs support doing one of the following:

1. Coerce JavaScript values to specific JavaScript types (such as `number` or `string`).
2. Check the type of a JavaScript value.
3. Check for equality between two JavaScript values.

`napi_coerce_to_bool`

```
napi_status napi_coerce_to_bool(napi_env env,
                                napi_value value,
                                napi_value* result)
```

- `[in] env`: The environment that the API is invoked under.
- `[in] value`: The JavaScript value to coerce.
- `[out] result`: `napi_value` representing the coerced JavaScript `Boolean`.

Returns `napi_ok` if the API succeeded.

This API implements the abstract operation `ToBoolean()` as defined in [Section 7.1.2](#) of the [ECMAScript Language Specification](#). This API can be re-entrant if getters are defined on the passed-in `Object`.

napi_coerce_to_number

```
napi_status napi_coerce_to_number(napi_env env,
                                napi_value value,
                                napi_value* result)
```

- `[in] env`: The environment that the API is invoked under.
- `[in] value`: The JavaScript value to coerce.
- `[out] result`: `napi_value` representing the coerced JavaScript `number`.

Returns `napi_ok` if the API succeeded.

This API implements the abstract operation `ToNumber()` as defined in [Section 7.1.3](#) of the ECMAScript Language Specification. This API can be re-entrant if getters are defined on the passed-in `Object`.

napi_coerce_to_object

```
napi_status napi_coerce_to_object(napi_env env,
                                 napi_value value,
                                 napi_value* result)
```

- `[in] env`: The environment that the API is invoked under.
- `[in] value`: The JavaScript value to coerce.
- `[out] result`: `napi_value` representing the coerced JavaScript `Object`.

Returns `napi_ok` if the API succeeded.

This API implements the abstract operation `ToObject()` as defined in [Section 7.1.13](#) of the ECMAScript Language Specification. This API can be re-entrant if getters are defined on the passed-in `Object`.

napi_coerce_to_string

```
napi_status napi_coerce_to_string(napi_env env,
                                  napi_value value,
                                  napi_value* result)
```

- `[in] env`: The environment that the API is invoked under.
- `[in] value`: The JavaScript value to coerce.
- `[out] result`: `napi_value` representing the coerced JavaScript `string`.

Returns `napi_ok` if the API succeeded.

This API implements the abstract operation `ToString()` as defined in [Section 7.1.13](#) of the ECMAScript Language Specification. This API can be re-entrant if getters are defined on the passed-in `Object`.

napi_typeof

```
napi_status napi_typeof(napi_env env, napi_value value, napi_valuetype* result)
```

- `[in] env`: The environment that the API is invoked under.

- `[in] value` : The JavaScript value whose type to query.
- `[out] result` : The type of the JavaScript value.

Returns `napi_ok` if the API succeeded.

- `napi_invalid_arg` if the type of `value` is not a known ECMAScript type and `value` is not an External value.

This API represents behavior similar to invoking the `typeof` Operator on the object as defined in [Section 12.5.5](#) of the ECMAScript Language Specification. However, there are some differences:

1. It has support for detecting an External value.
2. It detects `null` as a separate type, while ECMAScript `typeof` would detect `object`.

If `value` has a type that is invalid, an error is returned.

napi_instanceof

```
napi_status napi_instanceof(napi_env env,
                           napi_value object,
                           napi_value constructor,
                           bool* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] object` : The JavaScript value to check.
- `[in] constructor` : The JavaScript function object of the constructor function to check against.
- `[out] result` : Boolean that is set to true if `object instanceof constructor` is true.

Returns `napi_ok` if the API succeeded.

This API represents invoking the `instanceof` Operator on the object as defined in [Section 12.10.4](#) of the ECMAScript Language Specification.

napi_is_array

```
napi_status napi_is_array(napi_env env, napi_value value, bool* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : The JavaScript value to check.
- `[out] result` : Whether the given object is an array.

Returns `napi_ok` if the API succeeded.

This API represents invoking the `isArray` operation on the object as defined in [Section 7.2.2](#) of the ECMAScript Language Specification.

napi_is_arraybuffer

```
napi_status napi_is_arraybuffer(napi_env env, napi_value value, bool* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : The JavaScript value to check.
- `[out] result` : Whether the given object is an `ArrayBuffer`.

Returns `napi_ok` if the API succeeded.

This API checks if the `Object` passed in is an array buffer.

`napi_is_buffer`

```
napi_status napi_is_buffer(napi_env env, napi_value value, bool* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : The JavaScript value to check.
- `[out] result` : Whether the given `napi_value` represents a `node::Buffer` object.

Returns `napi_ok` if the API succeeded.

This API checks if the `Object` passed in is a buffer.

`napi_is_date`

```
napi_status napi_is_date(napi_env env, napi_value value, bool* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : The JavaScript value to check.
- `[out] result` : Whether the given `napi_value` represents a JavaScript `Date` object.

Returns `napi_ok` if the API succeeded.

This API checks if the `Object` passed in is a date.

`napi_is_error`

```
napi_status napi_is_error(napi_env env, napi_value value, bool* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : The JavaScript value to check.
- `[out] result` : Whether the given `napi_value` represents an `Error` object.

Returns `napi_ok` if the API succeeded.

This API checks if the `Object` passed in is an `Error`.

`napi_is_typedarray`

```
napi_status napi_is_typedarray(napi_env env, napi_value value, bool* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : The JavaScript value to check.
- `[out] result` : Whether the given `napi_value` represents a `TypedArray`.

Returns `napi_ok` if the API succeeded.

This API checks if the `Object` passed in is a typed array.

`napi_is_dataview`

```
napi_status napi_is_dataview(napi_env env, napi_value value, bool* result)
```

- [in] `env`: The environment that the API is invoked under.
- [in] `value`: The JavaScript value to check.
- [out] `result`: Whether the given `napi_value` represents a `DataView`.

Returns `napi_ok` if the API succeeded.

This API checks if the `Object` passed in is a `DataView`.

`napi_strict_equals`

```
napi_status napi_strict_equals(napi_env env,  
                              napi_value lhs,  
                              napi_value rhs,  
                              bool* result)
```

- [in] `env`: The environment that the API is invoked under.
- [in] `lhs`: The JavaScript value to check.
- [in] `rhs`: The JavaScript value to check against.
- [out] `result`: Whether the two `napi_value` objects are equal.

Returns `napi_ok` if the API succeeded.

This API represents the invocation of the Strict Equality algorithm as defined in [Section 7.2.14](#) of the ECMAScript Language Specification.

`napi_detach_arraybuffer`

```
napi_status napi_detach_arraybuffer(napi_env env,  
                                   napi_value arraybuffer)
```

- [in] `env`: The environment that the API is invoked under.
- [in] `arraybuffer`: The JavaScript `ArrayBuffer` to be detached.

Returns `napi_ok` if the API succeeded. If a non-detachable `ArrayBuffer` is passed in it returns `napi_detachable_arraybuffer_expected`.

Generally, an `ArrayBuffer` is non-detachable if it has been detached before. The engine may impose additional conditions on whether an `ArrayBuffer` is detachable. For example, V8 requires that the `ArrayBuffer` be external, that is, created with `napi_create_external_arraybuffer`.

This API represents the invocation of the `ArrayBuffer` detach operation as defined in [Section 24.1.1.3](#) of the ECMAScript Language Specification.

`napi_is_detached_arraybuffer`

```
napi_status napi_is_detached_arraybuffer(napi_env env,  
                                         napi_value arraybuffer,  
                                         bool* result)
```

- [in] `env`: The environment that the API is invoked under.

- `[in] arraybuffer` : The JavaScript `ArrayBuffer` to be checked.
- `[out] result` : Whether the `arraybuffer` is detached.

Returns `napi_ok` if the API succeeded.

The `ArrayBuffer` is considered detached if its internal data is `null`.

This API represents the invocation of the `ArrayBuffer IsDetachedBuffer` operation as defined in [Section 24.1.1.2](#) of the ECMAScript Language Specification.

Working with JavaScript properties

Node-API exposes a set of APIs to get and set properties on JavaScript objects. Some of these types are documented under [Section 7](#) of the [ECMAScript Language Specification](#).

Properties in JavaScript are represented as a tuple of a key and a value. Fundamentally, all property keys in Node-API can be represented in one of the following forms:

- Named: a simple UTF8-encoded string
- Integer-Indexed: an index value represented by `uint32_t`
- JavaScript value: these are represented in Node-API by `napi_value`. This can be a `napi_value` representing a `string`, `number`, or `symbol`.

Node-API values are represented by the type `napi_value`. Any Node-API call that requires a JavaScript value takes in a `napi_value`. However, it's the caller's responsibility to make sure that the `napi_value` in question is of the JavaScript type expected by the API.

The APIs documented in this section provide a simple interface to get and set properties on arbitrary JavaScript objects represented by `napi_value`.

For instance, consider the following JavaScript code snippet:

```
const obj = {};
obj.myProp = 123;
```

The equivalent can be done using Node-API values with the following snippet:

```
napi_status status = napi_generic_failure;

// const obj = {}
napi_value obj, value;
status = napi_create_object(env, &obj);
if (status != napi_ok) return status;

// Create a napi_value for 123
status = napi_create_int32(env, 123, &value);
if (status != napi_ok) return status;

// obj.myProp = 123
status = napi_set_named_property(env, obj, "myProp", value);
if (status != napi_ok) return status;
```

Indexed properties can be set in a similar manner. Consider the following JavaScript snippet:

```
const arr = [];  
arr[123] = 'hello';
```

The equivalent can be done using Node-API values with the following snippet:

```
napi_status status = napi_generic_failure;  
  
// const arr = [];  
napi_value arr, value;  
status = napi_create_array(env, &arr);  
if (status != napi_ok) return status;  
  
// Create a napi_value for 'hello'  
status = napi_create_string_utf8(env, "hello", NAPI_AUTO_LENGTH, &value);  
if (status != napi_ok) return status;  
  
// arr[123] = 'hello';  
status = napi_set_element(env, arr, 123, value);  
if (status != napi_ok) return status;
```

Properties can be retrieved using the APIs described in this section. Consider the following JavaScript snippet:

```
const arr = [];  
const value = arr[123];
```

The following is the approximate equivalent of the Node-API counterpart:

```
napi_status status = napi_generic_failure;  
  
// const arr = []  
napi_value arr, value;  
status = napi_create_array(env, &arr);  
if (status != napi_ok) return status;  
  
// const value = arr[123]  
status = napi_get_element(env, arr, 123, &value);  
if (status != napi_ok) return status;
```

Finally, multiple properties can also be defined on an object for performance reasons. Consider the following JavaScript:

```
const obj = {};  
Object.defineProperty(obj, {  
  'foo': { value: 123, writable: true, configurable: true, enumerable: true },  
  'bar': { value: 456, writable: true, configurable: true, enumerable: true }  
});
```

The following is the approximate equivalent of the Node-API counterpart:


```

napi_status status = napi_status_generic_failure;

// const obj = {};
napi_value obj;
status = napi_create_object(env, &obj);
if (status != napi_ok) return status;

// Create napi_values for 123 and 456
napi_value fooValue, barValue;
status = napi_create_int32(env, 123, &fooValue);
if (status != napi_ok) return status;
status = napi_create_int32(env, 456, &barValue);
if (status != napi_ok) return status;

// Set the properties
napi_property_descriptor descriptors[] = {
  { "foo", NULL, NULL, NULL, NULL, fooValue, napi_writable | napi_configurable, NULL },
  { "bar", NULL, NULL, NULL, NULL, barValue, napi_writable | napi_configurable, NULL }
}
status = napi_define_properties(env,
                                obj,
                                sizeof(descriptors) / sizeof(descriptors[0]),
                                descriptors);
if (status != napi_ok) return status;

```

Structures

napi_property_attributes

```

typedef enum {
  napi_default = 0,
  napi_writable = 1 << 0,
  napi_enumerable = 1 << 1,
  napi_configurable = 1 << 2,

  // Used with napi_define_class to distinguish static properties
  // from instance properties. Ignored by napi_define_properties.
  napi_static = 1 << 10,

  // Default for class methods.
  napi_default_method = napi_writable | napi_configurable,

  // Default for object properties, like in JS obj[prop].
  napi_default_jsproperty = napi_writable |
                             napi_enumerable |
                             napi_configurable,
} napi_property_attributes;

```

`napi_property_attributes` are flags used to control the behavior of properties set on a JavaScript object. Other than `napi_static` they correspond to the attributes listed in [Section 6.1.7.1](#) of the [ECMAScript Language Specification](#). They can be one or more of the following

bitflags:

- `napi_default` : No explicit attributes are set on the property. By default, a property is read only, not enumerable and not configurable.
- `napi_writable` : The property is writable.
- `napi_enumerable` : The property is enumerable.
- `napi_configurable` : The property is configurable as defined in [Section 6.1.7.1](#) of the [ECMAScript Language Specification](#) .
- `napi_static` : The property will be defined as a static property on a class as opposed to an instance property, which is the default. This is used only by `napi_define_class` . It is ignored by `napi_define_properties` .
- `napi_default_method` : Like a method in a JS class, the property is configurable and writeable, but not enumerable.
- `napi_default_jsproperty` : Like a property set via assignment in JavaScript, the property is writable, enumerable, and configurable.

napi_property_descriptor

```
typedef struct {  
    // One of utf8name or name should be NULL.  
    const char* utf8name;  
    napi_value name;  
  
    napi_callback method;  
    napi_callback getter;  
    napi_callback setter;  
    napi_value value;  
  
    napi_property_attributes attributes;  
    void* data;  
} napi_property_descriptor;
```

- `utf8name` : Optional string describing the key for the property, encoded as UTF8. One of `utf8name` or `name` must be provided for the property.
- `name` : Optional `napi_value` that points to a JavaScript string or symbol to be used as the key for the property. One of `utf8name` or `name` must be provided for the property.
- `value` : The value that's retrieved by a get access of the property if the property is a data property. If this is passed in, set `getter` , `setter` , `method` and `data` to `NULL` (since these members won't be used).
- `getter` : A function to call when a get access of the property is performed. If this is passed in, set `value` and `method` to `NULL` (since these members won't be used). The given function is called implicitly by the runtime when the property is accessed from JavaScript code (or if a get on the property is performed using a Node-API call). [napi_callback](#) provides more details.
- `setter` : A function to call when a set access of the property is performed. If this is passed in, set `value` and `method` to `NULL` (since these members won't be used). The given function is called implicitly by the runtime when the property is set from JavaScript code (or if a set on the property is performed using a Node-API call). [napi_callback](#) provides more details.
- `method` : Set this to make the property descriptor object's `value` property to be a JavaScript function represented by `method` . If this is passed in, set `value` , `getter` and `setter` to `NULL` (since these members won't be used). [napi_callback](#) provides more details.
- `attributes` : The attributes associated with the particular property. See [napi_property_attributes](#) .
- `data` : The callback data passed into `method` , `getter` and `setter` if this function is invoked.

Functions

napi_get_property_names

```
napi_status napi_get_property_names(napi_env env,
                                   napi_value object,
                                   napi_value* result);
```

- **[in] env** : The environment that the Node-API call is invoked under.
- **[in] object** : The object from which to retrieve the properties.
- **[out] result** : A **napi_value** representing an array of JavaScript values that represent the property names of the object. The API can be used to iterate over **result** using **napi_get_array_length** and **napi_get_element**.

Returns **napi_ok** if the API succeeded.

This API returns the names of the enumerable properties of **object** as an array of strings. The properties of **object** whose key is a symbol will not be included.

napi_get_all_property_names

```
napi_get_all_property_names(napi_env env,
                            napi_value object,
                            napi_key_collection_mode key_mode,
                            napi_key_filter key_filter,
                            napi_key_conversion key_conversion,
                            napi_value* result);
```

- **[in] env** : The environment that the Node-API call is invoked under.
- **[in] object** : The object from which to retrieve the properties.
- **[in] key_mode** : Whether to retrieve prototype properties as well.
- **[in] key_filter** : Which properties to retrieve (enumerable/readable/writable).
- **[in] key_conversion** : Whether to convert numbered property keys to strings.
- **[out] result** : A **napi_value** representing an array of JavaScript values that represent the property names of the object. **napi_get_array_length** and **napi_get_element** can be used to iterate over **result**.

Returns **napi_ok** if the API succeeded.

This API returns an array containing the names of the available properties of this object.

napi_set_property

```
napi_status napi_set_property(napi_env env,
                              napi_value object,
                              napi_value key,
                              napi_value value);
```

- **[in] env** : The environment that the Node-API call is invoked under.
- **[in] object** : The object on which to set the property.
- **[in] key** : The name of the property to set.
- **[in] value** : The property value.

Returns **napi_ok** if the API succeeded.

This API set a property on the **Object** passed in.

napi_get_property

```
napi_status napi_get_property(napi_env env,
                              napi_value object,
                              napi_value key,
                              napi_value* result);
```

- **[in] env** : The environment that the Node-API call is invoked under.
- **[in] object** : The object from which to retrieve the property.
- **[in] key** : The name of the property to retrieve.
- **[out] result** : The value of the property.

Returns **napi_ok** if the API succeeded.

This API gets the requested property from the **Object** passed in.

napi_has_property

```
napi_status napi_has_property(napi_env env,
                              napi_value object,
                              napi_value key,
                              bool* result);
```

- **[in] env** : The environment that the Node-API call is invoked under.
- **[in] object** : The object to query.
- **[in] key** : The name of the property whose existence to check.
- **[out] result** : Whether the property exists on the object or not.

Returns **napi_ok** if the API succeeded.

This API checks if the **Object** passed in has the named property.

napi_delete_property

```
napi_status napi_delete_property(napi_env env,
                                 napi_value object,
                                 napi_value key,
                                 bool* result);
```

- **[in] env** : The environment that the Node-API call is invoked under.
- **[in] object** : The object to query.
- **[in] key** : The name of the property to delete.
- **[out] result** : Whether the property deletion succeeded or not. **result** can optionally be ignored by passing **NULL** .

Returns **napi_ok** if the API succeeded.

This API attempts to delete the **key** own property from **object** .

napi_has_own_property

```
napi_status napi_has_own_property(napi_env env,
                                  napi_value object,
                                  napi_value key,
                                  bool* result);
```

- [in] `env` : The environment that the Node-API call is invoked under.
- [in] `object` : The object to query.
- [in] `key` : The name of the own property whose existence to check.
- [out] `result` : Whether the own property exists on the object or not.

Returns `napi_ok` if the API succeeded.

This API checks if the `object` passed in has the named own property. `key` must be a `string` or a `symbol`, or an error will be thrown. Node-API will not perform any conversion between data types.

`napi_set_named_property`

```
napi_status napi_set_named_property(napi_env env,
                                    napi_value object,
                                    const char* utf8Name,
                                    napi_value value);
```

- [in] `env` : The environment that the Node-API call is invoked under.
- [in] `object` : The object on which to set the property.
- [in] `utf8Name` : The name of the property to set.
- [in] `value` : The property value.

Returns `napi_ok` if the API succeeded.

This method is equivalent to calling `napi_set_property` with a `napi_value` created from the string passed in as `utf8Name`.

`napi_get_named_property`

```
napi_status napi_get_named_property(napi_env env,
                                    napi_value object,
                                    const char* utf8Name,
                                    napi_value* result);
```

- [in] `env` : The environment that the Node-API call is invoked under.
- [in] `object` : The object from which to retrieve the property.
- [in] `utf8Name` : The name of the property to get.
- [out] `result` : The value of the property.

Returns `napi_ok` if the API succeeded.

This method is equivalent to calling `napi_get_property` with a `napi_value` created from the string passed in as `utf8Name`.

`napi_has_named_property`

```
napi_status napi_has_named_property(napi_env env,
                                   napi_value object,
                                   const char* utf8Name,
                                   bool* result);
```

- `[in] env` : The environment that the Node-API call is invoked under.
- `[in] object` : The object to query.
- `[in] utf8Name` : The name of the property whose existence to check.
- `[out] result` : Whether the property exists on the object or not.

Returns `napi_ok` if the API succeeded.

This method is equivalent to calling `napi_has_property` with a `napi_value` created from the string passed in as `utf8Name`.

napi_set_element

```
napi_status napi_set_element(napi_env env,
                             napi_value object,
                             uint32_t index,
                             napi_value value);
```

- `[in] env` : The environment that the Node-API call is invoked under.
- `[in] object` : The object from which to set the properties.
- `[in] index` : The index of the property to set.
- `[in] value` : The property value.

Returns `napi_ok` if the API succeeded.

This API sets an element on the `Object` passed in.

napi_get_element

```
napi_status napi_get_element(napi_env env,
                             napi_value object,
                             uint32_t index,
                             napi_value* result);
```

- `[in] env` : The environment that the Node-API call is invoked under.
- `[in] object` : The object from which to retrieve the property.
- `[in] index` : The index of the property to get.
- `[out] result` : The value of the property.

Returns `napi_ok` if the API succeeded.

This API gets the element at the requested index.

napi_has_element

```
napi_status napi_has_element(napi_env env,
                             napi_value object,
```

```
uint32_t index,  
bool* result);
```

- `[in] env` : The environment that the Node-API call is invoked under.
- `[in] object` : The object to query.
- `[in] index` : The index of the property whose existence to check.
- `[out] result` : Whether the property exists on the object or not.

Returns `napi_ok` if the API succeeded.

This API returns if the `Object` passed in has an element at the requested index.

`napi_delete_element`

```
napi_status napi_delete_element(napi_env env,  
                               napi_value object,  
                               uint32_t index,  
                               bool* result);
```

- `[in] env` : The environment that the Node-API call is invoked under.
- `[in] object` : The object to query.
- `[in] index` : The index of the property to delete.
- `[out] result` : Whether the element deletion succeeded or not. `result` can optionally be ignored by passing `NULL`.

Returns `napi_ok` if the API succeeded.

This API attempts to delete the specified `index` from `object`.

`napi_define_properties`

```
napi_status napi_define_properties(napi_env env,  
                                  napi_value object,  
                                  size_t property_count,  
                                  const napi_property_descriptor* properties);
```

- `[in] env` : The environment that the Node-API call is invoked under.
- `[in] object` : The object from which to retrieve the properties.
- `[in] property_count` : The number of elements in the `properties` array.
- `[in] properties` : The array of property descriptors.

Returns `napi_ok` if the API succeeded.

This method allows the efficient definition of multiple properties on a given object. The properties are defined using property descriptors (see [napi_property_descriptor](#)). Given an array of such property descriptors, this API will set the properties on the object one at a time, as defined by `DefineOwnProperty()` (described in [Section 9.1.6](#) of the ECMA-262 specification).

`napi_object_freeze`

```
napi_status napi_object_freeze(napi_env env,  
                              napi_value object);
```

- `[in] env` : The environment that the Node-API call is invoked under.
- `[in] object` : The object to freeze.

Returns `napi_ok` if the API succeeded.

This method freezes a given object. This prevents new properties from being added to it, existing properties from being removed, prevents changing the enumerability, configurability, or writability of existing properties, and prevents the values of existing properties from being changed. It also prevents the object's prototype from being changed. This is described in [Section 19.1.2.6](#) of the ECMA-262 specification.

napi_object_seal

```
napi_status napi_object_seal(napi_env env,
                             napi_value object);
```

- `[in] env` : The environment that the Node-API call is invoked under.
- `[in] object` : The object to seal.

Returns `napi_ok` if the API succeeded.

This method seals a given object. This prevents new properties from being added to it, as well as marking all existing properties as non-configurable. This is described in [Section 19.1.2.20](#) of the ECMA-262 specification.

Working with JavaScript functions

Node-API provides a set of APIs that allow JavaScript code to call back into native code. Node-APIs that support calling back into native code take in a callback functions represented by the `napi_callback` type. When the JavaScript VM calls back to native code, the `napi_callback` function provided is invoked. The APIs documented in this section allow the callback function to do the following:

- Get information about the context in which the callback was invoked.
- Get the arguments passed into the callback.
- Return a `napi_value` back from the callback.

Additionally, Node-API provides a set of functions which allow calling JavaScript functions from native code. One can either call a function like a regular JavaScript function call, or as a constructor function.

Any non- `NULL` data which is passed to this API via the `data` field of the `napi_property_descriptor` items can be associated with `object` and freed whenever `object` is garbage-collected by passing both `object` and the data to `napi_add_finalizer`.

napi_call_function

```
NAPI_EXTERN napi_status napi_call_function(napi_env env,
                                           napi_value recv,
                                           napi_value func,
                                           size_t argc,
                                           const napi_value* argv,
                                           napi_value* result);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] recv` : The `this` value passed to the called function.
- `[in] func` : `napi_value` representing the JavaScript function to be invoked.
- `[in] argc` : The count of elements in the `argv` array.
- `[in] argv` : Array of `napi_values` representing JavaScript values passed in as arguments to the function.

- `[out] result`: `napi_value` representing the JavaScript object returned.

Returns `napi_ok` if the API succeeded.

This method allows a JavaScript function object to be called from a native add-on. This is the primary mechanism of calling back *from* the add-on's native code *into* JavaScript. For the special case of calling into JavaScript after an async operation, see `napi_make_callback`.

A sample use case might look as follows. Consider the following JavaScript snippet:

```
function AddTwo(num) {  
  return num + 2;  
}
```

Then, the above function can be invoked from a native add-on using the following code:

```
// Get the function named "AddTwo" on the global object  
napi_value global, add_two, arg;  
napi_status status = napi_get_global(env, &global);  
if (status != napi_ok) return;  
  
status = napi_get_named_property(env, global, "AddTwo", &add_two);  
if (status != napi_ok) return;  
  
// const arg = 1337  
status = napi_create_int32(env, 1337, &arg);  
if (status != napi_ok) return;  
  
napi_value* argv = &arg;  
size_t argc = 1;  
  
// AddTwo(arg);  
napi_value return_val;  
status = napi_call_function(env, global, add_two, argc, argv, &return_val);  
if (status != napi_ok) return;  
  
// Convert the result back to a native type  
int32_t result;  
status = napi_get_value_int32(env, return_val, &result);  
if (status != napi_ok) return;
```

`napi_create_function`

```
napi_status napi_create_function(napi_env env,  
                                const char* utf8name,  
                                size_t length,  
                                napi_callback cb,  
                                void* data,  
                                napi_value* result);
```

- `[in] env`: The environment that the API is invoked under.

- `[in] utf8Name` : The name of the function encoded as UTF8. This is visible within JavaScript as the new function object's `name` property.
- `[in] length` : The length of the `utf8name` in bytes, or `NAPI_AUTO_LENGTH` if it is null-terminated.
- `[in] cb` : The native function which should be called when this function object is invoked. `napi_callback` provides more details.
- `[in] data` : User-provided data context. This will be passed back into the function when invoked later.
- `[out] result` : `napi_value` representing the JavaScript function object for the newly created function.

Returns `napi_ok` if the API succeeded.

This API allows an add-on author to create a function object in native code. This is the primary mechanism to allow calling *into* the add-on's native code *from* JavaScript.

The newly created function is not automatically visible from script after this call. Instead, a property must be explicitly set on any object that is visible to JavaScript, in order for the function to be accessible from script.

In order to expose a function as part of the add-on's module exports, set the newly created function on the exports object. A sample module might look as follows:

```
napi_value SayHello(napi_env env, napi_callback_info info) {
    printf("Hello\n");
    return NULL;
}

napi_value Init(napi_env env, napi_value exports) {
    napi_status status;

    napi_value fn;
    status = napi_create_function(env, NULL, 0, SayHello, NULL, &fn);
    if (status != napi_ok) return NULL;

    status = napi_set_named_property(env, exports, "sayHello", fn);
    if (status != napi_ok) return NULL;

    return exports;
}

NAPI_MODULE(NODE_GYP_MODULE_NAME, Init)
```

Given the above code, the add-on can be used from JavaScript as follows:

```
const myaddon = require('./addon');
myaddon.sayHello();
```

The string passed to `require()` is the name of the target in `binding.gyp` responsible for creating the `.node` file.

Any non-`NULL` data which is passed to this API via the `data` parameter can be associated with the resulting JavaScript function (which is returned in the `result` parameter) and freed whenever the function is garbage-collected by passing both the JavaScript function and the data to `napi_add_finalizer`.

JavaScript `Function`s are described in [Section 19.2](#) of the ECMAScript Language Specification.

`napi_get_cb_info`

```
napi_status napi_get_cb_info(napi_env env,
                             napi_callback_info cbinfo,
                             size_t* argc,
                             napi_value* argv,
                             napi_value* thisArg,
                             void** data)
```

- **[in] env** : The environment that the API is invoked under.
- **[in] cbinfo** : The callback info passed into the callback function.
- **[in-out] argc** : Specifies the length of the provided **argv** array and receives the actual count of arguments.
- **[out] argv** : Buffer to which the **napi_value** representing the arguments are copied. If there are more arguments than the provided count, only the requested number of arguments are copied. If there are fewer arguments provided than claimed, the rest of **argv** is filled with **napi_value** values that represent **undefined**.
- **[out] this** : Receives the JavaScript **this** argument for the call.
- **[out] data** : Receives the data pointer for the callback.

Returns **napi_ok** if the API succeeded.

This method is used within a callback function to retrieve details about the call like the arguments and the **this** pointer from a given callback info.

napi_get_new_target

```
napi_status napi_get_new_target(napi_env env,
                                napi_callback_info cbinfo,
                                napi_value* result)
```

- **[in] env** : The environment that the API is invoked under.
- **[in] cbinfo** : The callback info passed into the callback function.
- **[out] result** : The **new.target** of the constructor call.

Returns **napi_ok** if the API succeeded.

This API returns the **new.target** of the constructor call. If the current callback is not a constructor call, the result is **NULL**.

napi_new_instance

```
napi_status napi_new_instance(napi_env env,
                              napi_value cons,
                              size_t argc,
                              napi_value* argv,
                              napi_value* result)
```

- **[in] env** : The environment that the API is invoked under.
- **[in] cons** : **napi_value** representing the JavaScript function to be invoked as a constructor.
- **[in] argc** : The count of elements in the **argv** array.
- **[in] argv** : Array of JavaScript values as **napi_value** representing the arguments to the constructor.
- **[out] result** : **napi_value** representing the JavaScript object returned, which in this case is the constructed object.

This method is used to instantiate a new JavaScript value using a given `napi_value` that represents the constructor for the object. For example, consider the following snippet:

```
function MyObject(param) {
  this.param = param;
}

const arg = 'hello';
const value = new MyObject(arg);
```

The following can be approximated in Node-API using the following snippet:

```
// Get the constructor function MyObject
napi_value global, constructor, arg, value;
napi_status status = napi_get_global(env, &global);
if (status != napi_ok) return;

status = napi_get_named_property(env, global, "MyObject", &constructor);
if (status != napi_ok) return;

// const arg = "hello"
status = napi_create_string_utf8(env, "hello", NAPI_AUTO_LENGTH, &arg);
if (status != napi_ok) return;

napi_value* argv = &arg;
size_t argc = 1;

// const value = new MyObject(arg)
status = napi_new_instance(env, constructor, argc, argv, &value);
```

Returns `napi_ok` if the API succeeded.

Object wrap

Node-API offers a way to "wrap" C++ classes and instances so that the class constructor and methods can be called from JavaScript.

1. The `napi_define_class` API defines a JavaScript class with constructor, static properties and methods, and instance properties and methods that correspond to the C++ class.
2. When JavaScript code invokes the constructor, the constructor callback uses `napi_wrap` to wrap a new C++ instance in a JavaScript object, then returns the wrapper object.
3. When JavaScript code invokes a method or property accessor on the class, the corresponding `napi_callback` C++ function is invoked. For an instance callback, `napi_unwrap` obtains the C++ instance that is the target of the call.

For wrapped objects it may be difficult to distinguish between a function called on a class prototype and a function called on an instance of a class. A common pattern used to address this problem is to save a persistent reference to the class constructor for later `instanceof` checks.

```
napi_value MyClass_constructor = NULL;
status = napi_get_reference_value(env, MyClass::es_constructor, &MyClass_constructor);
assert(napi_ok == status);
bool is_instance = false;
```

```

status = napi_instanceof(env, es_this, MyClass_constructor, &is_instance);
assert(napi_ok == status);
if (is_instance) {
    // napi_unwrap() ...
} else {
    // otherwise...
}

```

The reference must be freed once it is no longer needed.

There are occasions where `napi_instanceof()` is insufficient for ensuring that a JavaScript object is a wrapper for a certain native type. This is the case especially when wrapped JavaScript objects are passed back into the addon via static methods rather than as the `this` value of prototype methods. In such cases there is a chance that they may be unwrapped incorrectly.

```

const myAddon = require('./build/Release/my_addon.node');

// `openDatabase()` returns a JavaScript object that wraps a native database
// handle.
const dbHandle = myAddon.openDatabase();

// `query()` returns a JavaScript object that wraps a native query handle.
const queryHandle = myAddon.query(dbHandle, 'Gimme ALL the things!');

// There is an accidental error in the line below. The first parameter to
// `myAddon.queryHasRecords()` should be the database handle (`dbHandle`), not
// the query handle (`query`), so the correct condition for the while-loop
// should be
//
// myAddon.queryHasRecords(dbHandle, queryHandle)
//
while (myAddon.queryHasRecords(queryHandle, dbHandle)) {
    // retrieve records
}

```

In the above example `myAddon.queryHasRecords()` is a method that accepts two arguments. The first is a database handle and the second is a query handle. Internally, it unwraps the first argument and casts the resulting pointer to a native database handle. It then unwraps the second argument and casts the resulting pointer to a query handle. If the arguments are passed in the wrong order, the casts will work, however, there is a good chance that the underlying database operation will fail, or will even cause an invalid memory access.

To ensure that the pointer retrieved from the first argument is indeed a pointer to a database handle and, similarly, that the pointer retrieved from the second argument is indeed a pointer to a query handle, the implementation of `queryHasRecords()` has to perform a type validation. Retaining the JavaScript class constructor from which the database handle was instantiated and the constructor from which the query handle was instantiated in `napi_ref`s can help, because `napi_instanceof()` can then be used to ensure that the instances passed into `queryHashRecords()` are indeed of the correct type.

Unfortunately, `napi_instanceof()` does not protect against prototype manipulation. For example, the prototype of the database handle instance can be set to the prototype of the constructor for query handle instances. In this case, the database handle instance can appear as a query handle instance, and it will pass the `napi_instanceof()` test for a query handle instance, while still containing a pointer to a database handle.

To this end, Node-API provides type-tagging capabilities.

A type tag is a 128-bit integer unique to the addon. Node-API provides the `napi_type_tag` structure for storing a type tag. When such a value is passed along with a JavaScript object stored in a `napi_value` to `napi_type_tag_object()`, the JavaScript object will be "marked" with the type tag. The "mark" is invisible on the JavaScript side. When a JavaScript object arrives into a native binding, `napi_check_object_type_tag()` can be used along with the original type tag to determine whether the JavaScript object was previously "marked" with the type tag. This creates a type-checking capability of a higher fidelity than `napi_instanceof()` can provide, because such type-tagging survives prototype manipulation and addon unloading/reloading.

Continuing the above example, the following skeleton addon implementation illustrates the use of `napi_type_tag_object()` and `napi_check_object_type_tag()`.

```
// This value is the type tag for a database handle. The command
//
//  uuidgen | sed -r -e 's/-//g' -e 's/{16}(.*)/0x\1, 0x\2/'
//
// can be used to obtain the two values with which to initialize the structure.
static const napi_type_tag DatabaseHandleTypeTag = {
    0x1edf75a38336451d, 0xa5ed9ce2e4c00c38
};

// This value is the type tag for a query handle.
static const napi_type_tag QueryHandleTypeTag = {
    0x9c73317f9fad44a3, 0x93c3920bf3b0ad6a
};

static napi_value
openDatabase(napi_env env, napi_callback_info info) {
    napi_status status;
    napi_value result;

    // Perform the underlying action which results in a database handle.
    DatabaseHandle* dbHandle = open_database();

    // Create a new, empty JS object.
    status = napi_create_object(env, &result);
    if (status != napi_ok) return NULL;

    // Tag the object to indicate that it holds a pointer to a `DatabaseHandle`.
    status = napi_type_tag_object(env, result, &DatabaseHandleTypeTag);
    if (status != napi_ok) return NULL;

    // Store the pointer to the `DatabaseHandle` structure inside the JS object.
    status = napi_wrap(env, result, dbHandle, NULL, NULL, NULL);
    if (status != napi_ok) return NULL;

    return result;
}

// Later when we receive a JavaScript object purporting to be a database handle
// we can use `napi_check_object_type_tag()` to ensure that it is indeed such a
// handle.
```

```

static napi_value
query(napi_env env, napi_callback_info info) {
    napi_status status;
    size_t argc = 2;
    napi_value argv[2];
    bool is_db_handle;

    status = napi_get_cb_info(env, info, &argc, argv, NULL, NULL);
    if (status != napi_ok) return NULL;

    // Check that the object passed as the first parameter has the previously
    // applied tag.
    status = napi_check_object_type_tag(env,
                                        argv[0],
                                        &DatabaseHandleTypeTag,
                                        &is_db_handle);

    if (status != napi_ok) return NULL;

    // Throw a `TypeError` if it doesn't.
    if (!is_db_handle) {
        // Throw a TypeError.
        return NULL;
    }
}

```

napi_define_class

```

napi_status napi_define_class(napi_env env,
                             const char* utf8name,
                             size_t length,
                             napi_callback constructor,
                             void* data,
                             size_t property_count,
                             const napi_property_descriptor* properties,
                             napi_value* result);

```

- [in] env : The environment that the API is invoked under.
- [in] utf8name : Name of the JavaScript constructor function; When wrapping a C++ class, we recommend for clarity that this name be the same as that of the C++ class.
- [in] length : The length of the utf8name in bytes, or NAPI_AUTO_LENGTH if it is null-terminated.
- [in] constructor : Callback function that handles constructing instances of the class. When wrapping a C++ class, this method must be a static member with the napi_callback signature. A C++ class constructor cannot be used. napi_callback provides more details.
- [in] data : Optional data to be passed to the constructor callback as the data property of the callback info.
- [in] property_count : Number of items in the properties array argument.
- [in] properties : Array of property descriptors describing static and instance data properties, accessors, and methods on the class See napi_property_descriptor .
- [out] result : A napi_value representing the constructor function for the class.

Returns napi_ok if the API succeeded.

Defines a JavaScript class, including:

- A JavaScript constructor function that has the class name. When wrapping a corresponding C++ class, the callback passed via `constructor` can be used to instantiate a new C++ class instance, which can then be placed inside the JavaScript object instance being constructed using `napi_wrap`.
- Properties on the constructor function whose implementation can call corresponding *static* data properties, accessors, and methods of the C++ class (defined by property descriptors with the `napi_static` attribute).
- Properties on the constructor function's `prototype` object. When wrapping a C++ class, *non-static* data properties, accessors, and methods of the C++ class can be called from the static functions given in the property descriptors without the `napi_static` attribute after retrieving the C++ class instance placed inside the JavaScript object instance by using `napi_unwrap`.

When wrapping a C++ class, the C++ constructor callback passed via `constructor` should be a static method on the class that calls the actual class constructor, then wraps the new C++ instance in a JavaScript object, and returns the wrapper object. See `napi_wrap` for details.

The JavaScript constructor function returned from `napi_define_class` is often saved and used later to construct new instances of the class from native code, and/or to check whether provided values are instances of the class. In that case, to prevent the function value from being garbage-collected, a strong persistent reference to it can be created using `napi_create_reference`, ensuring that the reference count is kept ≥ 1 .

Any non- `NULL` data which is passed to this API via the `data` parameter or via the `data` field of the `napi_property_descriptor` array items can be associated with the resulting JavaScript constructor (which is returned in the `result` parameter) and freed whenever the class is garbage-collected by passing both the JavaScript function and the data to `napi_add_finalizer`.

`napi_wrap`

```
napi_status napi_wrap(napi_env env,
                     napi_value js_object,
                     void* native_object,
                     napi_finalize finalize_cb,
                     void* finalize_hint,
                     napi_ref* result);
```

- [in] `env`: The environment that the API is invoked under.
- [in] `js_object`: The JavaScript object that will be the wrapper for the native object.
- [in] `native_object`: The native instance that will be wrapped in the JavaScript object.
- [in] `finalize_cb`: Optional native callback that can be used to free the native instance when the JavaScript object is ready for garbage-collection. `napi_finalize` provides more details.
- [in] `finalize_hint`: Optional contextual hint that is passed to the finalize callback.
- [out] `result`: Optional reference to the wrapped object.

Returns `napi_ok` if the API succeeded.

Wraps a native instance in a JavaScript object. The native instance can be retrieved later using `napi_unwrap()`.

When JavaScript code invokes a constructor for a class that was defined using `napi_define_class()`, the `napi_callback` for the constructor is invoked. After constructing an instance of the native class, the callback must then call `napi_wrap()` to wrap the newly constructed instance in the already-created JavaScript object that is the `this` argument to the constructor callback. (That `this` object was created from the constructor function's `prototype`, so it already has definitions of all the instance properties and methods.)

Typically when wrapping a class instance, a finalize callback should be provided that simply deletes the native instance that is received as the `data` argument to the finalize callback.

The optional returned reference is initially a weak reference, meaning it has a reference count of 0. Typically this reference count would be incremented temporarily during async operations that require the instance to remain valid.

Caution: The optional returned reference (if obtained) should be deleted via `napi_delete_reference` ONLY in response to the finalize callback invocation. If it is deleted before then, then the finalize callback may never be invoked. Therefore, when obtaining a reference a finalize callback is also required in order to enable correct disposal of the reference.

Calling `napi_wrap()` a second time on an object will return an error. To associate another native instance with the object, use `napi_remove_wrap()` first.

`napi_unwrap`

```
napi_status napi_unwrap(napi_env env,
                        napi_value js_object,
                        void** result);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] js_object` : The object associated with the native instance.
- `[out] result` : Pointer to the wrapped native instance.

Returns `napi_ok` if the API succeeded.

Retrieves a native instance that was previously wrapped in a JavaScript object using `napi_wrap()`.

When JavaScript code invokes a method or property accessor on the class, the corresponding `napi_callback` is invoked. If the callback is for an instance method or accessor, then the `this` argument to the callback is the wrapper object; the wrapped C++ instance that is the target of the call can be obtained then by calling `napi_unwrap()` on the wrapper object.

`napi_remove_wrap`

```
napi_status napi_remove_wrap(napi_env env,
                             napi_value js_object,
                             void** result);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] js_object` : The object associated with the native instance.
- `[out] result` : Pointer to the wrapped native instance.

Returns `napi_ok` if the API succeeded.

Retrieves a native instance that was previously wrapped in the JavaScript object `js_object` using `napi_wrap()` and removes the wrapping. If a finalize callback was associated with the wrapping, it will no longer be called when the JavaScript object becomes garbage-collected.

`napi_type_tag_object`

```
napi_status napi_type_tag_object(napi_env env,
                                napi_value js_object,
                                const napi_type_tag* type_tag);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] js_object` : The JavaScript object to be marked.

- `[in] type_tag` : The tag with which the object is to be marked.

Returns `napi_ok` if the API succeeded.

Associates the value of the `type_tag` pointer with the JavaScript object. `napi_check_object_type_tag()` can then be used to compare the tag that was attached to the object with one owned by the addon to ensure that the object has the right type.

If the object already has an associated type tag, this API will return `napi_invalid_arg`.

napi_check_object_type_tag

```
napi_status napi_check_object_type_tag(napi_env env,
                                      napi_value js_object,
                                      const napi_type_tag* type_tag,
                                      bool* result);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] js_object` : The JavaScript object whose type tag to examine.
- `[in] type_tag` : The tag with which to compare any tag found on the object.
- `[out] result` : Whether the type tag given matched the type tag on the object. `false` is also returned if no type tag was found on the object.

Returns `napi_ok` if the API succeeded.

Compares the pointer given as `type_tag` with any that can be found on `js_object`. If no tag is found on `js_object` or, if a tag is found but it does not match `type_tag`, then `result` is set to `false`. If a tag is found and it matches `type_tag`, then `result` is set to `true`.

napi_add_finalizer

```
napi_status napi_add_finalizer(napi_env env,
                              napi_value js_object,
                              void* native_object,
                              napi_finalize finalize_cb,
                              void* finalize_hint,
                              napi_ref* result);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] js_object` : The JavaScript object to which the native data will be attached.
- `[in] native_object` : The native data that will be attached to the JavaScript object.
- `[in] finalize_cb` : Native callback that will be used to free the native data when the JavaScript object is ready for garbage-collection. `napi_finalize` provides more details.
- `[in] finalize_hint` : Optional contextual hint that is passed to the finalize callback.
- `[out] result` : Optional reference to the JavaScript object.

Returns `napi_ok` if the API succeeded.

Adds a `napi_finalize` callback which will be called when the JavaScript object in `js_object` is ready for garbage collection. This API is similar to `napi_wrap()` except that:

- the native data cannot be retrieved later using `napi_unwrap()`,
- nor can it be removed later using `napi_remove_wrap()`, and
- the API can be called multiple times with different data items in order to attach each of them to the JavaScript object, and

- the object manipulated by the API can be used with `napi_wrap()`.

Caution: The optional returned reference (if obtained) should be deleted via `napi_delete_reference` ONLY in response to the finalize callback invocation. If it is deleted before then, then the finalize callback may never be invoked. Therefore, when obtaining a reference a finalize callback is also required in order to enable correct disposal of the reference.

Simple asynchronous operations

Addon modules often need to leverage async helpers from libuv as part of their implementation. This allows them to schedule work to be executed asynchronously so that their methods can return in advance of the work being completed. This allows them to avoid blocking overall execution of the Node.js application.

Node-API provides an ABI-stable interface for these supporting functions which covers the most common asynchronous use cases.

Node-API defines the `napi_async_work` structure which is used to manage asynchronous workers. Instances are created/deleted with `napi_create_async_work` and `napi_delete_async_work`.

The `execute` and `complete` callbacks are functions that will be invoked when the executor is ready to execute and when it completes its task respectively.

The `execute` function should avoid making any Node-API calls that could result in the execution of JavaScript or interaction with JavaScript objects. Most often, any code that needs to make Node-API calls should be made in `complete` callback instead. Avoid using the `napi_env` parameter in the execute callback as it will likely execute JavaScript.

These functions implement the following interfaces:

```
typedef void (*napi_async_execute_callback)(napi_env env,
                                             void* data);
typedef void (*napi_async_complete_callback)(napi_env env,
                                             napi_status status,
                                             void* data);
```

When these methods are invoked, the `data` parameter passed will be the addon-provided `void*` data that was passed into the `napi_create_async_work` call.

Once created the async worker can be queued for execution using the `napi_queue_async_work` function:

```
napi_status napi_queue_async_work(napi_env env,  
                                   napi_async_work work);
```

`napi_cancel_async_work` can be used if the work needs to be cancelled before the work has started execution.

After calling `napi_cancel_async_work`, the `complete` callback will be invoked with a status value of `napi_cancelled`. The work should not be deleted before the `complete` callback invocation, even when it was cancelled.

napi_create_async_work

[illegible]

```
void* data,  
napi_async_work* result);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] async_resource` : An optional object associated with the async work that will be passed to possible `async_hooks` `init` hooks.
- `[in] async_resource_name` : Identifier for the kind of resource that is being provided for diagnostic information exposed by the `async_hooks` API.
- `[in] execute` : The native function which should be called to execute the logic asynchronously. The given function is called from a worker pool thread and can execute in parallel with the main event loop thread.
- `[in] complete` : The native function which will be called when the asynchronous logic is completed or is cancelled. The given function is called from the main event loop thread. `napi_async_complete_callback` provides more details.
- `[in] data` : User-provided data context. This will be passed back into the execute and complete functions.
- `[out] result` : `napi_async_work*` which is the handle to the newly created async work.

Returns `napi_ok` if the API succeeded.

This API allocates a work object that is used to execute logic asynchronously. It should be freed using `napi_delete_async_work` once the work is no longer required.

`async_resource_name` should be a null-terminated, UTF-8-encoded string.

The `async_resource_name` identifier is provided by the user and should be representative of the type of async work being performed. It is also recommended to apply namespacing to the identifier, e.g. by including the module name. See the [async_hooks documentation](#) for more information.

`napi_delete_async_work`

```
napi_status napi_delete_async_work(napi_env env,  
napi_async_work work);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] work` : The handle returned by the call to `napi_create_async_work`.

Returns `napi_ok` if the API succeeded.

This API frees a previously allocated work object.

This API can be called even if there is a pending JavaScript exception.

`napi_queue_async_work`

```
napi_status napi_queue_async_work(napi_env env,  
napi_async_work work);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] work` : The handle returned by the call to `napi_create_async_work`.

Returns `napi_ok` if the API succeeded.

This API requests that the previously allocated work be scheduled for execution. Once it returns successfully, this API must not be called again with the same `napi_async_work` item or the result will be undefined.

napi_cancel_async_work

```
napi_status napi_cancel_async_work(napi_env env,
                                   napi_async_work work);
```

- [in] `env` : The environment that the API is invoked under.
- [in] `work` : The handle returned by the call to `napi_create_async_work`.

Returns `napi_ok` if the API succeeded.

This API cancels queued work if it has not yet been started. If it has already started executing, it cannot be cancelled and `napi_generic_failure` will be returned. If successful, the `complete` callback will be invoked with a status value of `napi_cancelled`. The work should not be deleted before the `complete` callback invocation, even if it has been successfully cancelled.

This API can be called even if there is a pending JavaScript exception.

Custom asynchronous operations

The simple asynchronous work APIs above may not be appropriate for every scenario. When using any other asynchronous mechanism, the following APIs are necessary to ensure an asynchronous operation is properly tracked by the runtime.

napi_async_init

```
napi_status napi_async_init(napi_env env,
                            napi_value async_resource,
                            napi_value async_resource_name,
                            napi_async_context* result)
```

- [in] `env` : The environment that the API is invoked under.
- [in] `async_resource` : Object associated with the async work that will be passed to possible `async_hooks` `init` hooks and can be accessed by `async_hooks.executionAsyncResource()`.
- [in] `async_resource_name` : Identifier for the kind of resource that is being provided for diagnostic information exposed by the `async_hooks` API.
- [out] `result` : The initialized async context.

Returns `napi_ok` if the API succeeded.

The `async_resource` object needs to be kept alive until `napi_async_destroy` to keep `async_hooks` related API acts correctly. In order to retain ABI compatibility with previous versions, `napi_async_context` s are not maintaining the strong reference to the `async_resource` objects to avoid introducing causing memory leaks. However, if the `async_resource` is garbage collected by JavaScript engine before the `napi_async_context` was destroyed by `napi_async_destroy`, calling `napi_async_context` related APIs like `napi_open_callback_scope` and `napi_make_callback` can cause problems like loss of async context when using the `AsyncLocalStorage` API.

In order to retain ABI compatibility with previous versions, passing `NULL` for `async_resource` does not result in an error. However, this is not recommended as this will result poor results with `async_hooks` `init` hooks and `async_hooks.executionAsyncResource()` as the resource is now required by the underlying `async_hooks` implementation in order to provide the linkage between async callbacks.

napi_async_destroy

[illegible]

- `[in] env` : The environment that the API is invoked under.
- `[in] async_context` : The async context to be destroyed.

Returns `napi_ok` if the API succeeded.

This API can be called even if there is a pending JavaScript exception.

napi_make_callback

[illegible]

- [in] `env` : The environment that the API is invoked under.
- [in] `async_context` : Context for the async operation that is invoking the callback. This should normally be a value previously obtained from `napi_async_init` . In order to retain ABI compatibility with previous versions, passing `NULL` for `async_context` does not result in an error. However, this results in incorrect operation of async hooks. Potential issues include loss of async context when using the `AsyncLocalStorage` API.
- [in] `recv` : The `this` value passed to the called function.
- [in] `func` : `napi_value` representing the JavaScript function to be invoked.
- [in] `argc` : The count of elements in the `argv` array.
- [in] `argv` : Array of JavaScript values as `napi_value` representing the arguments to the function.
- [out] `result` : `napi_value` representing the JavaScript object returned.

Returns `napi_ok` if the API succeeded.

This method allows a JavaScript function object to be called from a native add-on. This API is similar to `napi_call_function`. However, it is used to call *from* native code back *into* JavaScript *after* returning from an async operation (when there is no other script on the stack). It is a fairly simple wrapper around `node::MakeCallback`.

Note it is *not* necessary to use `napi_make_callback` from within a `napi_async_complete_callback`; in that situation the callback's async context has already been set up, so a direct call to `napi_call_function` is sufficient and appropriate. Use of the `napi_make_callback` function may be required when implementing custom async behavior that does not use `napi_create_async_work`.

Any `process.nextTick`s or Promises scheduled on the microtask queue by JavaScript during the callback are ran before returning back to C/C++.

napi_open_callback_scope

[illegible]

- `[in] env` : The environment that the API is invoked under.
- `[in] resource_object` : An object associated with the async work that will be passed to possible `async_hooks` `init` hooks . This parameter has been deprecated and is ignored at runtime. Use the `async_resource` parameter in `napi_async_init` instead.
- `[in] context` : Context for the async operation that is invoking the callback. This should be a value previously obtained from `napi_async_init` .
- `[out] result` : The newly created scope.

There are cases (for example, resolving promises) where it is necessary to have the equivalent of the scope associated with a callback in place when making certain Node-API calls. If there is no other script on the stack the `napi_open_callback_scope` and `napi_close_callback_scope` functions can be used to open/close the required scope.

napi_close_callback_scope

```

NAPI_EXTERN napi_status napi_close_callback_scope(napi_env env,
                                                  napi_callback_scope scope)

```

- `[in] env` : The environment that the API is invoked under.
- `[in] scope` : The scope to be closed.

This API can be called even if there is a pending JavaScript exception.

Version management

napi_get_node_version

```

typedef struct {
    uint32_t major;
    uint32_t minor;
    uint32_t patch;
    const char* release;
} napi_node_version;

napi_status napi_get_node_version(napi_env env,
                                  const napi_node_version** version);

```

- `[in] env` : The environment that the API is invoked under.
- `[out] version` : A pointer to version information for Node.js itself.

Returns `napi_ok` if the API succeeded.

This function fills the `version` struct with the major, minor, and patch version of Node.js that is currently running, and the `release` field with the value of `process.release.name` .

The returned buffer is statically allocated and does not need to be freed.

napi_get_version

```

napi_status napi_get_version(napi_env env,
                             uint32_t* result);

```

- `[in] env` : The environment that the API is invoked under.

- `[out] result` : The highest version of Node-API supported.

Returns `napi_ok` if the API succeeded.

This API returns the highest Node-API version supported by the Node.js runtime. Node-API is planned to be additive such that newer releases of Node.js may support additional API functions. In order to allow an addon to use a newer function when running with versions of Node.js that support it, while providing fallback behavior when running with Node.js versions that don't support it:

- Call `napi_get_version()` to determine if the API is available.
- If available, dynamically load a pointer to the function using `uv_dlsym()`.
- Use the dynamically loaded pointer to invoke the function.
- If the function is not available, provide an alternate implementation that does not use the function.

Memory management

`napi_adjust_external_memory`

```
NAPI_EXTERN napi_status napi_adjust_external_memory(napi_env env,
                                                    int64_t change_in_bytes,
                                                    int64_t* result);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] change_in_bytes` : The change in externally allocated memory that is kept alive by JavaScript objects.
- `[out] result` : The adjusted value

Returns `napi_ok` if the API succeeded.

This function gives V8 an indication of the amount of externally allocated memory that is kept alive by JavaScript objects (i.e. a JavaScript object that points to its own memory allocated by a native module). Registering externally allocated memory will trigger global garbage collections more often than it would otherwise.

Promises

Node-API provides facilities for creating `Promise` objects as described in [Section 25.4](#) of the ECMA specification. It implements promises as a pair of objects. When a promise is created by `napi_create_promise()`, a "deferred" object is created and returned alongside the `Promise`. The deferred object is bound to the created `Promise` and is the only means to resolve or reject the `Promise` using `napi_resolve_deferred()` or `napi_reject_deferred()`. The deferred object that is created by `napi_create_promise()` is freed by `napi_resolve_deferred()` or `napi_reject_deferred()`. The `Promise` object may be returned to JavaScript where it can be used in the usual fashion.

For example, to create a promise and pass it to an asynchronous worker:

```
napi_deferred deferred;
napi_value promise;
napi_status status;

// Create the promise.
status = napi_create_promise(env, &deferred, &promise);
if (status != napi_ok) return NULL;

// Pass the deferred to a function that performs an asynchronous action.
```



```
do_something_asynchronous(deferred);

// Return the promise to JS
return promise;
```

The above function `do_something_asynchronous()` would perform its asynchronous action and then it would resolve or reject the deferred, thereby concluding the promise and freeing the deferred:

```
napi_deferred deferred;
napi_value undefined;
napi_status status;

// Create a value with which to conclude the deferred.
status = napi_get_undefined(env, &undefined);
if (status != napi_ok) return NULL;

// Resolve or reject the promise associated with the deferred depending on
// whether the asynchronous action succeeded.
if (asynchronous_action_succeeded) {
    status = napi_resolve_deferred(env, deferred, undefined);
} else {
    status = napi_reject_deferred(env, deferred, undefined);
}
if (status != napi_ok) return NULL;

// At this point the deferred has been freed, so we should assign NULL to it.
deferred = NULL;
```

napi_create_promise

```
napi_status napi_create_promise(napi_env env,
                               napi_deferred* deferred,
                               napi_value* promise);
```

- `[in] env`: The environment that the API is invoked under.
- `[out] deferred`: A newly created deferred object which can later be passed to `napi_resolve_deferred()` or `napi_reject_deferred()` to resolve resp. reject the associated promise.
- `[out] promise`: The JavaScript promise associated with the deferred object.

Returns `napi_ok` if the API succeeded.

This API creates a deferred object and a JavaScript promise.

napi_resolve_deferred

```
napi_status napi_resolve_deferred(napi_env env,
                                  napi_deferred deferred,
                                  napi_value resolution);
```

- `[in] env`: The environment that the API is invoked under.

- `[in] deferred` : The deferred object whose associated promise to resolve.
- `[in] resolution` : The value with which to resolve the promise.

This API resolves a JavaScript promise by way of the deferred object with which it is associated. Thus, it can only be used to resolve JavaScript promises for which the corresponding deferred object is available. This effectively means that the promise must have been created using `napi_create_promise()` and the deferred object returned from that call must have been retained in order to be passed to this API.

The deferred object is freed upon successful completion.

napi_reject_deferred

```
napi_status napi_reject_deferred(napi_env env,
                                napi_deferred deferred,
                                napi_value rejection);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] deferred` : The deferred object whose associated promise to resolve.
- `[in] rejection` : The value with which to reject the promise.

This API rejects a JavaScript promise by way of the deferred object with which it is associated. Thus, it can only be used to reject JavaScript promises for which the corresponding deferred object is available. This effectively means that the promise must have been created using `napi_create_promise()` and the deferred object returned from that call must have been retained in order to be passed to this API.

The deferred object is freed upon successful completion.

napi_is_promise

```
napi_status napi_is_promise(napi_env env,
                            napi_value value,
                            bool* is_promise);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : The value to examine
- `[out] is_promise` : Flag indicating whether `promise` is a native promise object (that is, a promise object created by the underlying engine).

Script execution

Node-API provides an API for executing a string containing JavaScript using the underlying JavaScript engine.

napi_run_script

```
NAPI_EXTERN napi_status napi_run_script(napi_env env,
                                        napi_value script,
                                        napi_value* result);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] script` : A JavaScript string containing the script to execute.
- `[out] result` : The value resulting from having executed the script.

This function executes a string of JavaScript code and returns its result with the following caveats:

- Unlike `eval`, this function does not allow the script to access the current lexical scope, and therefore also does not allow to access the `module scope`, meaning that pseudo-globals such as `require` will not be available.
- The script can access the `global scope`. Function and `var` declarations in the script will be added to the `global` object. Variable declarations made using `let` and `const` will be visible globally, but will not be added to the `global` object.
- The value of `this` is `global` within the script.

libuv event loop

Node-API provides a function for getting the current event loop associated with a specific `napi_env`.

napi_get_uv_event_loop

```
NAPI_EXTERN napi_status napi_get_uv_event_loop(napi_env env,
                                              struct uv_loop_s** loop);
```

- `[in] env`: The environment that the API is invoked under.
- `[out] loop`: The current libuv loop instance.

Asynchronous thread-safe function calls

JavaScript functions can normally only be called from a native addon's main thread. If an addon creates additional threads, then Node-API functions that require a `napi_env`, `napi_value`, or `napi_ref` must not be called from those threads.

When an addon has additional threads and JavaScript functions need to be invoked based on the processing completed by those threads, those threads must communicate with the addon's main thread so that the main thread can invoke the JavaScript function on their behalf. The thread-safe function APIs provide an easy way to do this.

These APIs provide the type `napi_threadsafe_function` as well as APIs to create, destroy, and call objects of this type.

`napi_create_threadsafe_function()` creates a persistent reference to a `napi_value` that holds a JavaScript function which can be called from multiple threads. The calls happen asynchronously. This means that values with which the JavaScript callback is to be called will be placed in a queue, and, for each value in the queue, a call will eventually be made to the JavaScript function.

Upon creation of a `napi_threadsafe_function` a `napi_finalize` callback can be provided. This callback will be invoked on the main thread when the thread-safe function is about to be destroyed. It receives the context and the finalize data given during construction, and provides an opportunity for cleaning up after the threads e.g. by calling `uv_thread_join()`. **Aside from the main loop thread, no threads should be using the thread-safe function after the finalize callback completes.**

The `context` given during the call to `napi_create_threadsafe_function()` can be retrieved from any thread with a call to `napi_get_threadsafe_function_context()`.

Calling a thread-safe function

`napi_call_threadsafe_function()` can be used for initiating a call into JavaScript. `napi_call_threadsafe_function()` accepts a parameter which controls whether the API behaves blockingly. If set to `napi_tsf_noblocking`, the API behaves non-blockingly, returning `napi_queue_full` if the queue was full, preventing data from being successfully added to the queue. If set to `napi_tsf_blocking`, the API blocks until space becomes available in the queue. `napi_call_threadsafe_function()` never blocks if the thread-safe function was created with a maximum queue size of 0.

`napi_call_threadsafe_function()` should not be called with `napi_tsf_blocking` from a JavaScript thread, because, if the queue is full, it may cause the JavaScript thread to deadlock.

The actual call into JavaScript is controlled by the callback given via the `call_js_cb` parameter. `call_js_cb` is invoked on the main thread once for each value that was placed into the queue by a successful call to `napi_call_threadsafe_function()`. If such a callback is not

given, a default callback will be used, and the resulting JavaScript call will have no arguments. The `call_js_cb` callback receives the JavaScript function to call as a `napi_value` in its parameters, as well as the `void*` context pointer used when creating the `napi_threadsafe_function`, and the next data pointer that was created by one of the secondary threads. The callback can then use an API such as `napi_call_function()` to call into JavaScript.

The callback may also be invoked with `env` and `call_js_cb` both set to `NULL` to indicate that calls into JavaScript are no longer possible, while items remain in the queue that may need to be freed. This normally occurs when the Node.js process exits while there is a thread-safe function still active.

It is not necessary to call into JavaScript via `napi_make_callback()` because Node-API runs `call_js_cb` in a context appropriate for callbacks.

Reference counting of thread-safe functions

Threads can be added to and removed from a `napi_threadsafe_function` object during its existence. Thus, in addition to specifying an initial number of threads upon creation, `napi_acquire_threadsafe_function` can be called to indicate that a new thread will start making use of the thread-safe function. Similarly, `napi_release_threadsafe_function` can be called to indicate that an existing thread will stop making use of the thread-safe function.

`napi_threadsafe_function` objects are destroyed when every thread which uses the object has called `napi_release_threadsafe_function()` or has received a return status of `napi_closing` in response to a call to `napi_call_threadsafe_function`. The queue is emptied before the `napi_threadsafe_function` is destroyed. `napi_release_threadsafe_function()` should be the last API call made in conjunction with a given `napi_threadsafe_function`, because after the call completes, there is no guarantee that the `napi_threadsafe_function` is still allocated. For the same reason, do not use a thread-safe function after receiving a return value of `napi_closing` in response to a call to `napi_call_threadsafe_function`. Data associated with the `napi_threadsafe_function` can be freed in its `napi_finalize` callback which was passed to `napi_create_threadsafe_function()`. The parameter `initial_thread_count` of `napi_create_threadsafe_function` marks the initial number of acquisitions of the thread-safe functions, instead of calling `napi_acquire_threadsafe_function` multiple times at creation.

Once the number of threads making use of a `napi_threadsafe_function` reaches zero, no further threads can start making use of it by calling `napi_acquire_threadsafe_function()`. In fact, all subsequent API calls associated with it, except `napi_release_threadsafe_function()`, will return an error value of `napi_closing`.

The thread-safe function can be "aborted" by giving a value of `napi_tsf_abort` to `napi_release_threadsafe_function()`. This will cause all subsequent APIs associated with the thread-safe function except `napi_release_threadsafe_function()` to return `napi_closing` even before its reference count reaches zero. In particular, `napi_call_threadsafe_function()` will return `napi_closing`, thus informing the threads that it is no longer possible to make asynchronous calls to the thread-safe function. This can be used as a criterion for terminating the thread. **Upon receiving a return value of `napi_closing` from `napi_call_threadsafe_function()` a thread must not use the thread-safe function anymore because it is no longer guaranteed to be allocated.**

Deciding whether to keep the process running

Similarly to libuv handles, thread-safe functions can be "referenced" and "unreferenced". A "referenced" thread-safe function will cause the event loop on the thread on which it is created to remain alive until the thread-safe function is destroyed. In contrast, an "unreferenced" thread-safe function will not prevent the event loop from exiting. The APIs `napi_ref_threadsafe_function` and `napi_unref_threadsafe_function` exist for this purpose.

Neither does `napi_unref_threadsafe_function` mark the thread-safe functions as able to be destroyed nor does `napi_ref_threadsafe_function` prevent it from being destroyed.

`napi_create_threadsafe_function`

```

NAPI_EXTERN napi_status
napi_create_threadsafe_function(napi_env env,
                               napi_value func,
                               napi_value async_resource,
                               napi_value async_resource_name,
                               size_t max_queue_size,
                               size_t initial_thread_count,
                               void* thread_finalize_data,
                               napi_finalize thread_finalize_cb,
                               void* context,
                               napi_threadsafe_function_call_js call_js_cb,
                               napi_threadsafe_function* result);

```

- [in] `env` : The environment that the API is invoked under.
- [in] `func` : An optional JavaScript function to call from another thread. It must be provided if `NULL` is passed to `call_js_cb`.
- [in] `async_resource` : An optional object associated with the async work that will be passed to possible `async_hooks` `init hooks`.
- [in] `async_resource_name` : A JavaScript string to provide an identifier for the kind of resource that is being provided for diagnostic information exposed by the `async_hooks` API.
- [in] `max_queue_size` : Maximum size of the queue. `0` for no limit.
- [in] `initial_thread_count` : The initial number of acquisitions, i.e. the initial number of threads, including the main thread, which will be making use of this function.
- [in] `thread_finalize_data` : Optional data to be passed to `thread_finalize_cb`.
- [in] `thread_finalize_cb` : Optional function to call when the `napi_threadsafe_function` is being destroyed.
- [in] `context` : Optional data to attach to the resulting `napi_threadsafe_function`.
- [in] `call_js_cb` : Optional callback which calls the JavaScript function in response to a call on a different thread. This callback will be called on the main thread. If not given, the JavaScript function will be called with no parameters and with `undefined` as its `this` value. `napi_threadsafe_function_call_js` provides more details.
- [out] `result` : The asynchronous thread-safe JavaScript function.

napi_get_threadsafe_function_context

```

NAPI_EXTERN napi_status
napi_get_threadsafe_function_context(napi_threadsafe_function func,
                                     void** result);

```

- [in] `func` : The thread-safe function for which to retrieve the context.
- [out] `result` : The location where to store the context.

This API may be called from any thread which makes use of `func`.

napi_call_threadsafe_function

```

NAPI_EXTERN napi_status
napi_call_threadsafe_function(napi_threadsafe_function func,
                              void* data,
                              napi_threadsafe_function_call_mode is_blocking);

```

- [in] `func` : The asynchronous thread-safe JavaScript function to invoke.

- `[in] data` : Data to send into JavaScript via the callback `call_js_cb` provided during the creation of the thread-safe JavaScript function.
- `[in] is_blocking` : Flag whose value can be either `napi_tsfn_blocking` to indicate that the call should block if the queue is full or `napi_tsfn_nonblocking` to indicate that the call should return immediately with a status of `napi_queue_full` whenever the queue is full.

This API should not be called with `napi_tsfn_blocking` from a JavaScript thread, because, if the queue is full, it may cause the JavaScript thread to deadlock.

This API will return `napi_closing` if `napi_release_threadsafe_function()` was called with `abort` set to `napi_tsfn_abort` from any thread. The value is only added to the queue if the API returns `napi_ok`.

This API may be called from any thread which makes use of `func`.

napi_acquire_threadsafe_function

```
NAPI_EXTERN napi_status
napi_acquire_threadsafe_function(napi_threadsafe_function func);
```

- `[in] func` : The asynchronous thread-safe JavaScript function to start making use of.

A thread should call this API before passing `func` to any other thread-safe function APIs to indicate that it will be making use of `func`. This prevents `func` from being destroyed when all other threads have stopped making use of it.

This API may be called from any thread which will start making use of `func`.

napi_release_threadsafe_function

```
NAPI_EXTERN napi_status
napi_release_threadsafe_function(napi_threadsafe_function func,
                                napi_threadsafe_function_release_mode mode);
```

- `[in] func` : The asynchronous thread-safe JavaScript function whose reference count to decrement.
- `[in] mode` : Flag whose value can be either `napi_tsfn_release` to indicate that the current thread will make no further calls to the thread-safe function, or `napi_tsfn_abort` to indicate that in addition to the current thread, no other thread should make any further calls to the thread-safe function. If set to `napi_tsfn_abort`, further calls to `napi_call_threadsafe_function()` will return `napi_closing`, and no further values will be placed in the queue.

A thread should call this API when it stops making use of `func`. Passing `func` to any thread-safe APIs after having called this API has undefined results, as `func` may have been destroyed.

This API may be called from any thread which will stop making use of `func`.

napi_ref_threadsafe_function

```
NAPI_EXTERN napi_status
napi_ref_threadsafe_function(napi_env env, napi_threadsafe_function func);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] func` : The thread-safe function to reference.

This API is used to indicate that the event loop running on the main thread should not exit until `func` has been destroyed. Similar to `uv_ref` it is also idempotent.

Neither does `napi_unref_threadsafe_function` mark the thread-safe functions as able to be destroyed nor does `napi_ref_threadsafe_function` prevent it from being destroyed. `napi_acquire_threadsafe_function` and `napi_release_threadsafe_function` are available for that purpose.

This API may only be called from the main thread.

napi_unref_threadsafe_function

```
NAPI_EXTERN napi_status  
napi_unref_threadsafe_function(napi_env env, napi_threadsafe_function func);
```

- `[in] env`: The environment that the API is invoked under.
- `[in] func`: The thread-safe function to unreference.

This API is used to indicate that the event loop running on the main thread may exit before `func` is destroyed. Similar to `uv_unref` it is also idempotent.

This API may only be called from the main thread.

Miscellaneous utilities

node_api_get_module_file_name

Stability: 1 - Experimental

```
NAPI_EXTERN napi_status  
node_api_get_module_file_name(napi_env env, const char** result);
```

- `[in] env`: The environment that the API is invoked under.
- `[out] result`: A URL containing the absolute path of the location from which the add-on was loaded. For a file on the local file system it will start with `file://`. The string is null-terminated and owned by `env` and must thus not be modified or freed.

`result` may be an empty string if the add-on loading process fails to establish the add-on's file name during loading.

C++ embedder API

Node.js provides a number of C++ APIs that can be used to execute JavaScript in a Node.js environment from other C++ software.

The documentation for these APIs can be found in `src/node.h` in the Node.js source tree. In addition to the APIs exposed by Node.js, some required concepts are provided by the V8 embedder API.

Because using Node.js as an embedded library is different from writing code that is executed by Node.js, breaking changes do not follow typical Node.js [deprecation policy](#) and may occur on each semver-major release without prior warning.

Example embedding application

The following sections will provide an overview over how to use these APIs to create an application from scratch that will perform the equivalent of `node -e <code>`, i.e. that will take a piece of JavaScript and run it in a Node.js-specific environment.

The full code can be found [in the Node.js source tree](#) .

Setting up per-process state

Node.js requires some per-process state management in order to run:

- Arguments parsing for Node.js `CLI options` ,
- V8 per-process requirements, such as a `v8::Platform` instance.

The following example shows how these can be set up. Some class names are from the `node` and `v8` C++ namespaces, respectively.

```
int main(int argc, char** argv) {
    argv = uv_setup_args(argc, argv);
    std::vector<std::string> args(argv, argv + argc);
    std::vector<std::string> exec_args;
    std::vector<std::string> errors;
    // Parse Node.js CLI options, and print any errors that have occurred while
    // trying to parse them.
    int exit_code = node::InitializeNodeWithArgs(&args, &exec_args, &errors);
    for (const std::string& error : errors)
        fprintf(stderr, "%s: %s\n", args[0].c_str(), error.c_str());
    if (exit_code != 0) {
        return exit_code;
    }

    // Create a v8::Platform instance. `MultiIsolatePlatform::Create()` is a way
    // to create a v8::Platform instance that Node.js can use when creating
    // Worker threads. When no `MultiIsolatePlatform` instance is present,
    // Worker threads are disabled.
    std::unique_ptr<MultiIsolatePlatform> platform =
        MultiIsolatePlatform::Create(4);
    V8::InitializePlatform(platform.get());
    V8::Initialize();

    // See below for the contents of this function.
    int ret = RunNodeInstance(platform.get(), args, exec_args);

    V8::Dispose();
    V8::ShutdownPlatform();
    return ret;
}
```

Per-instance state

Node.js has a concept of a “Node.js instance”, that is commonly being referred to as `node::Environment` . Each `node::Environment` is associated with:

- Exactly one `v8::Isolate` , i.e. one JS Engine instance,
- Exactly one `uv_loop_t` , i.e. one event loop, and
- A number of `v8::Context` s, but exactly one main `v8::Context` .
- One `node::IsolateData` instance that contains information that could be shared by multiple `node::Environment` s that use the same `v8::Isolate` . Currently, no testing is performed for this scenario.

In order to set up a `v8::Isolate`, an `v8::ArrayBuffer::Allocator` needs to be provided. One possible choice is the default Node.js allocator, which can be created through `node::ArrayBufferAllocator::Create()`. Using the Node.js allocator allows minor performance optimizations when addons use the Node.js C++ `Buffer` API, and is required in order to track `ArrayBuffer` memory in `process.memoryUsage()`.

Additionally, each `v8::Isolate` that is used for a Node.js instance needs to be registered and unregistered with the `MultiIsolatePlatform` instance, if one is being used, in order for the platform to know which event loop to use for tasks scheduled by the `v8::Isolate`.

The `node::NewIsolate()` helper function creates a `v8::Isolate`, sets it up with some Node.js-specific hooks (e.g. the Node.js error handler), and registers it with the platform automatically.

```
int RunNodeInstance(MultiIsolatePlatform* platform,
                   const std::vector<std::string>& args,
                   const std::vector<std::string>& exec_args) {
    int exit_code = 0;

    // Setup up a libuv event loop, v8::Isolate, and Node.js Environment.
    std::vector<std::string> errors;
    std::unique_ptr<CommonEnvironmentSetup> setup =
        CommonEnvironmentSetup::Create(platform, &errors, args, exec_args);
    if (!setup) {
        for (const std::string& err : errors)
            fprintf(stderr, "%s: %s\n", args[0].c_str(), err.c_str());
        return 1;
    }

    Isolate* isolate = setup->isolate();
    Environment* env = setup->env();

    {
        Locker locker(isolate);
        Isolate::Scope isolate_scope(isolate);
        // The v8::Context needs to be entered when node::CreateEnvironment() and
        // node::LoadEnvironment() are being called.
        Context::Scope context_scope(setup->context());

        // Set up the Node.js instance for execution, and run code inside of it.
        // There is also a variant that takes a callback and provides it with
        // the `require` and `process` objects, so that it can manually compile
        // and run scripts as needed.
        // The `require` function inside this script does *not* access the file
        // system, and can only load built-in Node.js modules.
        // `module.createRequire()` is being used to create one that is able to
        // load files from the disk, and uses the standard CommonJS file loader
        // instead of the internal-only `require` function.
        MaybeLocal<Value> loadenv_ret = node::LoadEnvironment(
            env,
            "const publicRequire ="
            "  require('module').createRequire(process.cwd() + '/');"
            "globalThis.require = publicRequire;");
    }
```

```

    "require('vm').runInThisContext(process.argv[1]);");

    if (loadenv_ret.IsEmpty()) // There has been a JS exception.
        return 1;

    exit_code = node::SpinEventLoop(env).FromMaybe(1);

    // node::Stop() can be used to explicitly stop the event loop and keep
    // further JavaScript from running. It can be called from any thread,
    // and will act like worker.terminate() if called from another thread.
    node::Stop(env);
}

return exit_code;
}

```

Child process

Stability: 2 - Stable

Source Code: [lib/child_process.js](#)

The `child_process` module provides the ability to spawn subprocesses in a manner that is similar, but not identical, to `popen(3)`. This capability is primarily provided by the `child_process.spawn()` function:

```

const { spawn } = require('child_process');
const ls = spawn('ls', ['-lh', '/usr']);

ls.stdout.on('data', (data) => {
    console.log(`stdout: ${data}`);
});

ls.stderr.on('data', (data) => {
    console.error(`stderr: ${data}`);
});

ls.on('close', (code) => {
    console.log(`child process exited with code ${code}`);
});

```

By default, pipes for `stdin`, `stdout`, and `stderr` are established between the parent Node.js process and the spawned subprocess. These pipes have limited (and platform-specific) capacity. If the subprocess writes to `stdout` in excess of that limit without the output being captured, the subprocess blocks waiting for the pipe buffer to accept more data. This is identical to the behavior of pipes in the shell. Use the `{ stdio: 'ignore' }` option if the output will not be consumed.

The command lookup is performed using the `options.env.PATH` environment variable if it is in the `options` object. Otherwise, `process.env.PATH` is used.

On Windows, environment variables are case-insensitive. Node.js lexicographically sorts the `env` keys and uses the first one that case-insensitively matches. Only first (in lexicographic order) entry will be passed to the subprocess. This might lead to issues on Windows when passing objects to the `env` option that have multiple variants of the same key, such as `PATH` and `Path`.

The `child_process.spawn()` method spawns the child process asynchronously, without blocking the Node.js event loop. The `child_process.spawnSync()` function provides equivalent functionality in a synchronous manner that blocks the event loop until the spawned process either exits or is terminated.

For convenience, the `child_process` module provides a handful of synchronous and asynchronous alternatives to `child_process.spawn()` and `child_process.spawnSync()`. Each of these alternatives are implemented on top of `child_process.spawn()` or `child_process.spawnSync()`.

- `child_process.exec()` : spawns a shell and runs a command within that shell, passing the `stdout` and `stderr` to a callback function when complete.
- `child_process.execFile()` : similar to `child_process.exec()` except that it spawns the command directly without first spawning a shell by default.
- `child_process.fork()` : spawns a new Node.js process and invokes a specified module with an IPC communication channel established that allows sending messages between parent and child.
- `child_process.execSync()` : a synchronous version of `child_process.exec()` that will block the Node.js event loop.
- `child_process.execFileSync()` : a synchronous version of `child_process.execFile()` that will block the Node.js event loop.

For certain use cases, such as automating shell scripts, the `synchronous counterparts` may be more convenient. In many cases, however, the synchronous methods can have significant impact on performance due to stalling the event loop while spawned processes complete.

Asynchronous process creation

The `child_process.spawn()`, `child_process.fork()`, `child_process.exec()`, and `child_process.execFile()` methods all follow the idiomatic asynchronous programming pattern typical of other Node.js APIs.

Each of the methods returns a `ChildProcess` instance. These objects implement the Node.js `EventEmitter` API, allowing the parent process to register listener functions that are called when certain events occur during the life cycle of the child process.

The `child_process.exec()` and `child_process.execFile()` methods additionally allow for an optional `callback` function to be specified that is invoked when the child process terminates.

Spawning `.bat` and `.cmd` files on Windows

The importance of the distinction between `child_process.exec()` and `child_process.execFile()` can vary based on platform. On Unix-type operating systems (Unix, Linux, macOS) `child_process.execFile()` can be more efficient because it does not spawn a shell by default. On Windows, however, `.bat` and `.cmd` files are not executable on their own without a terminal, and therefore cannot be launched using `child_process.execFile()`. When running on Windows, `.bat` and `.cmd` files can be invoked using `child_process.spawn()` with the `shell` option set, with `child_process.exec()`, or by spawning `cmd.exe` and passing the `.bat` or `.cmd` file as an argument (which is what the `shell` option and `child_process.exec()` do). In any case, if the script filename contains spaces it needs to be quoted.

```
// On Windows Only...
const { spawn } = require('child_process');
const bat = spawn('cmd.exe', ['/c', 'my.bat']);

bat.stdout.on('data', (data) => {
  console.log(data.toString());
});

bat.stderr.on('data', (data) => {
```

```

    console.error(data.toString());
  });

  bat.on('exit', (code) => {
    console.log(`Child exited with code ${code}`);
  });

```

```

// OR...
const { exec, spawn } = require('child_process');
exec('my.bat', (err, stdout, stderr) => {
  if (err) {
    console.error(err);
    return;
  }
  console.log(stdout);
});

// Script with spaces in the filename:
const bat = spawn('my script.cmd', ['a', 'b'], { shell: true });
// or:
exec('"my script.cmd" a b', (err, stdout, stderr) => {
  // ...
});

```

child_process.exec(command[, options][, callback])

- **command** `<string>` The command to run, with space-separated arguments.
- **options** `<Object>`
 - **cwd** `<string> | <URL>` Current working directory of the child process. **Default:** `process.cwd()`.
 - **env** `<Object>` Environment key-value pairs. **Default:** `process.env`.
 - **encoding** `<string>` **Default:** `'utf8'`
 - **shell** `<string>` Shell to execute the command with. See [Shell requirements](#) and [Default Windows shell](#). **Default:** `'/bin/sh'` on Unix, `process.env.ComSpec` on Windows.
 - **signal** `<AbortSignal>` allows aborting the child process using an `AbortSignal`.
 - **timeout** `<number>` **Default:** `0`
 - **maxBuffer** `<number>` Largest amount of data in bytes allowed on stdout or stderr. If exceeded, the child process is terminated and any output is truncated. See caveat at [maxBuffer and Unicode](#). **Default:** `1024 * 1024`.
 - **killSignal** `<string> | <integer>` **Default:** `'SIGTERM'`
 - **uid** `<number>` Sets the user identity of the process (see `setuid(2)`).
 - **gid** `<number>` Sets the group identity of the process (see `setgid(2)`).
 - **windowsHide** `<boolean>` Hide the subprocess console window that would normally be created on Windows systems. **Default:** `false`.
- **callback** `<Function>` called with the output when process terminates.
 - **error** `<Error>`
 - **stdout** `<string> | <Buffer>`
 - **stderr** `<string> | <Buffer>`

- Returns: `<ChildProcess>`

Spawns a shell then executes the `command` within that shell, buffering any generated output. The `command` string passed to the `exec` function is processed directly by the shell and special characters (vary based on `shell`) need to be dealt with accordingly:

```
const { exec } = require('child_process');

exec('"/path/to/test file/test.sh" arg1 arg2');
// Double quotes are used so that the space in the path is not interpreted as
// a delimiter of multiple arguments.

exec('echo "The \\$HOME variable is $HOME"');
// The $HOME variable is escaped in the first instance, but not in the second.
```

Never pass unsanitized user input to this function. Any input containing shell metacharacters may be used to trigger arbitrary command execution.

If a `callback` function is provided, it is called with the arguments `(error, stdout, stderr)`. On success, `error` will be `null`. On error, `error` will be an instance of `Error`. The `error.code` property will be the exit code of the process. By convention, any exit code other than `0` indicates an error. `error.signal` will be the signal that terminated the process.

The `stdout` and `stderr` arguments passed to the callback will contain the stdout and stderr output of the child process. By default, Node.js will decode the output as UTF-8 and pass strings to the callback. The `encoding` option can be used to specify the character encoding used to decode the stdout and stderr output. If `encoding` is `'buffer'`, or an unrecognized character encoding, `Buffer` objects will be passed to the callback instead.

```
const { exec } = require('child_process');
exec('cat *.js missing_file | wc -l', (error, stdout, stderr) => {
  if (error) {
    console.error(`exec error: ${error}`);
    return;
  }
  console.log(`stdout: ${stdout}`);
  console.error(`stderr: ${stderr}`);
});
```

If `timeout` is greater than `0`, the parent will send the signal identified by the `killSignal` property (the default is `'SIGTERM'`) if the child runs longer than `timeout` milliseconds.

Unlike the `exec(3)` POSIX system call, `child_process.exec()` does not replace the existing process and uses a shell to execute the command.

If this method is invoked as its `util.promisify()` ed version, it returns a `Promise` for an `Object` with `stdout` and `stderr` properties. The returned `ChildProcess` instance is attached to the `Promise` as a `child` property. In case of an error (including any error resulting in an exit code other than `0`), a rejected promise is returned, with the same `error` object given in the callback, but with two additional properties `stdout` and `stderr`.

```
const util = require('util');
const exec = util.promisify(require('child_process').exec);

async function lsExample() {
  const { stdout, stderr } = await exec('ls');
```

```

console.log('stdout:', stdout);
console.error('stderr:', stderr);
}
lsExample();

```

If the `signal` option is enabled, calling `.abort()` on the corresponding `AbortController` is similar to calling `.kill()` on the child process except the error passed to the callback will be an `AbortError` :

```

const { exec } = require('child_process');
const controller = new AbortController();
const { signal } = controller;
const child = exec('grep ssh', { signal }, (error) => {
  console.log(error); // an AbortError
});
controller.abort();

```

`child_process.execFile(file[, args][, options][, callback])`

- `file` `<string>` The name or path of the executable file to run.
- `args` `<string[]>` List of string arguments.
- `options` `<Object>`
 - `cwd` `<string>` | `<URL>` Current working directory of the child process.
 - `env` `<Object>` Environment key-value pairs. **Default:** `process.env`.
 - `encoding` `<string>` **Default:** `'utf8'`
 - `timeout` `<number>` **Default:** `0`
 - `maxBuffer` `<number>` Largest amount of data in bytes allowed on stdout or stderr. If exceeded, the child process is terminated and any output is truncated. See caveat at `maxBuffer and Unicode`. **Default:** `1024 * 1024`.
 - `killSignal` `<string>` | `<integer>` **Default:** `'SIGTERM'`
 - `uid` `<number>` Sets the user identity of the process (see `setuid(2)`).
 - `gid` `<number>` Sets the group identity of the process (see `setgid(2)`).
 - `windowsHide` `<boolean>` Hide the subprocess console window that would normally be created on Windows systems. **Default:** `false`.
 - `windowsVerbatimArguments` `<boolean>` No quoting or escaping of arguments is done on Windows. Ignored on Unix. **Default:** `false`.
 - `shell` `<boolean>` | `<string>` If `true`, runs `command` inside of a shell. Uses `'/bin/sh'` on Unix, and `process.env.ComSpec` on Windows. A different shell can be specified as a string. See `Shell requirements` and `Default Windows shell`. **Default:** `false` (no shell).
 - `signal` `<AbortSignal>` allows aborting the child process using an `AbortSignal`.
- `callback` `<Function>` Called with the output when process terminates.
 - `error` `<Error>`
 - `stdout` `<string>` | `<Buffer>`
 - `stderr` `<string>` | `<Buffer>`
- Returns: `<ChildProcess>`

The `child_process.execFile()` function is similar to `child_process.exec()` except that it does not spawn a shell by default. Rather, the specified executable `file` is spawned directly as a new process making it slightly more efficient than `child_process.exec()`.

The same options as `child_process.exec()` are supported. Since a shell is not spawned, behaviors such as I/O redirection and file globbing are not supported.

```
const { execFile } = require('child_process');
const child = execFile('node', ['--version'], (error, stdout, stderr) => {
  if (error) {
    throw error;
  }
  console.log(stdout);
});
```

The `stdout` and `stderr` arguments passed to the callback will contain the stdout and stderr output of the child process. By default, Node.js will decode the output as UTF-8 and pass strings to the callback. The `encoding` option can be used to specify the character encoding used to decode the stdout and stderr output. If `encoding` is `'buffer'`, or an unrecognized character encoding, `Buffer` objects will be passed to the callback instead.

If this method is invoked as its `util.promisify()` ed version, it returns a `Promise` for an `Object` with `stdout` and `stderr` properties. The returned `ChildProcess` instance is attached to the `Promise` as a `child` property. In case of an error (including any error resulting in an exit code other than 0), a rejected promise is returned, with the same `error` object given in the callback, but with two additional properties `stdout` and `stderr`.

```
const util = require('util');
const execFile = util.promisify(require('child_process').execFile);
async function getVersion() {
  const { stdout } = await execFile('node', ['--version']);
  console.log(stdout);
}
getVersion();
```

If the `shell` option is enabled, do not pass unsanitized user input to this function. Any input containing shell metacharacters may be used to trigger arbitrary command execution.

If the `signal` option is enabled, calling `.abort()` on the corresponding `AbortController` is similar to calling `.kill()` on the child process except the error passed to the callback will be an `AbortError`:

```
const { execFile } = require('child_process');
const controller = new AbortController();
const { signal } = controller;
const child = execFile('node', ['--version'], { signal }, (error) => {
  console.log(error); // an AbortError
});
controller.abort();
```

`child_process.fork(modulePath[, args][, options])`

- `modulePath` `<string>` The module to run in the child.
- `args` `<string[]>` List of string arguments.
- `options` `<Object>`
 - `cwd` `<string>` | `<URL>` Current working directory of the child process.

- `detached` `<boolean>` Prepare child to run independently of its parent process. Specific behavior depends on the platform, see `options.detached`).
- `env` `<Object>` Environment key-value pairs. **Default:** `process.env`.
- `execPath` `<string>` Executable used to create the child process.
- `execArgv` `<string[]>` List of string arguments passed to the executable. **Default:** `process.execArgv`.
- `gid` `<number>` Sets the group identity of the process (see `setgid(2)`).
- `serialization` `<string>` Specify the kind of serialization used for sending messages between processes. Possible values are `'json'` and `'advanced'`. See [Advanced serialization](#) for more details. **Default:** `'json'`.
- `signal` `<AbortSignal>` Allows closing the child process using an `AbortSignal`.
- `killSignal` `<string> | <integer>` The signal value to be used when the spawned process will be killed by timeout or abort signal. **Default:** `'SIGTERM'`.
- `silent` `<boolean>` If `true`, `stdin`, `stdout`, and `stderr` of the child will be piped to the parent, otherwise they will be inherited from the parent, see the `'pipe'` and `'inherit'` options for `child_process.spawn()`'s `stdio` for more details. **Default:** `false`.
- `stdio` `<Array> | <string>` See `child_process.spawn()`'s `stdio`. When this option is provided, it overrides `silent`. If the array variant is used, it must contain exactly one item with value `'ipc'` or an error will be thrown. For instance `[0, 1, 2, 'ipc']`.
- `uid` `<number>` Sets the user identity of the process (see `setuid(2)`).
- `windowsVerbatimArguments` `<boolean>` No quoting or escaping of arguments is done on Windows. Ignored on Unix. **Default:** `false`.
- `timeout` `<number>` In milliseconds the maximum amount of time the process is allowed to run. **Default:** `undefined`.
- Returns: `<ChildProcess>`

The `child_process.fork()` method is a special case of `child_process.spawn()` used specifically to spawn new Node.js processes. Like `child_process.spawn()`, a `ChildProcess` object is returned. The returned `ChildProcess` will have an additional communication channel built-in that allows messages to be passed back and forth between the parent and child. See `subprocess.send()` for details.

Keep in mind that spawned Node.js child processes are independent of the parent with exception of the IPC communication channel that is established between the two. Each process has its own memory, with their own V8 instances. Because of the additional resource allocations required, spawning a large number of child Node.js processes is not recommended.

By default, `child_process.fork()` will spawn new Node.js instances using the `process.execPath` of the parent process. The `execPath` property in the `options` object allows for an alternative execution path to be used.

Node.js processes launched with a custom `execPath` will communicate with the parent process using the file descriptor (fd) identified using the environment variable `NODE_CHANNEL_FD` on the child process.

Unlike the `fork(2)` POSIX system call, `child_process.fork()` does not clone the current process.

The `shell` option available in `child_process.spawn()` is not supported by `child_process.fork()` and will be ignored if set.

If the `signal` option is enabled, calling `.abort()` on the corresponding `AbortController` is similar to calling `.kill()` on the child process except the error passed to the callback will be an `AbortError`:

```
if (process.argv[2] === 'child') {
  setTimeout(() => {
    console.log(`Hello from ${process.argv[2]}!`);
  }, 1_000);
} else {
  const { fork } = require('child_process');
  const controller = new AbortController();
  const { signal } = controller;
```



```
const child = fork(__filename, ['child'], { signal });
child.on('error', (err) => {
  // This will be called with err being an AbortError if the controller aborts
});
controller.abort(); // Stops the child process
}
```

child_process.spawn(command[, args][, options])

- **command** `<string>` The command to run.
- **args** `<string[]>` List of string arguments.
- **options** `<Object>`
 - **cwd** `<string>` | `<URL>` Current working directory of the child process.
 - **env** `<Object>` Environment key-value pairs. **Default:** `process.env`.
 - **argv0** `<string>` Explicitly set the value of `argv[0]` sent to the child process. This will be set to `command` if not specified.
 - **stdio** `<Array>` | `<string>` Child's stdio configuration (see `options.stdio`).
 - **detached** `<boolean>` Prepare child to run independently of its parent process. Specific behavior depends on the platform, see `options.detached`.
 - **uid** `<number>` Sets the user identity of the process (see `setuid(2)`).
 - **gid** `<number>` Sets the group identity of the process (see `setgid(2)`).
 - **serialization** `<string>` Specify the kind of serialization used for sending messages between processes. Possible values are `'json'` and `'advanced'`. See [Advanced serialization](#) for more details. **Default:** `'json'`.
 - **shell** `<boolean>` | `<string>` If `true`, runs `command` inside of a shell. Uses `'/bin/sh'` on Unix, and `process.env.ComSpec` on Windows. A different shell can be specified as a string. See [Shell requirements](#) and [Default Windows shell](#). **Default:** `false` (no shell).
 - **windowsVerbatimArguments** `<boolean>` No quoting or escaping of arguments is done on Windows. Ignored on Unix. This is set to `true` automatically when `shell` is specified and is CMD. **Default:** `false`.
 - **windowsHide** `<boolean>` Hide the subprocess console window that would normally be created on Windows systems. **Default:** `false`.
 - **signal** `<AbortSignal>` allows aborting the child process using an `AbortSignal`.
 - **timeout** `<number>` In milliseconds the maximum amount of time the process is allowed to run. **Default:** `undefined`.
 - **killSignal** `<string>` | `<integer>` The signal value to be used when the spawned process will be killed by timeout or abort signal. **Default:** `'SIGTERM'`.
- Returns: `<ChildProcess>`

The `child_process.spawn()` method spawns a new process using the given `command`, with command-line arguments in `args`. If omitted, `args` defaults to an empty array.

If the `shell` option is enabled, do not pass unsanitized user input to this function. Any input containing shell metacharacters may be used to trigger arbitrary command execution.

A third argument may be used to specify additional options, with these defaults:

```
const defaults = {
  cwd: undefined,
```

```
env: process.env  
};
```

Use `cwd` to specify the working directory from which the process is spawned. If not given, the default is to inherit the current working directory. If given, but the path does not exist, the child process emits an `ENOENT` error and exits immediately. `ENOENT` is also emitted when the command does not exist.

Use `env` to specify environment variables that will be visible to the new process, the default is `process.env`.

undefined values in `env` will be ignored.

Example of running `ls -lh /usr`, capturing `stdout`, `stderr`, and the exit code:

```
const { spawn } = require('child_process');  
const ls = spawn('ls', ['-lh', '/usr']);  
  
ls.stdout.on('data', (data) => {  
  console.log(`stdout: ${data}`);  
});  
  
ls.stderr.on('data', (data) => {  
  console.error(`stderr: ${data}`);  
});  
  
ls.on('close', (code) => {  
  console.log(`child process exited with code ${code}`);  
});
```

Example: A very elaborate way to run `ps ax | grep ssh`

```
const { spawn } = require('child_process');  
const ps = spawn('ps', ['ax']);  
const grep = spawn('grep', ['ssh']);  
  
ps.stdout.on('data', (data) => {  
  grep.stdin.write(data);  
});  
  
ps.stderr.on('data', (data) => {  
  console.error(`ps stderr: ${data}`);  
});  
  
ps.on('close', (code) => {  
  if (code !== 0) {  
    console.log(`ps process exited with code ${code}`);  
  }  
  grep.stdin.end();  
});  
  
grep.stdout.on('data', (data) => {  
  console.log(data.toString());  
});
```

```
});

grep.stderr.on('data', (data) => {
  console.error(`grep stderr: ${data}`);
});

grep.on('close', (code) => {
  if (code !== 0) {
    console.log(`grep process exited with code ${code}`);
  }
});
```

Example of checking for failed `spawn` :

```
const { spawn } = require('child_process');
const subprocess = spawn('bad_command');

subprocess.on('error', (err) => {
  console.error('Failed to start subprocess.');
```

Certain platforms (macOS, Linux) will use the value of `argv[0]` for the process title while others (Windows, SunOS) will use `command` .

Node.js currently overwrites `argv[0]` with `process.execPath` on startup, so `process.argv[0]` in a Node.js child process will not match the `argv0` parameter passed to `spawn` from the parent, retrieve it with the `process.argv0` property instead.

If the `signal` option is enabled, calling `.abort()` on the corresponding `AbortController` is similar to calling `.kill()` on the child process except the error passed to the callback will be an `AbortError` :

```
const { spawn } = require('child_process');
const controller = new AbortController();
const { signal } = controller;
const grep = spawn('grep', ['ssh'], { signal });
grep.on('error', (err) => {
  // This will be called with err being an AbortError if the controller aborts
});
controller.abort(); // Stops the child process
```

options.detached

On Windows, setting `options.detached` to `true` makes it possible for the child process to continue running after the parent exits. The child will have its own console window. Once enabled for a child process, it cannot be disabled.

On non-Windows platforms, if `options.detached` is set to `true` , the child process will be made the leader of a new process group and session. Child processes may continue running after the parent exits regardless of whether they are detached or not. See [setsid\(2\)](#) for more information.

By default, the parent will wait for the detached child to exit. To prevent the parent from waiting for a given `subprocess` to exit, use the `subprocess.unref()` method. Doing so will cause the parent's event loop to not include the child in its reference count, allowing the parent to exit independently of the child, unless there is an established IPC channel between the child and the parent.

When using the `detached` option to start a long-running process, the process will not stay running in the background after the parent exits unless it is provided with a `stdio` configuration that is not connected to the parent. If the parent's `stdio` is inherited, the child will remain attached to the controlling terminal.

Example of a long-running process, by detaching and also ignoring its parent `stdio` file descriptors, in order to ignore the parent's termination:

```
const { spawn } = require('child_process');

const subprocess = spawn(process.argv[0], ['child_program.js'], {
  detached: true,
  stdio: 'ignore'
});

subprocess.unref();
```

Alternatively one can redirect the child process' output into files:

```
const fs = require('fs');
const { spawn } = require('child_process');
const out = fs.openSync('./out.log', 'a');
const err = fs.openSync('./out.log', 'a');

const subprocess = spawn('prg', [], {
  detached: true,
  stdio: [ 'ignore', out, err ]
});

subprocess.unref();
```

options.stdio

The `options.stdio` option is used to configure the pipes that are established between the parent and child process. By default, the child's `stdin`, `stdout`, and `stderr` are redirected to corresponding `subprocess.stdin`, `subprocess.stdout`, and `subprocess.stderr` streams on the `ChildProcess` object. This is equivalent to setting the `options.stdio` equal to `['pipe', 'pipe', 'pipe']`.

For convenience, `options.stdio` may be one of the following strings:

- `'pipe'`: equivalent to `['pipe', 'pipe', 'pipe']` (the default)
- `'overlapped'`: equivalent to `['overlapped', 'overlapped', 'overlapped']`
- `'ignore'`: equivalent to `['ignore', 'ignore', 'ignore']`
- `'inherit'`: equivalent to `['inherit', 'inherit', 'inherit']` or `[0, 1, 2]`

Otherwise, the value of `options.stdio` is an array where each index corresponds to an fd in the child. The fds 0, 1, and 2 correspond to `stdin`, `stdout`, and `stderr`, respectively. Additional fds can be specified to create additional pipes between the parent and child. The value is one of the following:

1. `'pipe'`: Create a pipe between the child process and the parent process. The parent end of the pipe is exposed to the parent as a property on the `child_process` object as `subprocess.stdio[fd]`. Pipes created for fds 0, 1, and 2 are also available as `subprocess.stdin`, `subprocess.stdout` and `subprocess.stderr`, respectively.

2. `'overlapped'` : Same as `'pipe'` except that the `FILE_FLAG_OVERLAPPED` flag is set on the handle. This is necessary for overlapped I/O on the child process's stdio handles. See the [docs](#) for more details. This is exactly the same as `'pipe'` on non-Windows systems.
3. `'ipc'` : Create an IPC channel for passing messages/file descriptors between parent and child. A `ChildProcess` may have at most one IPC stdio file descriptor. Setting this option enables the `subprocess.send()` method. If the child is a Node.js process, the presence of an IPC channel will enable `process.send()` and `process.disconnect()` methods, as well as `'disconnect'` and `'message'` events within the child.

Accessing the IPC channel fd in any way other than `process.send()` or using the IPC channel with a child process that is not a Node.js instance is not supported.
4. `'ignore'` : Instructs Node.js to ignore the fd in the child. While Node.js will always open fds 0, 1, and 2 for the processes it spawns, setting the fd to `'ignore'` will cause Node.js to open `/dev/null` and attach it to the child's fd.
5. `'inherit'` : Pass through the corresponding stdio stream to/from the parent process. In the first three positions, this is equivalent to `process.stdin`, `process.stdout`, and `process.stderr`, respectively. In any other position, equivalent to `'ignore'`.
6. `<Stream>` object: Share a readable or writable stream that refers to a tty, file, socket, or a pipe with the child process. The stream's underlying file descriptor is duplicated in the child process to the fd that corresponds to the index in the `stdio` array. The stream must have an underlying descriptor (file streams do not until the `'open'` event has occurred).
7. Positive integer: The integer value is interpreted as a file descriptor that is currently open in the parent process. It is shared with the child process, similar to how `<Stream>` objects can be shared. Passing sockets is not supported on Windows.
8. `null`, `undefined` : Use default value. For stdio fds 0, 1, and 2 (in other words, stdin, stdout, and stderr) a pipe is created. For fd 3 and up, the default is `'ignore'`.

```
const { spawn } = require('child_process');

// Child will use parent's stdios.
spawn('prg', [], { stdio: 'inherit' });

// Spawn child sharing only stderr.
spawn('prg', [], { stdio: ['pipe', 'pipe', process.stderr] });

// Open an extra fd=4, to interact with programs presenting a
// startd-style interface.
spawn('prg', [], { stdio: ['pipe', null, null, null, 'pipe' ] });
```

It is worth noting that when an IPC channel is established between the parent and child processes, and the child is a Node.js process, the child is launched with the IPC channel unreferenced (using `unref()`) until the child registers an event handler for the `'disconnect'` event or the `'message'` event. This allows the child to exit normally without the process being held open by the open IPC channel.

On Unix-like operating systems, the `child_process.spawn()` method performs memory operations synchronously before decoupling the event loop from the child. Applications with a large memory footprint may find frequent `child_process.spawn()` calls to be a bottleneck. For more information, see [V8 issue 7381](#).

See also: `child_process.exec()` and `child_process.fork()`.

Synchronous process creation

The `child_process.spawnSync()`, `child_process.execSync()`, and `child_process.execFileSync()` methods are synchronous and will block the Node.js event loop, pausing execution of any additional code until the spawned process exits.

Blocking calls like these are mostly useful for simplifying general-purpose scripting tasks and for simplifying the loading/processing of application configuration at startup.

`child_process.execFileSync(file[, args][, options])`

- `file` `<string>` The name or path of the executable file to run.
- `args` `<string[]>` List of string arguments.
- `options` `<Object>`
 - `cwd` `<string>` | `<URL>` Current working directory of the child process.
 - `input` `<string>` | `<Buffer>` | `<TypedArray>` | `<DataView>` The value which will be passed as stdin to the spawned process. Supplying this value will override `stdio[0]`.
 - `stdio` `<string>` | `<Array>` Child's stdio configuration. `stderr` by default will be output to the parent process' stderr unless `stdio` is specified. **Default:** `'pipe'`.
 - `env` `<Object>` Environment key-value pairs. **Default:** `process.env`.
 - `uid` `<number>` Sets the user identity of the process (see `setuid(2)`).
 - `gid` `<number>` Sets the group identity of the process (see `setgid(2)`).
 - `timeout` `<number>` In milliseconds the maximum amount of time the process is allowed to run. **Default:** `undefined`.
 - `killSignal` `<string>` | `<integer>` The signal value to be used when the spawned process will be killed. **Default:** `'SIGTERM'`.
 - `maxBuffer` `<number>` Largest amount of data in bytes allowed on stdout or stderr. If exceeded, the child process is terminated. See caveat at `maxBuffer and Unicode`. **Default:** `1024 * 1024`.
 - `encoding` `<string>` The encoding used for all stdio inputs and outputs. **Default:** `'buffer'`.
 - `windowsHide` `<boolean>` Hide the subprocess console window that would normally be created on Windows systems. **Default:** `false`.
 - `shell` `<boolean>` | `<string>` If `true`, runs `command` inside of a shell. Uses `'/bin/sh'` on Unix, and `process.env.ComSpec` on Windows. A different shell can be specified as a string. See `Shell requirements` and `Default Windows shell`. **Default:** `false` (no shell).
- Returns: `<Buffer>` | `<string>` The stdout from the command.

The `child_process.execFileSync()` method is generally identical to `child_process.execFile()` with the exception that the method will not return until the child process has fully closed. When a timeout has been encountered and `killSignal` is sent, the method won't return until the process has completely exited.

If the child process intercepts and handles the `SIGTERM` signal and does not exit, the parent process will still wait until the child process has exited.

If the process times out or has a non-zero exit code, this method will throw an `Error` that will include the full result of the underlying `child_process.spawnSync()`.

If the `shell` option is enabled, do not pass unsanitized user input to this function. Any input containing shell metacharacters may be used to trigger arbitrary command execution.

`child_process.execSync(command[, options])`

- `command` `<string>` The command to run.
- `options` `<Object>`
 - `cwd` `<string>` | `<URL>` Current working directory of the child process.
 - `input` `<string>` | `<Buffer>` | `<TypedArray>` | `<DataView>` The value which will be passed as stdin to the spawned process. Supplying this value will override `stdio[0]`.
 - `stdio` `<string>` | `<Array>` Child's stdio configuration. `stderr` by default will be output to the parent process' stderr unless `stdio` is specified. **Default:** `'pipe'`.

- `env` `<Object>` Environment key-value pairs. **Default:** `process.env`.
 - `shell` `<string>` Shell to execute the command with. See [Shell requirements](#) and [Default Windows shell](#). **Default:** `'/bin/sh'` on Unix, `process.env.ComSpec` on Windows.
 - `uid` `<number>` Sets the user identity of the process. (See `setuid(2)`).
 - `gid` `<number>` Sets the group identity of the process. (See `setgid(2)`).
 - `timeout` `<number>` In milliseconds the maximum amount of time the process is allowed to run. **Default:** `undefined`.
 - `killSignal` `<string> | <integer>` The signal value to be used when the spawned process will be killed. **Default:** `'SIGTERM'`.
 - `maxBuffer` `<number>` Largest amount of data in bytes allowed on stdout or stderr. If exceeded, the child process is terminated and any output is truncated. See caveat at [maxBuffer and Unicode](#). **Default:** `1024 * 1024`.
 - `encoding` `<string>` The encoding used for all stdio inputs and outputs. **Default:** `'buffer'`.
 - `windowsHide` `<boolean>` Hide the subprocess console window that would normally be created on Windows systems. **Default:** `false`.
- Returns: `<Buffer> | <string>` The stdout from the command.

The `child_process.execSync()` method is generally identical to `child_process.exec()` with the exception that the method will not return until the child process has fully closed. When a timeout has been encountered and `killSignal` is sent, the method won't return until the process has completely exited. If the child process intercepts and handles the `SIGTERM` signal and doesn't exit, the parent process will wait until the child process has exited.

If the process times out or has a non-zero exit code, this method will throw. The `Error` object will contain the entire result from `child_process.spawnSync()`.

Never pass unsanitized user input to this function. Any input containing shell metacharacters may be used to trigger arbitrary command execution.

`child_process.spawnSync(command[, args][, options])`

- `command` `<string>` The command to run.
- `args` `<string[]>` List of string arguments.
- `options` `<Object>`
 - `cwd` `<string> | <URL>` Current working directory of the child process.
 - `input` `<string> | <Buffer> | <TypedArray> | <DataView>` The value which will be passed as stdin to the spawned process. Supplying this value will override `stdio[0]`.
 - `argv0` `<string>` Explicitly set the value of `argv[0]` sent to the child process. This will be set to `command` if not specified.
 - `stdio` `<string> | <Array>` Child's stdio configuration.
 - `env` `<Object>` Environment key-value pairs. **Default:** `process.env`.
 - `uid` `<number>` Sets the user identity of the process (see `setuid(2)`).
 - `gid` `<number>` Sets the group identity of the process (see `setgid(2)`).
 - `timeout` `<number>` In milliseconds the maximum amount of time the process is allowed to run. **Default:** `undefined`.
 - `killSignal` `<string> | <integer>` The signal value to be used when the spawned process will be killed. **Default:** `'SIGTERM'`.
 - `maxBuffer` `<number>` Largest amount of data in bytes allowed on stdout or stderr. If exceeded, the child process is terminated and any output is truncated. See caveat at [maxBuffer and Unicode](#). **Default:** `1024 * 1024`.
 - `encoding` `<string>` The encoding used for all stdio inputs and outputs. **Default:** `'buffer'`.
 - `shell` `<boolean> | <string>` If `true`, runs `command` inside of a shell. Uses `'/bin/sh'` on Unix, and `process.env.ComSpec` on Windows. A different shell can be specified as a string. See [Shell requirements](#) and [Default Windows shell](#). **Default:** `false` (no shell).
 - `windowsVerbatimArguments` `<boolean>` No quoting or escaping of arguments is done on Windows. Ignored on Unix. This is set to `true` automatically when `shell` is specified and is CMD. **Default:** `false`.

- `windowsHide` `<boolean>` Hide the subprocess console window that would normally be created on Windows systems. **Default:** `false`.
- Returns: `<Object>`
 - `pid` `<number>` Pid of the child process.
 - `output` `<Array>` Array of results from stdio output.
 - `stdout` `<Buffer>` | `<string>` The contents of `output[1]`.
 - `stderr` `<Buffer>` | `<string>` The contents of `output[2]`.
 - `status` `<number>` | `<null>` The exit code of the subprocess, or `null` if the subprocess terminated due to a signal.
 - `signal` `<string>` | `<null>` The signal used to kill the subprocess, or `null` if the subprocess did not terminate due to a signal.
 - `error` `<Error>` The error object if the child process failed or timed out.

The `child_process.spawnSync()` method is generally identical to `child_process.spawn()` with the exception that the function will not return until the child process has fully closed. When a timeout has been encountered and `killSignal` is sent, the method won't return until the process has completely exited. If the process intercepts and handles the `SIGTERM` signal and doesn't exit, the parent process will wait until the child process has exited.

If the `shell` option is enabled, do not pass unsanitized user input to this function. Any input containing shell metacharacters may be used to trigger arbitrary command execution.

Class: `ChildProcess`

- Extends: `<EventEmitter>`

Instances of the `ChildProcess` represent spawned child processes.

Instances of `ChildProcess` are not intended to be created directly. Rather, use the `child_process.spawn()`, `child_process.exec()`, `child_process.execFile()`, or `child_process.fork()` methods to create instances of `ChildProcess`.

Event: `'close'`

- `code` `<number>` The exit code if the child exited on its own.
- `signal` `<string>` The signal by which the child process was terminated.

The `'close'` event is emitted after a process has ended *and* the stdio streams of a child process have been closed. This is distinct from the `'exit'` event, since multiple processes might share the same stdio streams. The `'close'` event will always emit after `'exit'` was already emitted, or `'error'` if the child failed to spawn.

```
const { spawn } = require('child_process');
const ls = spawn('ls', ['-lh', '/usr']);

ls.stdout.on('data', (data) => {
  console.log(`stdout: ${data}`);
});

ls.on('close', (code) => {
  console.log(`child process close all stdio with code ${code}`);
});

ls.on('exit', (code) => {
  console.log(`child process exited with code ${code}`);
});
```


Event: 'disconnect'

The `'disconnect'` event is emitted after calling the `subprocess.disconnect()` method in parent process or `process.disconnect()` in child process. After disconnecting it is no longer possible to send or receive messages, and the `subprocess.connected` property is `false`.

Event: 'error'

- `err` `<Error>` The error.

The `'error'` event is emitted whenever:

1. The process could not be spawned, or
2. The process could not be killed, or
3. Sending a message to the child process failed.

The `'exit'` event may or may not fire after an error has occurred. When listening to both the `'exit'` and `'error'` events, guard against accidentally invoking handler functions multiple times.

See also `subprocess.kill()` and `subprocess.send()`.

Event: 'exit'

- `code` `<number>` The exit code if the child exited on its own.
- `signal` `<string>` The signal by which the child process was terminated.

The `'exit'` event is emitted after the child process ends. If the process exited, `code` is the final exit code of the process, otherwise `null`. If the process terminated due to receipt of a signal, `signal` is the string name of the signal, otherwise `null`. One of the two will always be non-`null`.

When the `'exit'` event is triggered, child process stdio streams might still be open.

Node.js establishes signal handlers for `SIGINT` and `SIGTERM` and Node.js processes will not terminate immediately due to receipt of those signals. Rather, Node.js will perform a sequence of cleanup actions and then will re-raise the handled signal.

See `waitpid(2)`.

Event: 'message'

- `message` `<Object>` A parsed JSON object or primitive value.
- `sendHandle` `<Handle>` A `net.Socket` or `net.Server` object, or undefined.

The `'message'` event is triggered when a child process uses `process.send()` to send messages.

The message goes through serialization and parsing. The resulting message might not be the same as what is originally sent.

If the `serialization` option was set to `'advanced'` used when spawning the child process, the `message` argument can contain data that JSON is not able to represent. See [Advanced serialization](#) for more details.

Event: 'spawn'

The `'spawn'` event is emitted once the child process has spawned successfully. If the child process does not spawn successfully, the `'spawn'` event is not emitted and the `'error'` event is emitted instead.

If emitted, the `'spawn'` event comes before all other events and before any data is received via `stdout` or `stderr`.

The `'spawn'` event will fire regardless of whether an error occurs **within** the spawned process. For example, if `bash some-command` spawns successfully, the `'spawn'` event will fire, though `bash` may fail to spawn `some-command`. This caveat also applies when using `{ shell: true }`.

subprocess.channel

- `<Object>` A pipe representing the IPC channel to the child process.

The `subprocess.channel` property is a reference to the child's IPC channel. If no IPC channel currently exists, this property is `undefined`.

subprocess.channel.ref()

This method makes the IPC channel keep the event loop of the parent process running if `.unref()` has been called before.

subprocess.channel.unref()

This method makes the IPC channel not keep the event loop of the parent process running, and lets it finish even while the channel is open.

subprocess.connected

- `<boolean>` Set to `false` after `subprocess.disconnect()` is called.

The `subprocess.connected` property indicates whether it is still possible to send and receive messages from a child process. When `subprocess.connected` is `false`, it is no longer possible to send or receive messages.

subprocess.disconnect()

Closes the IPC channel between parent and child, allowing the child to exit gracefully once there are no other connections keeping it alive. After calling this method the `subprocess.connected` and `process.connected` properties in both the parent and child (respectively) will be set to `false`, and it will be no longer possible to pass messages between the processes.

The `'disconnect'` event will be emitted when there are no messages in the process of being received. This will most often be triggered immediately after calling `subprocess.disconnect()`.

When the child process is a Node.js instance (e.g. spawned using `child_process.fork()`), the `process.disconnect()` method can be invoked within the child process to close the IPC channel as well.

subprocess.exitCode

- `<integer>`

The `subprocess.exitCode` property indicates the exit code of the child process. If the child process is still running, the field will be `null`.

subprocess.kill([signal])

- `signal` `<number>` | `<string>`
- Returns: `<boolean>`

The `subprocess.kill()` method sends a signal to the child process. If no argument is given, the process will be sent the `'SIGTERM'` signal. See `signal(7)` for a list of available signals. This function returns `true` if `kill(2)` succeeds, and `false` otherwise.

```
const { spawn } = require('child_process');
const grep = spawn('grep', ['ssh']);

grep.on('close', (code, signal) => {
  console.log(
    `child process terminated due to receipt of signal ${signal}`);
});

// Send SIGHUP to process.
grep.kill('SIGHUP');
```

The `ChildProcess` object may emit an `'error'` event if the signal cannot be delivered. Sending a signal to a child process that has already exited is not an error but may have unforeseen consequences. Specifically, if the process identifier (PID) has been reassigned to another process, the signal will be delivered to that process instead which can have unexpected results.

While the function is called `kill`, the signal delivered to the child process may not actually terminate the process.

See `kill(2)` for reference.

On Windows, where POSIX signals do not exist, the `signal` argument will be ignored, and the process will be killed forcefully and abruptly (similar to `'SIGKILL'`). See [Signal Events](#) for more details.

On Linux, child processes of child processes will not be terminated when attempting to kill their parent. This is likely to happen when running a new process in a shell or with the use of the `shell` option of `ChildProcess`:

```
'use strict';
const { spawn } = require('child_process');

const subprocess = spawn(
  'sh',
  [
    '-c',
    `node -e "setInterval(() => {
      console.log(process.pid, 'is alive')
    }, 500);"`,
  ], {
    stdio: ['inherit', 'inherit', 'inherit']
  }
);

setTimeout(() => {
  subprocess.kill(); // Does not terminate the Node.js process in the shell.
}, 2000);
```

subprocess.killed

- `<boolean>` Set to `true` after `subprocess.kill()` is used to successfully send a signal to the child process.

The `subprocess.killed` property indicates whether the child process successfully received a signal from `subprocess.kill()`. The `killed` property does not indicate that the child process has been terminated.

subprocess.pid

- `<integer> | <undefined>`

Returns the process identifier (PID) of the child process. If the child process fails to spawn due to errors, then the value is `undefined` and `error` is emitted.

```
const { spawn } = require('child_process');
const grep = spawn('grep', ['ssh']);

console.log(`Spawned child pid: ${grep.pid}`);
grep.stdin.end();
```

subprocess.ref()

Calling `subprocess.ref()` after making a call to `subprocess.unref()` will restore the removed reference count for the child process, forcing the parent to wait for the child to exit before exiting itself.

```
const { spawn } = require('child_process');

const subprocess = spawn(process.argv[0], ['child_program.js'], {
  detached: true,
  stdio: 'ignore'
});

subprocess.unref();
subprocess.ref();
```

subprocess.send(message[, sendHandle[, options]][, callback])

- `message` `<Object>`
- `sendHandle` `<Handle>`
- `options` `<Object>` The `options` argument, if present, is an object used to parameterize the sending of certain types of handles. `options` supports the following properties:
 - `keepOpen` `<boolean>` A value that can be used when passing instances of `net.Socket`. When `true`, the socket is kept open in the sending process. **Default:** `false`.
- `callback` `<Function>`
- Returns: `<boolean>`

When an IPC channel has been established between the parent and child (i.e. when using `child_process.fork()`), the `subprocess.send()` method can be used to send messages to the child process. When the child process is a Node.js instance, these messages can be received via the `'message'` event.

The message goes through serialization and parsing. The resulting message might not be the same as what is originally sent.

For example, in the parent script:

```
const cp = require('child_process');
const n = cp.fork(`${__dirname}/sub.js`);

n.on('message', (m) => {
  console.log('PARENT got message:', m);
});

// Causes the child to print: CHILD got message: { hello: 'world' }
n.send({ hello: 'world' });
```

And then the child script, `'sub.js'` might look like this:

```
process.on('message', (m) => {
  console.log('CHILD got message:', m);
});
```

```
// Causes the parent to print: PARENT got message: { foo: 'bar', baz: null }
process.send({ foo: 'bar', baz: NaN });
```

Child Node.js processes will have a `process.send()` method of their own that allows the child to send messages back to the parent.

There is a special case when sending a `{cmd: 'NODE_foo'}` message. Messages containing a `NODE_` prefix in the `cmd` property are reserved for use within Node.js core and will not be emitted in the child's `'message'` event. Rather, such messages are emitted using the `'internalMessage'` event and are consumed internally by Node.js. Applications should avoid using such messages or listening for `'internalMessage'` events as it is subject to change without notice.

The optional `sendHandle` argument that may be passed to `subprocess.send()` is for passing a TCP server or socket object to the child process. The child will receive the object as the second argument passed to the callback function registered on the `'message'` event. Any data that is received and buffered in the socket will not be sent to the child.

The optional `callback` is a function that is invoked after the message is sent but before the child may have received it. The function is called with a single argument: `null` on success, or an `Error` object on failure.

If no `callback` function is provided and the message cannot be sent, an `'error'` event will be emitted by the `ChildProcess` object. This can happen, for instance, when the child process has already exited.

`subprocess.send()` will return `false` if the channel has closed or when the backlog of unsent messages exceeds a threshold that makes it unwise to send more. Otherwise, the method returns `true`. The `callback` function can be used to implement flow control.

Example: sending a server object

The `sendHandle` argument can be used, for instance, to pass the handle of a TCP server object to the child process as illustrated in the example below:

```
const subprocess = require('child_process').fork('subprocess.js');

// Open up the server object and send the handle.
const server = require('net').createServer();
server.on('connection', (socket) => {
  socket.end('handled by parent');
});
server.listen(1337, () => {
  subprocess.send('server', server);
});
```

The child would then receive the server object as:

```
process.on('message', (m, server) => {
  if (m === 'server') {
    server.on('connection', (socket) => {
      socket.end('handled by child');
    });
  }
});
```

Once the server is now shared between the parent and child, some connections can be handled by the parent and some by the child.

While the example above uses a server created using the `net` module, `dgram` module servers use exactly the same workflow with the exceptions of listening on a `'message'` event instead of `'connection'` and using `server.bind()` instead of `server.listen()`. This is, however, currently only supported on Unix platforms.

Example: sending a socket object

Similarly, the `sendHandler` argument can be used to pass the handle of a socket to the child process. The example below spawns two children that each handle connections with "normal" or "special" priority:

```
const { fork } = require('child_process');
const normal = fork('subprocess.js', ['normal']);
const special = fork('subprocess.js', ['special']);

// Open up the server and send sockets to child. Use pauseOnConnect to prevent
// the sockets from being read before they are sent to the child process.
const server = require('net').createServer({ pauseOnConnect: true });
server.on('connection', (socket) => {

  // If this is special priority...
  if (socket.remoteAddress === '74.125.127.100') {
    special.send('socket', socket);
    return;
  }
  // This is normal priority.
  normal.send('socket', socket);
});
server.listen(1337);
```

The `subprocess.js` would receive the socket handle as the second argument passed to the event callback function:

```
process.on('message', (m, socket) => {
  if (m === 'socket') {
    if (socket) {
      // Check that the client socket exists.
      // It is possible for the socket to be closed between the time it is
      // sent and the time it is received in the child process.
      socket.end(`Request handled with ${process.argv[2]} priority`);
    }
  }
});
```

Do not use `.maxConnections` on a socket that has been passed to a subprocess. The parent cannot track when the socket is destroyed.

Any `'message'` handlers in the subprocess should verify that `socket` exists, as the connection may have been closed during the time it takes to send the connection to the child.

subprocess.signalCode

- `<string> | <null>`

The `subprocess.signalCode` property indicates the signal received by the child process if any, else `null`.

subprocess.spawnargs

- `<Array>`

The `subprocess.spawnargs` property represents the full list of command-line arguments the child process was launched with.

subprocess.spawnfile

- `<string>`

The `subprocess.spawnfile` property indicates the executable file name of the child process that is launched.

For `child_process.fork()`, its value will be equal to `process.execPath`. For `child_process.spawn()`, its value will be the name of the executable file. For `child_process.exec()`, its value will be the name of the shell in which the child process is launched.

subprocess.stderr

- `<stream.Readable>`

A `Readable Stream` that represents the child process's `stderr`.

If the child was spawned with `stdio[2]` set to anything other than `'pipe'`, then this will be `null`.

`subprocess.stderr` is an alias for `subprocess.stdio[2]`. Both properties will refer to the same value.

The `subprocess.stderr` property can be `null` if the child process could not be successfully spawned.

subprocess.stdin

- `<stream.Writable>`

A `Writable Stream` that represents the child process's `stdin`.

If a child process waits to read all of its input, the child will not continue until this stream has been closed via `end()`.

If the child was spawned with `stdio[0]` set to anything other than `'pipe'`, then this will be `null`.

`subprocess.stdin` is an alias for `subprocess.stdio[0]`. Both properties will refer to the same value.

The `subprocess.stdin` property can be `undefined` if the child process could not be successfully spawned.

subprocess.stdio

- `<Array>`

A sparse array of pipes to the child process, corresponding with positions in the `stdio` option passed to `child_process.spawn()` that have been set to the value `'pipe'`. `subprocess.stdio[0]`, `subprocess.stdio[1]`, and `subprocess.stdio[2]` are also available as `subprocess.stdin`, `subprocess.stdout`, and `subprocess.stderr`, respectively.

In the following example, only the child's fd `1` (stdout) is configured as a pipe, so only the parent's `subprocess.stdio[1]` is a stream, all other values in the array are `null`.

```
const assert = require('assert');
const fs = require('fs');
const child_process = require('child_process');

const subprocess = child_process.spawn('ls', {
  stdio: [
    0, // Use parent's stdin for child.
```

```

    'pipe', // Pipe child's stdout to parent.
    fs.openSync('err.out', 'w'), // Direct child's stderr to a file.
  ]
});

assert.strictEqual(subprocess.stdio[0], null);
assert.strictEqual(subprocess.stdio[0], subprocess.stdin);

assert(subprocess.stdout);
assert.strictEqual(subprocess.stdio[1], subprocess.stdout);

assert.strictEqual(subprocess.stdio[2], null);
assert.strictEqual(subprocess.stdio[2], subprocess.stderr);

```

The `subprocess.stdio` property can be `undefined` if the child process could not be successfully spawned.

subprocess.stdout

- `<stream.Readable>`

A `Readable Stream` that represents the child process's `stdout`.

If the child was spawned with `stdio[1]` set to anything other than `'pipe'`, then this will be `null`.

`subprocess.stdout` is an alias for `subprocess.stdio[1]`. Both properties will refer to the same value.

```

const { spawn } = require('child_process');

const subprocess = spawn('ls');

subprocess.stdout.on('data', (data) => {
  console.log(`Received chunk ${data}`);
});

```

The `subprocess.stdout` property can be `null` if the child process could not be successfully spawned.

subprocess.unref()

By default, the parent will wait for the detached child to exit. To prevent the parent from waiting for a given `subprocess` to exit, use the `subprocess.unref()` method. Doing so will cause the parent's event loop to not include the child in its reference count, allowing the parent to exit independently of the child, unless there is an established IPC channel between the child and the parent.

```

const { spawn } = require('child_process');

const subprocess = spawn(process.argv[0], ['child_program.js'], {
  detached: true,
  stdio: 'ignore'
});

subprocess.unref();

```


maxBuffer and Unicode

The `maxBuffer` option specifies the largest number of bytes allowed on `stdout` or `stderr`. If this value is exceeded, then the child process is terminated. This impacts output that includes multibyte character encodings such as UTF-8 or UTF-16. For instance, `console.log('中文测试')` will send 13 UTF-8 encoded bytes to `stdout` although there are only 4 characters.

Shell requirements

The shell should understand the `-c` switch. If the shell is `'cmd.exe'`, it should understand the `/d /s /c` switches and command-line parsing should be compatible.

Default Windows shell

Although Microsoft specifies `%COMSPEC%` must contain the path to `'cmd.exe'` in the root environment, child processes are not always subject to the same requirement. Thus, in `child_process` functions where a shell can be spawned, `'cmd.exe'` is used as a fallback if `process.env.ComSpec` is unavailable.

Advanced serialization

Child processes support a serialization mechanism for IPC that is based on the [serialization API of the v8 module](#), based on the [HTML structured clone algorithm](#). This is generally more powerful and supports more built-in JavaScript object types, such as `BigInt`, `Map` and `Set`, `ArrayBuffer` and `TypedArray`, `Buffer`, `Error`, `RegExp` etc.

However, this format is not a full superset of JSON, and e.g. properties set on objects of such built-in types will not be passed on through the serialization step. Additionally, performance may not be equivalent to that of JSON, depending on the structure of the passed data. Therefore, this feature requires opting in by setting the `serialization` option to `'advanced'` when calling `child_process.spawn()` or `child_process.fork()`.

Cluster

Stability: 2 - Stable

Source Code: [lib/cluster.js](#)

A single instance of Node.js runs in a single thread. To take advantage of multi-core systems, the user will sometimes want to launch a cluster of Node.js processes to handle the load.

The cluster module allows easy creation of child processes that all share server ports.

```
import cluster from 'cluster';
import http from 'http';
import { cpus } from 'os';
import process from 'process';

const numCPUs = cpus().length;

if (cluster.isPrimary) {
  console.log(`Primary ${process.pid} is running`);
```

```

// Fork workers.
for (let i = 0; i < numCPUs; i++) {
  cluster.fork();
}

cluster.on('exit', (worker, code, signal) => {
  console.log(`worker ${worker.process.pid} died`);
});
} else {
  // Workers can share any TCP connection
  // In this case it is an HTTP server
  http.createServer((req, res) => {
    res.writeHead(200);
    res.end('hello world\n');
  }).listen(8000);

  console.log(`Worker ${process.pid} started`);
}const cluster = require('cluster');
const http = require('http');
const numCPUs = require('os').cpus().length;
const process = require('process');

if (cluster.isPrimary) {
  console.log(`Primary ${process.pid} is running`);

  // Fork workers.
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
  }

  cluster.on('exit', (worker, code, signal) => {
    console.log(`worker ${worker.process.pid} died`);
  });
} else {
  // Workers can share any TCP connection
  // In this case it is an HTTP server
  http.createServer((req, res) => {
    res.writeHead(200);
    res.end('hello world\n');
  }).listen(8000);

  console.log(`Worker ${process.pid} started`);
}

```

Running Node.js will now share port 8000 between the workers:

```

$ node server.js
Primary 3596 is running
Worker 4324 started
Worker 4520 started

```

```
Worker 6056 started
Worker 5644 started
```

On Windows, it is not yet possible to set up a named pipe server in a worker.

How it works

The worker processes are spawned using the `child_process.fork()` method, so that they can communicate with the parent via IPC and pass server handles back and forth.

The cluster module supports two methods of distributing incoming connections.

The first one (and the default one on all platforms except Windows), is the round-robin approach, where the primary process listens on a port, accepts new connections and distributes them across the workers in a round-robin fashion, with some built-in smarts to avoid overloading a worker process.

The second approach is where the primary process creates the listen socket and sends it to interested workers. The workers then accept incoming connections directly.

The second approach should, in theory, give the best performance. In practice however, distribution tends to be very unbalanced due to operating system scheduler vagaries. Loads have been observed where over 70% of all connections ended up in just two processes, out of a total of eight.

Because `server.listen()` hands off most of the work to the primary process, there are three cases where the behavior between a normal Node.js process and a cluster worker differs:

1. `server.listen({fd: 7})` Because the message is passed to the primary, file descriptor 7 **in the parent** will be listened on, and the handle passed to the worker, rather than listening to the worker's idea of what the number 7 file descriptor references.
2. `server.listen(handle)` Listening on handles explicitly will cause the worker to use the supplied handle, rather than talk to the primary process.
3. `server.listen(0)` Normally, this will cause servers to listen on a random port. However, in a cluster, each worker will receive the same "random" port each time they do `listen(0)`. In essence, the port is random the first time, but predictable thereafter. To listen on a unique port, generate a port number based on the cluster worker ID.

Node.js does not provide routing logic. It is, therefore important to design an application such that it does not rely too heavily on in-memory data objects for things like sessions and login.

Because workers are all separate processes, they can be killed or re-spawned depending on a program's needs, without affecting other workers. As long as there are some workers still alive, the server will continue to accept connections. If no workers are alive, existing connections will be dropped and new connections will be refused. Node.js does not automatically manage the number of workers, however. It is the application's responsibility to manage the worker pool based on its own needs.

Although a primary use case for the `cluster` module is networking, it can also be used for other use cases requiring worker processes.

Class: Worker

- Extends: `<EventEmitter>`

A `Worker` object contains all public information and method about a worker. In the primary it can be obtained using `cluster.workers`. In a worker it can be obtained using `cluster.worker`.

Event: 'disconnect'

Similar to the `cluster.on('disconnect')` event, but specific to this worker.

```
cluster.fork().on('disconnect', () => {  
  // Worker has disconnected  
});
```

Event: 'error'

This event is the same as the one provided by `child_process.fork()`.

Within a worker, `process.on('error')` may also be used.

Event: 'exit'

- `code` `<number>` The exit code, if it exited normally.
- `signal` `<string>` The name of the signal (e.g. `'SIGHUP'`) that caused the process to be killed.

Similar to the `cluster.on('exit')` event, but specific to this worker.

```
import cluster from 'cluster';  
  
const worker = cluster.fork();  
worker.on('exit', (code, signal) => {  
  if (signal) {  
    console.log(`worker was killed by signal: ${signal}`);  
  } else if (code !== 0) {  
    console.log(`worker exited with error code: ${code}`);  
  } else {  
    console.log('worker success!');  
  }  
});const cluster = require('cluster');  
  
const worker = cluster.fork();  
worker.on('exit', (code, signal) => {  
  if (signal) {  
    console.log(`worker was killed by signal: ${signal}`);  
  } else if (code !== 0) {  
    console.log(`worker exited with error code: ${code}`);  
  } else {  
    console.log('worker success!');  
  }  
});
```

Event: 'listening'

- `address` `<Object>`

Similar to the `cluster.on('listening')` event, but specific to this worker.

```
import cluster from 'cluster';  
  
cluster.fork().on('listening', (address) => {  
  // Worker is listening  
});
```

```
});const cluster = require('cluster');

cluster.fork().on('listening', (address) => {
  // Worker is listening
});
```

It is not emitted in the worker.

Event: 'message'

- `message` `<Object>`
- `handle` `<undefined> | <Object>`

Similar to the `'message'` event of `cluster`, but specific to this worker.

Within a worker, `process.on('message')` may also be used.

See [process event: 'message'](#).

Here is an example using the message system. It keeps a count in the primary process of the number of HTTP requests received by the workers:

```
import cluster from 'cluster';
import http from 'http';
import { cpus } from 'os';
import process from 'process';

if (cluster.isPrimary) {

  // Keep track of http requests
  let numReqs = 0;
  setInterval(() => {
    console.log(`numReqs = ${numReqs}`);
  }, 1000);

  // Count requests
  function messageHandler(msg) {
    if (msg.cmd && msg.cmd === 'notifyRequest') {
      numReqs += 1;
    }
  }

  // Start workers and listen for messages containing notifyRequest
  const numCPUs = cpus().length;
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
  }

  for (const id in cluster.workers) {
    cluster.workers[id].on('message', messageHandler);
  }
}
```

```

} else {

  // Worker processes have a http server.
  http.Server((req, res) => {
    res.writeHead(200);
    res.end('hello world\n');

    // Notify primary about the request
    process.send({ cmd: 'notifyRequest' });
  }).listen(8000);
}const cluster = require('cluster');
const http = require('http');
const process = require('process');

if (cluster.isPrimary) {

  // Keep track of http requests
  let numReqs = 0;
  setInterval(() => {
    console.log(`numReqs = ${numReqs}`);
  }, 1000);

  // Count requests
  function messageHandler(msg) {
    if (msg.cmd && msg.cmd === 'notifyRequest') {
      numReqs += 1;
    }
  }

  // Start workers and listen for messages containing notifyRequest
  const numCPUs = require('os').cpus().length;
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
  }

  for (const id in cluster.workers) {
    cluster.workers[id].on('message', messageHandler);
  }
} else {

  // Worker processes have a http server.
  http.Server((req, res) => {
    res.writeHead(200);
    res.end('hello world\n');

    // Notify primary about the request
    process.send({ cmd: 'notifyRequest' });
  }).listen(8000);
}

```

Event: 'online'

Similar to the `cluster.on('online')` event, but specific to this worker.

```
cluster.fork().on('online', () => {
  // Worker is online
});
```

It is not emitted in the worker.

worker.disconnect()

- Returns: `<cluster.Worker>` A reference to `worker`.

In a worker, this function will close all servers, wait for the `'close'` event on those servers, and then disconnect the IPC channel.

In the primary, an internal message is sent to the worker causing it to call `.disconnect()` on itself.

Causes `.exitedAfterDisconnect` to be set.

After a server is closed, it will no longer accept new connections, but connections may be accepted by any other listening worker. Existing connections will be allowed to close as usual. When no more connections exist, see `server.close()`, the IPC channel to the worker will close allowing it to die gracefully.

The above applies *only* to server connections, client connections are not automatically closed by workers, and `disconnect` does not wait for them to close before exiting.

In a worker, `process.disconnect` exists, but it is not this function; it is `disconnect()`.

Because long living server connections may block workers from disconnecting, it may be useful to send a message, so application specific actions may be taken to close them. It also may be useful to implement a timeout, killing a worker if the `'disconnect'` event has not been emitted after some time.

```
if (cluster.isPrimary) {
  const worker = cluster.fork();
  let timeout;

  worker.on('listening', (address) => {
    worker.send('shutdown');
    worker.disconnect();
    timeout = setTimeout(() => {
      worker.kill();
    }, 2000);
  });

  worker.on('disconnect', () => {
    clearTimeout(timeout);
  });
} else if (cluster.isWorker) {
  const net = require('net');
  const server = net.createServer((socket) => {
    // Connections never end
  });
}
```

```
server.listen(8000);

process.on('message', (msg) => {
  if (msg === 'shutdown') {
    // Initiate graceful close of any connections to server
  }
});
}
```

worker.exitedAfterDisconnect

- `<boolean>`

This property is `true` if the worker exited due to `.kill()` or `.disconnect()`. If the worker exited any other way, it is `false`. If the worker has not exited, it is `undefined`.

The boolean `worker.exitedAfterDisconnect` allows distinguishing between voluntary and accidental exit, the primary may choose not to respawn a worker based on this value.

```
cluster.on('exit', (worker, code, signal) => {
  if (worker.exitedAfterDisconnect === true) {
    console.log('Oh, it was just voluntary - no need to worry');
  }
});

// kill worker
worker.kill();
```

worker.id

- `<number>`

Each new worker is given its own unique id, this id is stored in the `id`.

While a worker is alive, this is the key that indexes it in `cluster.workers`.

worker.isConnected()

This function returns `true` if the worker is connected to its primary via its IPC channel, `false` otherwise. A worker is connected to its primary after it has been created. It is disconnected after the `'disconnect'` event is emitted.

worker.isDead()

This function returns `true` if the worker's process has terminated (either because of exiting or being signaled). Otherwise, it returns `false`.

```
import cluster from 'cluster';
import http from 'http';
import { cpus } from 'os';
import process from 'process';

const numCPUs = cpus().length;
```



```

if (cluster.isPrimary) {
  console.log(`Primary ${process.pid} is running`);

  // Fork workers.
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
  }

  cluster.on('fork', (worker) => {
    console.log('worker is dead:', worker.isDead());
  });

  cluster.on('exit', (worker, code, signal) => {
    console.log('worker is dead:', worker.isDead());
  });
} else {
  // Workers can share any TCP connection. In this case, it is an HTTP server.
  http.createServer((req, res) => {
    res.writeHead(200);
    res.end(`Current process\n ${process.pid}`);
    process.kill(process.pid);
  }).listen(8000);
}

const cluster = require('cluster');
const http = require('http');
const numCPUs = require('os').cpus().length;
const process = require('process');

if (cluster.isPrimary) {
  console.log(`Primary ${process.pid} is running`);

  // Fork workers.
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
  }

  cluster.on('fork', (worker) => {
    console.log('worker is dead:', worker.isDead());
  });

  cluster.on('exit', (worker, code, signal) => {
    console.log('worker is dead:', worker.isDead());
  });
} else {
  // Workers can share any TCP connection. In this case, it is an HTTP server.
  http.createServer((req, res) => {
    res.writeHead(200);
    res.end(`Current process\n ${process.pid}`);
    process.kill(process.pid);
  }).listen(8000);
}

```

worker.kill([signal])

- `signal` `<string>` Name of the kill signal to send to the worker process. **Default:** `'SIGTERM'`

This function will kill the worker. In the primary, it does this by disconnecting the `worker.process`, and once disconnected, killing with `signal`. In the worker, it does it by disconnecting the channel, and then exiting with code `0`.

Because `kill()` attempts to gracefully disconnect the worker process, it is susceptible to waiting indefinitely for the disconnect to complete. For example, if the worker enters an infinite loop, a graceful disconnect will never occur. If the graceful disconnect behavior is not needed, use `worker.process.kill()`.

Causes `.exitedAfterDisconnect` to be set.

This method is aliased as `worker.destroy()` for backward compatibility.

In a worker, `process.kill()` exists, but it is not this function; it is `kill()`.

worker.process

- `<ChildProcess>`

All workers are created using `child_process.fork()`, the returned object from this function is stored as `.process`. In a worker, the global `process` is stored.

See: [Child Process module](#).

Workers will call `process.exit(0)` if the `'disconnect'` event occurs on `process` and `.exitedAfterDisconnect` is not `true`. This protects against accidental disconnection.

worker.send(message[, sendHandle[, options]][, callback])

- `message` `<Object>`
- `sendHandle` `<Handle>`
- `options` `<Object>` The `options` argument, if present, is an object used to parameterize the sending of certain types of handles. `options` supports the following properties:
 - `keepOpen` `<boolean>` A value that can be used when passing instances of `net.Socket`. When `true`, the socket is kept open in the sending process. **Default:** `false`.
- `callback` `<Function>`
- Returns: `<boolean>`

Send a message to a worker or primary, optionally with a handle.

In the primary this sends a message to a specific worker. It is identical to `ChildProcess.send()`.

In a worker this sends a message to the primary. It is identical to `process.send()`.

This example will echo back all messages from the primary:

```
if (cluster.isPrimary) {
  const worker = cluster.fork();
  worker.send('hi there');

} else if (cluster.isWorker) {
  process.on('message', (msg) => {
    process.send(msg);
  });
}
```

```
});  
}
```

Event: 'disconnect'

- `worker` `<cluster.Worker>`

Emitted after the worker IPC channel has disconnected. This can occur when a worker exits gracefully, is killed, or is disconnected manually (such as with `worker.disconnect()`).

There may be a delay between the `'disconnect'` and `'exit'` events. These events can be used to detect if the process is stuck in a cleanup or if there are long-living connections.

```
cluster.on('disconnect', (worker) => {  
  console.log(`The worker #${worker.id} has disconnected`);  
});
```

Event: 'exit'

- `worker` `<cluster.Worker>`
- `code` `<number>` The exit code, if it exited normally.
- `signal` `<string>` The name of the signal (e.g. `'SIGHUP'`) that caused the process to be killed.

When any of the workers die the cluster module will emit the `'exit'` event.

This can be used to restart the worker by calling `.fork()` again.

```
cluster.on('exit', (worker, code, signal) => {  
  console.log(`worker %d died (%s). restarting...`,  
    worker.process.pid, signal || code);  
  cluster.fork();  
});
```

See `child_process` event: `'exit'`.

Event: 'fork'

- `worker` `<cluster.Worker>`

When a new worker is forked the cluster module will emit a `'fork'` event. This can be used to log worker activity, and create a custom timeout.

```
const timeouts = [];  
function errorMsg() {  
  console.error('Something must be wrong with the connection ...');  
}  
  
cluster.on('fork', (worker) => {  
  timeouts[worker.id] = setTimeout(errorMsg, 2000);  
});
```

```
cluster.on('listening', (worker, address) => {
  clearTimeout(timeouts[worker.id]);
});
cluster.on('exit', (worker, code, signal) => {
  clearTimeout(timeouts[worker.id]);
  errorMsg();
});
```

Event: 'listening'

- `worker` <cluster.Worker>
- `address` <Object>

After calling `listen()` from a worker, when the `'listening'` event is emitted on the server a `'listening'` event will also be emitted on `cluster` in the primary.

The event handler is executed with two arguments, the `worker` contains the worker object and the `address` object contains the following connection properties: `address`, `port` and `addressType`. This is very useful if the worker is listening on more than one address.

```
cluster.on('listening', (worker, address) => {
  console.log(
    `A worker is now connected to ${address.address}:${address.port}`);
});
```

The `addressType` is one of:

- 4 (TCPv4)
- 6 (TCPv6)
- -1 (Unix domain socket)
- 'udp4' or 'udp6' (UDP v4 or v6)

Event: 'message'

- `worker` <cluster.Worker>
- `message` <Object>
- `handle` <undefined> | <Object>

Emitted when the cluster primary receives a message from any worker.

See `child_process` event: `'message'`.

Event: 'online'

- `worker` <cluster.Worker>

After forking a new worker, the worker should respond with an online message. When the primary receives an online message it will emit this event. The difference between `'fork'` and `'online'` is that `fork` is emitted when the primary forks a worker, and `'online'` is emitted when the worker is running.

```
cluster.on('online', (worker) => {  
  console.log('Yay, the worker responded after it was forked');  
});
```

Event: 'setup'

- `settings` `<Object>`

Emitted every time `.setupPrimary()` is called.

The `settings` object is the `cluster.settings` object at the time `.setupPrimary()` was called and is advisory only, since multiple calls to `.setupPrimary()` can be made in a single tick.

If accuracy is important, use `cluster.settings`.

`cluster.disconnect([callback])`

- `callback` `<Function>` Called when all workers are disconnected and handles are closed.

Calls `.disconnect()` on each worker in `cluster.workers`.

When they are disconnected all internal handles will be closed, allowing the primary process to die gracefully if no other event is waiting.

The method takes an optional callback argument which will be called when finished.

This can only be called from the primary process.

`cluster.fork([env])`

- `env` `<Object>` Key/value pairs to add to worker process environment.
- Returns: `<cluster.Worker>`

Spawn a new worker process.

This can only be called from the primary process.

`cluster.isMaster`

Deprecated alias for `cluster.isPrimary` details.

`cluster.isPrimary`

- `<boolean>`

True if the process is a primary. This is determined by the `process.env.NODE_UNIQUE_ID`. If `process.env.NODE_UNIQUE_ID` is undefined, then `isPrimary` is `true`.

`cluster.isWorker`

- `<boolean>`

True if the process is not a primary (it is the negation of `cluster.isPrimary`).

cluster.schedulingPolicy

The scheduling policy, either `cluster.SCHED_RR` for round-robin or `cluster.SCHED_NONE` to leave it to the operating system. This is a global setting and effectively frozen once either the first worker is spawned, or `.setupPrimary()` is called, whichever comes first.

`SCHED_RR` is the default on all operating systems except Windows. Windows will change to `SCHED_RR` once libuv is able to effectively distribute IOCP handles without incurring a large performance hit.

`cluster.schedulingPolicy` can also be set through the `NODE_CLUSTER_SCHED_POLICY` environment variable. Valid values are `'rr'` and `'none'`.

cluster.settings

- `<Object>`
 - `execArgv <string[]>` List of string arguments passed to the Node.js executable. **Default:** `process.execArgv`.
 - `exec <string>` File path to worker file. **Default:** `process.argv[1]`.
 - `args <string[]>` String arguments passed to worker. **Default:** `process.argv.slice(2)`.
 - `cwd <string>` Current working directory of the worker process. **Default:** `undefined` (inherits from parent process).
 - `serialization <string>` Specify the kind of serialization used for sending messages between processes. Possible values are `'json'` and `'advanced'`. See [Advanced serialization for child_process](#) for more details. **Default:** `false`.
 - `silent <boolean>` Whether or not to send output to parent's stdio. **Default:** `false`.
 - `stdio <Array>` Configures the stdio of forked processes. Because the cluster module relies on IPC to function, this configuration must contain an `'ipc'` entry. When this option is provided, it overrides `silent`.
 - `uid <number>` Sets the user identity of the process. (See `setuid(2)`.)
 - `gid <number>` Sets the group identity of the process. (See `setgid(2)`.)
 - `inspectPort <number> | <Function>` Sets inspector port of worker. This can be a number, or a function that takes no arguments and returns a number. By default each worker gets its own port, incremented from the primary's `process.debugPort`.
 - `windowsHide <boolean>` Hide the forked processes console window that would normally be created on Windows systems. **Default:** `false`.

After calling `.setupPrimary()` (or `.fork()`) this settings object will contain the settings, including the default values.

This object is not intended to be changed or set manually.

cluster.setupMaster([settings])

Deprecated alias for `.setupPrimary()`.

cluster.setupPrimary([settings])

- `settings <Object>` See `cluster.settings`.

`setupPrimary` is used to change the default 'fork' behavior. Once called, the settings will be present in `cluster.settings`.

Any settings changes only affect future calls to `.fork()` and have no effect on workers that are already running.

The only attribute of a worker that cannot be set via `.setupPrimary()` is the `env` passed to `.fork()`.

The defaults above apply to the first call only; the defaults for later calls are the current values at the time of `cluster.setupPrimary()` is called.

```

import cluster from 'cluster';

cluster.setupPrimary({
  exec: 'worker.js',
  args: ['--use', 'https'],
  silent: true
});
cluster.fork(); // https worker
cluster.setupPrimary({
  exec: 'worker.js',
  args: ['--use', 'http']
});
cluster.fork(); // http workerconst cluster = require('cluster');

cluster.setupPrimary({
  exec: 'worker.js',
  args: ['--use', 'https'],
  silent: true
});
cluster.fork(); // https worker
cluster.setupPrimary({
  exec: 'worker.js',
  args: ['--use', 'http']
});
cluster.fork(); // http worker

```

This can only be called from the primary process.

cluster.worker

- `<Object>`

A reference to the current worker object. Not available in the primary process.

```

import cluster from 'cluster';

if (cluster.isPrimary) {
  console.log('I am primary');
  cluster.fork();
  cluster.fork();
} else if (cluster.isWorker) {
  console.log(`I am worker #${cluster.worker.id}`);
}const cluster = require('cluster');

if (cluster.isPrimary) {
  console.log('I am primary');
  cluster.fork();
  cluster.fork();
} else if (cluster.isWorker) {

```

```
console.log(`I am worker #${cluster.worker.id}`);  
}
```

cluster.workers

- `<Object>`

A hash that stores the active worker objects, keyed by `id` field. Makes it easy to loop through all the workers. It is only available in the primary process.

A worker is removed from `cluster.workers` after the worker has disconnected *and* exited. The order between these two events cannot be determined in advance. However, it is guaranteed that the removal from the `cluster.workers` list happens before last `'disconnect'` or `'exit'` event is emitted.

```
import cluster from 'cluster';  
  
// Go through all workers  
function eachWorker(callback) {  
  for (const id in cluster.workers) {  
    callback(cluster.workers[id]);  
  }  
}  
  
eachWorker((worker) => {  
  worker.send('big announcement to all workers');  
});  
  
const cluster = require('cluster');  
  
// Go through all workers  
function eachWorker(callback) {  
  for (const id in cluster.workers) {  
    callback(cluster.workers[id]);  
  }  
}  
  
eachWorker((worker) => {  
  worker.send('big announcement to all workers');  
});
```

Using the worker's unique id is the easiest way to locate the worker.

```
socket.on('data', (id) => {  
  const worker = cluster.workers[id];  
});
```

Command-line options

Node.js comes with a variety of CLI options. These options expose built-in debugging, multiple ways to execute scripts, and other helpful runtime options.

To view this documentation as a manual page in a terminal, run `man node`.

Synopsis

```
node [options] [V8 options] [script.js | -e "script" | -] [--] [arguments]
```

```
node inspect [script.js | -e "script" | <host>:<port>] ...
```

```
node --v8-options
```

Execute without arguments to start the [REPL](#).

For more info about `node inspect`, see the [debugger](#) documentation.

Options

All options, including V8 options, allow words to be separated by both dashes (`-`) or underscores (`_`). For example, `--pending-deprecation` is equivalent to `--pending_deprecation`.

If an option that takes a single value (such as `--max-http-header-size`) is passed more than once, then the last passed value is used. Options from the command line take precedence over options passed through the `NODE_OPTIONS` environment variable.

```
-
```

Alias for stdin. Analogous to the use of `-` in other command-line utilities, meaning that the script is read from stdin, and the rest of the options are passed to that script.

```
--
```

Indicate the end of node options. Pass the rest of the arguments to the script. If no script filename or eval/print script is supplied prior to this, then the next argument is used as a script filename.

--abort-on-uncaught-exception

Aborting instead of exiting causes a core file to be generated for post-mortem analysis using a debugger (such as `lldb`, `gdb`, and `mdb`).

If this flag is passed, the behavior can still be set to not abort through `process.setUncaughtExceptionCaptureCallback()` (and through usage of the `domain` module that uses it).

--completion-bash

Print source-able bash completion script for Node.js.

```
$ node --completion-bash > node_bash_completion
$ source node_bash_completion
```

-C=condition, --conditions=condition

Stability: 1 - Experimental

Enable experimental support for custom [conditional exports](#) resolution conditions.

Any number of custom string condition names are permitted.

The default Node.js conditions of `"node"`, `"default"`, `"import"`, and `"require"` will always apply as defined.

For example, to run a module with "development" resolutions:

```
$ node -C=development app.js
```

--cpu-prof

Stability: 1 - Experimental

Starts the V8 CPU profiler on start up, and writes the CPU profile to disk before exit.

If `--cpu-prof-dir` is not specified, the generated profile is placed in the current working directory.

If `--cpu-prof-name` is not specified, the generated profile is named `CPU.{yyyymmdd}.{hhmmss}.{pid}.{tid}.{seq}.cpuprofile`.

```
$ node --cpu-prof index.js
$ ls *.cpuprofile
CPU.20190409.202950.15293.0.0.cpuprofile
```

--cpu-prof-dir

Stability: 1 - Experimental

Specify the directory where the CPU profiles generated by `--cpu-prof` will be placed.

The default value is controlled by the `--diagnostic-dir` command-line option.

--cpu-prof-interval

Stability: 1 - Experimental

Specify the sampling interval in microseconds for the CPU profiles generated by `--cpu-prof`. The default is 1000 microseconds.

--cpu-prof-name

Stability: 1 - Experimental

Specify the file name of the CPU profile generated by `--cpu-prof`.

--diagnostic-dir=directory

Set the directory to which all diagnostic output files are written. Defaults to current working directory.

Affects the default output directory of:

- `--cpu-prof-dir`
- `--heap-prof-dir`

- `--redirect-warnings`

`--disable-proto=mode`

Disable the `Object.prototype.__proto__` property. If `mode` is `delete`, the property is removed entirely. If `mode` is `throw`, accesses to the property throw an exception with the code `ERR_PROTO_ACCESS`.

`--disallow-code-generation-from-strings`

Make built-in language features like `eval` and `new Function` that generate code from strings throw an exception instead. This does not affect the Node.js `vm` module.

`--dns-result-order=order`

Set the default value of `verbatim` in `dns.lookup()` and `dnsPromises.lookup()`. The value could be:

- `ipv4first`: sets default `verbatim` `false`.
- `verbatim`: sets default `verbatim` `true`.

The default is `ipv4first` and `dns.setDefaultResultOrder()` have higher priority than `--dns-result-order`.

`--enable-fips`

Enable FIPS-compliant crypto at startup. (Requires Node.js to be built against FIPS-compatible OpenSSL.)

`--enable-source-maps`

Enable [Source Map v3](#) support for stack traces.

When using a transpiler, such as TypeScript, stack traces thrown by an application reference the transpiled code, not the original source position. `--enable-source-maps` enables caching of Source Maps and makes a best effort to report stack traces relative to the original source file.

Overriding `Error.prepareStackTrace` prevents `--enable-source-maps` from modifying the stack trace.

`--experimental-abortcontroller`

`AbortController` and `AbortSignal` support is enabled by default. Use of this command-line flag is no longer required.

`--experimental-import-meta-resolve`

Enable experimental `import.meta.resolve()` support.

`--experimental-json-modules`

Enable experimental JSON support for the ES Module loader.

`--experimental-loader=module`

Specify the `module` of a custom experimental [ECMAScript Module loader](#). `module` may be either a path to a file, or an ECMAScript Module name.

`--experimental-modules`

Enable latest experimental modules features (deprecated).

`--experimental-policy`

Use the specified file as a security policy.

--no-experimental-repl-await

Use this flag to disable top-level await in REPL.

--experimental-specifier-resolution=mode

Sets the resolution algorithm for resolving ES module specifiers. Valid options are `explicit` and `node`.

The default is `explicit`, which requires providing the full path to a module. The `node` mode enables support for optional file extensions and the ability to import a directory that has an index file.

See [customizing ESM specifier resolution](#) for example usage.

--experimental-vm-modules

Enable experimental ES Module support in the `vm` module.

--experimental-wasi-unstable-preview1

Enable experimental WebAssembly System Interface (WASI) support.

--experimental-wasm-modules

Enable experimental WebAssembly module support.

--force-context-aware

Disable loading native addons that are not `context-aware`.

--force-fips

Force FIPS-compliant crypto on startup. (Cannot be disabled from script code.) (Same requirements as `--enable-fips`.)

--frozen-intrinsics

Stability: 1 - Experimental

Enable experimental frozen intrinsics like `Array` and `Object`.

Support is currently only provided for the root context and no guarantees are currently provided that `global.Array` is indeed the default intrinsic reference. Code may break under this flag.

`--require` runs prior to freezing intrinsics in order to allow polyfills to be added.

--heapsnapshot-near-heap-limit=max_count

Stability: 1 - Experimental

Writes a V8 heap snapshot to disk when the V8 heap usage is approaching the heap limit. `count` should be a non-negative integer (in which case Node.js will write no more than `max_count` snapshots to disk).

When generating snapshots, garbage collection may be triggered and bring the heap usage down, therefore multiple snapshots may be written to disk before the Node.js instance finally runs out of memory. These heap snapshots can be compared to determine what objects are being allocated during the time consecutive snapshots are taken. It's not guaranteed that Node.js will write exactly `max_count` snapshots to disk, but it will try its best to generate at least one and up to `max_count` snapshots before the Node.js instance runs out of memory when `max_count` is greater than `0`.

Generating V8 snapshots takes time and memory (both memory managed by the V8 heap and native memory outside the V8 heap). The bigger the heap is, the more resources it needs. Node.js will adjust the V8 heap to accommodate the additional V8 heap memory overhead, and try its best to avoid using up all the memory available to the process. When the process uses more memory than the system deems appropriate, the process may be terminated abruptly by the system, depending on the system configuration.

```
$ node --max-old-space-size=100 --heapsnapshot-near-heap-limit=3 index.js
Wrote snapshot to Heap.20200430.100036.49580.0.001.heapsnapshot
Wrote snapshot to Heap.20200430.100037.49580.0.002.heapsnapshot
Wrote snapshot to Heap.20200430.100038.49580.0.003.heapsnapshot

<--- Last few GCs --->

[49580:0x110000000]      4826 ms: Mark-sweep 130.6 (147.8) -> 130.5 (147.8) MB, 27.4 / 0.0 ms (average mu = 0.126, current mu = 0.088)
[49580:0x110000000]      4845 ms: Mark-sweep 130.6 (147.8) -> 130.6 (147.8) MB, 18.8 / 0.0 ms (average mu = 0.088, current mu = 0.088)

<--- JS stacktrace --->

FATAL ERROR: Ineffective mark-compacts near heap limit Allocation failed - JavaScript heap out of memory
....
```

--heapsnapshot-signal=signal

Enables a signal handler that causes the Node.js process to write a heap dump when the specified signal is received. `signal` must be a valid signal name. Disabled by default.

```
$ node --heapsnapshot-signal=SIGUSR2 index.js &
$ ps aux
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
node         1  5.5  6.1 787252 247004 ?        Ssl  16:43   0:02 node --heapsnapshot-signal=SIGUSR2 index.js
$ kill -USR2 1
$ ls
Heap.20190718.133405.15554.0.001.heapsnapshot
```

--heap-prof

Stability: 1 - Experimental

Starts the V8 heap profiler on start up, and writes the heap profile to disk before exit.

If `--heap-prof-dir` is not specified, the generated profile is placed in the current working directory.

If `--heap-prof-name` is not specified, the generated profile is named `Heap.{yyyymmdd}.{hhmmss}.{pid}.{tid}.{seq}.heapprofile`.

```
$ node --heap-prof index.js
$ ls *.heapprofile
Heap.20190409.202950.15293.0.001.heapprofile
```

--heap-prof-dir

Stability: 1 - Experimental

Specify the directory where the heap profiles generated by `--heap-prof` will be placed.

The default value is controlled by the `--diagnostic-dir` command-line option.

--heap-prof-interval

Stability: 1 - Experimental

Specify the average sampling interval in bytes for the heap profiles generated by `--heap-prof`. The default is $512 * 1024$ bytes.

--heap-prof-name

Stability: 1 - Experimental

Specify the file name of the heap profile generated by `--heap-prof`.

--icu-data-dir=file

Specify ICU data load path. (Overrides `NODE_ICU_DATA`.)

--input-type=type

This configures Node.js to interpret string input as CommonJS or as an ES module. String input is input via `--eval`, `--print`, or `STDIN`.

Valid values are `"commonjs"` and `"module"`. The default is `"commonjs"`.

--inspect-brk[=[host:]port]

Activate inspector on `host:port` and break at start of user script. Default `host:port` is `127.0.0.1:9229`.

--inspect-port=[host:]port

Set the `host:port` to be used when the inspector is activated. Useful when activating the inspector by sending the `SIGUSR1` signal.

Default host is `127.0.0.1`.

See the [security warning](#) below regarding the `host` parameter usage.

--inspect[=[host:]port]

Activate inspector on `host:port`. Default is `127.0.0.1:9229`.

V8 inspector integration allows tools such as Chrome DevTools and IDEs to debug and profile Node.js instances. The tools attach to Node.js instances via a tcp port and communicate using the [Chrome DevTools Protocol](#).

Warning: binding inspector to a public IP:port combination is insecure

Binding the inspector to a public IP (including `0.0.0.0`) with an open port is insecure, as it allows external hosts to connect to the inspector and perform a [remote code execution](#) attack.

If specifying a host, make sure that either:

- The host is not accessible from public networks.
- A firewall disallows unwanted connections on the port.

More specifically, `--inspect=0.0.0.0` is insecure if the port (`9229` by default) is not firewall-protected.

See the [debugging security implications](#) section for more information.

--inspect-publish-uid=stderr,http

Specify ways of the inspector web socket url exposure.

By default inspector websocket url is available in stderr and under `/json/list` endpoint on `http://host:port/json/list`.

--insecure-http-parser

Use an insecure HTTP parser that accepts invalid HTTP headers. This may allow interoperability with non-conformant HTTP implementations. It may also allow request smuggling and other HTTP attacks that rely on invalid headers being accepted. Avoid using this option.

--jitless

Disable [runtime allocation of executable memory](#). This may be required on some platforms for security reasons. It can also reduce attack surface on other platforms, but the performance impact may be severe.

This flag is inherited from V8 and is subject to change upstream. It may disappear in a non-semver-major release.

--max-http-header-size=size

Specify the maximum size, in bytes, of HTTP headers. Defaults to 16 KB.

--napi-modules

This option is a no-op. It is kept for compatibility.

--no-deprecation

Silence deprecation warnings.

--no-force-async-hooks-checks

Disables runtime checks for `async_hooks`. These will still be enabled dynamically when `async_hooks` is enabled.

--no-warnings

Silence all process warnings (including deprecations).

--node-memory-debug

Enable extra debug checks for memory leaks in Node.js internals. This is usually only useful for developers debugging Node.js itself.

--openssl-config=file

Load an OpenSSL configuration file on startup. Among other uses, this can be used to enable FIPS-compliant crypto if Node.js is built against FIPS-enabled OpenSSL.

--pending-deprecation

Emit pending deprecation warnings.

Pending deprecations are generally identical to a runtime deprecation with the notable exception that they are turned *off* by default and will not be emitted unless either the `--pending-deprecation` command-line flag, or the `NODE_PENDING_DEPRECATION=1` environment variable, is set. Pending deprecations are used to provide a kind of selective "early warning" mechanism that developers may leverage to detect deprecated API usage.

--policy-integrity=sri

Stability: 1 - Experimental

Instructs Node.js to error prior to running any code if the policy does not have the specified integrity. It expects a [Subresource Integrity](#) string as a parameter.

--preserve-symlinks

Instructs the module loader to preserve symbolic links when resolving and caching modules.

By default, when Node.js loads a module from a path that is symbolically linked to a different on-disk location, Node.js will dereference the link and use the actual on-disk "real path" of the module as both an identifier and as a root path to locate other dependency modules. In most cases, this default behavior is acceptable. However, when using symbolically linked peer dependencies, as illustrated in the example below, the default behavior causes an exception to be thrown if `moduleA` attempts to require `moduleB` as a peer dependency:

```
{appDir}
├─ app
│   ├─ index.js
│   └─ node_modules
│       ├─ moduleA -> {appDir}/moduleA
│       └─ moduleB
│           ├─ index.js
│           └─ package.json
└─ moduleA
    ├─ index.js
    └─ package.json
```

The `--preserve-symlinks` command-line flag instructs Node.js to use the symlink path for modules as opposed to the real path, allowing symbolically linked peer dependencies to be found.

Note, however, that using `--preserve-symlinks` can have other side effects. Specifically, symbolically linked *native* modules can fail to load if those are linked from more than one location in the dependency tree (Node.js would see those as two separate modules and would attempt to load the module multiple times, causing an exception to be thrown).

The `--preserve-symlinks` flag does not apply to the main module, which allows `node --preserve-symlinks node_module/.bin/<foo>` to work. To apply the same behavior for the main module, also use `--preserve-symlinks-main`.

`--preserve-symlinks-main`

Instructs the module loader to preserve symbolic links when resolving and caching the main module (`require.main`).

This flag exists so that the main module can be opted-in to the same behavior that `--preserve-symlinks` gives to all other imports; they are separate flags, however, for backward compatibility with older Node.js versions.

`--preserve-symlinks-main` does not imply `--preserve-symlinks`; use `--preserve-symlinks-main` in addition to `--preserve-symlinks` when it is not desirable to follow symlinks before resolving relative paths.

See `--preserve-symlinks` for more information.

`--prof`

Generate V8 profiler output.

`--prof-process`

Process V8 profiler output generated using the V8 option `--prof`.

`--redirect-warnings=file`

Write process warnings to the given file instead of printing to stderr. The file will be created if it does not exist, and will be appended to if it does. If an error occurs while attempting to write the warning to the file, the warning will be written to stderr instead.

The `file` name may be an absolute path. If it is not, the default directory it will be written to is controlled by the `--diagnostic-dir` command-line option.

`--report-compact`

Write reports in a compact format, single-line JSON, more easily consumable by log processing systems than the default multi-line format designed for human consumption.

`--report-dir=directory`, `report-directory=directory`

Location at which the report will be generated.

`--report-filename=filename`

Name of the file to which the report will be written.

`--report-on-fatalerror`

Enables the report to be triggered on fatal errors (internal errors within the Node.js runtime such as out of memory) that lead to termination of the application. Useful to inspect various diagnostic data elements such as heap, stack, event loop state, resource consumption etc. to reason about the fatal error.

`--report-on-signal`

Enables report to be generated upon receiving the specified (or predefined) signal to the running Node.js process. The signal to trigger the report is specified through `--report-signal`.

`--report-signal=signal`

Sets or resets the signal for report generation (not supported on Windows). Default signal is `SIGUSR2`.

--report-uncaught-exception

Enables report to be generated on uncaught exceptions. Useful when inspecting the JavaScript stack in conjunction with native stack and other runtime environment data.

--secure-heap=n

Initializes an OpenSSL secure heap of `n` bytes. When initialized, the secure heap is used for selected types of allocations within OpenSSL during key generation and other operations. This is useful, for instance, to prevent sensitive information from leaking due to pointer overruns or underruns.

The secure heap is a fixed size and cannot be resized at runtime so, if used, it is important to select a large enough heap to cover all application uses.

The heap size given must be a power of two. Any value less than 2 will disable the secure heap.

The secure heap is disabled by default.

The secure heap is not available on Windows.

See `CRYPTO_secure_malloc_init` for more details.

--secure-heap-min=n

When using `--secure-heap`, the `--secure-heap-min` flag specifies the minimum allocation from the secure heap. The minimum value is `2`. The maximum value is the lesser of `--secure-heap` or `2147483647`. The value given must be a power of two.

--throw-deprecation

Throw errors for deprecations.

--title=title

Set `process.title` on startup.

--tls-cipher-list=list

Specify an alternative default TLS cipher list. Requires Node.js to be built with crypto support (default).

--tls-keylog=file

Log TLS key material to a file. The key material is in NSS `SSLKEYLOGFILE` format and can be used by software (such as Wireshark) to decrypt the TLS traffic.

--tls-max-v1.2

Set `tls.DEFAULT_MAX_VERSION` to 'TLSv1.2'. Use to disable support for TLSv1.3.

--tls-max-v1.3

Set default `tls.DEFAULT_MAX_VERSION` to 'TLSv1.3'. Use to enable support for TLSv1.3.

--tls-min-v1.0

Set default `tls.DEFAULT_MIN_VERSION` to 'TLSv1'. Use for compatibility with old TLS clients or servers.

--tls-min-v1.1

Set default `tls.DEFAULT_MIN_VERSION` to 'TLSv1.1'. Use for compatibility with old TLS clients or servers.

--tls-min-v1.2

Set default `tls.DEFAULT_MIN_VERSION` to 'TLSv1.2'. This is the default for 12.x and later, but the option is supported for compatibility with older Node.js versions.

--tls-min-v1.3

Set default `tls.DEFAULT_MIN_VERSION` to 'TLSv1.3'. Use to disable support for TLSv1.2, which is not as secure as TLSv1.3.

--trace-atomics-wait

Print short summaries of calls to `Atomics.wait()` to stderr. The output could look like this:

```
(node:15701) [Thread 0] Atomics.wait(&lt;address> + 0, 1, inf) started
(node:15701) [Thread 0] Atomics.wait(&lt;address> + 0, 1, inf) did not wait because the values mismatched
(node:15701) [Thread 0] Atomics.wait(&lt;address> + 0, 0, 10) started
(node:15701) [Thread 0] Atomics.wait(&lt;address> + 0, 0, 10) timed out
(node:15701) [Thread 0] Atomics.wait(&lt;address> + 4, 0, inf) started
(node:15701) [Thread 1] Atomics.wait(&lt;address> + 4, -1, inf) started
(node:15701) [Thread 0] Atomics.wait(&lt;address> + 4, 0, inf) was woken up by another thread
(node:15701) [Thread 1] Atomics.wait(&lt;address> + 4, -1, inf) was woken up by another thread
```

The fields here correspond to:

- The thread id as given by `worker_threads.threadId`
- The base address of the `SharedArrayBuffer` in question, as well as the byte offset corresponding to the index passed to `Atomics.wait()`
- The expected value that was passed to `Atomics.wait()`
- The timeout passed to `Atomics.wait`

--trace-deprecation

Print stack traces for deprecations.

--trace-event-categories

A comma separated list of categories that should be traced when trace event tracing is enabled using `--trace-events-enabled`.

--trace-event-file-pattern

Template string specifying the filepath for the trace event data, it supports `${rotation}` and `${pid}`.

--trace-events-enabled

Enables the collection of trace event tracing information.

--trace-exit

Prints a stack trace whenever an environment is exited proactively, i.e. invoking `process.exit()`.

--trace-sigint

Prints a stack trace on SIGINT.

--trace-sync-io

Prints a stack trace whenever synchronous I/O is detected after the first turn of the event loop.

--trace-tls

Prints TLS packet trace information to `stderr`. This can be used to debug TLS connection problems.

--trace-uncaught

Print stack traces for uncaught exceptions; usually, the stack trace associated with the creation of an `Error` is printed, whereas this makes Node.js also print the stack trace associated with throwing the value (which does not need to be an `Error` instance).

Enabling this option may affect garbage collection behavior negatively.

--trace-warnings

Print stack traces for process warnings (including deprecations).

--track-heap-objects

Track heap object allocations for heap snapshots.

--unhandled-rejections=mode

Using this flag allows to change what should happen when an unhandled rejection occurs. One of the following modes can be chosen:

- `throw`: Emit `unhandledRejection`. If this hook is not set, raise the unhandled rejection as an uncaught exception. This is the default.
- `strict`: Raise the unhandled rejection as an uncaught exception.
- `warn`: Always trigger a warning, no matter if the `unhandledRejection` hook is set or not but do not print the deprecation warning.
- `warn-with-error-code`: Emit `unhandledRejection`. If this hook is not set, trigger a warning, and set the process exit code to 1.
- `none`: Silence all warnings.

--use-bundled-ca, --use-openssl-ca

Use bundled Mozilla CA store as supplied by current Node.js version or use OpenSSL's default CA store. The default store is selectable at build-time.

The bundled CA store, as supplied by Node.js, is a snapshot of Mozilla CA store that is fixed at release time. It is identical on all supported platforms.

Using OpenSSL store allows for external modifications of the store. For most Linux and BSD distributions, this store is maintained by the distribution maintainers and system administrators. OpenSSL CA store location is dependent on configuration of the OpenSSL library but this can be altered at runtime using environment variables.

See `SSL_CERT_DIR` and `SSL_CERT_FILE`.

--use-largepages=mode

Re-map the Node.js static code to large memory pages at startup. If supported on the target system, this will cause the Node.js static code to be moved onto 2 MiB pages instead of 4 KiB pages.

The following values are valid for `mode` :

- `off` : No mapping will be attempted. This is the default.
- `on` : If supported by the OS, mapping will be attempted. Failure to map will be ignored and a message will be printed to standard error.
- `silent` : If supported by the OS, mapping will be attempted. Failure to map will be ignored and will not be reported.

`--v8-options`

Print V8 command-line options.

`--v8-pool-size=num`

Set V8's thread pool size which will be used to allocate background jobs.

If set to `0` then V8 will choose an appropriate size of the thread pool based on the number of online processors.

If the value provided is larger than V8's maximum, then the largest value will be chosen.

`--zero-fill-buffers`

Automatically zero-fills all newly allocated `Buffer` and `SlowBuffer` instances.

`-c, --check`

Syntax check the script without executing.

`-e, --eval "script"`

Evaluate the following argument as JavaScript. The modules which are predefined in the REPL can also be used in `script` .

On Windows, using `cmd.exe` a single quote will not work correctly because it only recognizes double `"` for quoting. In Powershell or Git bash, both `'` and `"` are usable.

`-h, --help`

Print node command-line options. The output of this option is less detailed than this document.

`-i, --interactive`

Opens the REPL even if stdin does not appear to be a terminal.

`-p, --print "script"`

Identical to `-e` but prints the result.

`-r, --require module`

Preload the specified module at startup.

Follows `require()` 's module resolution rules. `module` may be either a path to a file, or a node module name.

Only CommonJS modules are supported. Attempting to preload a ES6 Module using `--require` will fail with an error.

`-v, --version`

Print node's version.

Environment variables

FORCE_COLOR=[1, 2, 3]

The `FORCE_COLOR` environment variable is used to enable ANSI colorized output. The value may be:

- `1`, `true`, or the empty string `''` indicate 16-color support,
- `2` to indicate 256-color support, or
- `3` to indicate 16 million-color support.

When `FORCE_COLOR` is used and set to a supported value, both the `NO_COLOR`, and `NODE_DISABLE_COLORS` environment variables are ignored.

Any other value will result in colorized output being disabled.

NODE_DEBUG=module[,...]

`' , '`-separated list of core modules that should print debug information.

NODE_DEBUG_NATIVE=module[,...]

`' , '`-separated list of core C++ modules that should print debug information.

NODE_DISABLE_COLORS=1

When set, colors will not be used in the REPL.

NODE_EXTRA_CA_CERTS=file

When set, the well known "root" CAs (like VeriSign) will be extended with the extra certificates in `file`. The file should consist of one or more trusted certificates in PEM format. A message will be emitted (once) with `process.emitWarning()` if the file is missing or malformed, but any errors are otherwise ignored.

Neither the well known nor extra certificates are used when the `ca` options property is explicitly specified for a TLS or HTTPS client or server.

This environment variable is ignored when `node` runs as `setuid root` or has Linux file capabilities set.

The `NODE_EXTRA_CA_CERTS` environment variable is only read when the Node.js process is first launched. Changing the value at runtime using `process.env.NODE_EXTRA_CA_CERTS` has no effect on the current process.

NODE_ICU_DATA=file

Data path for ICU (`Intl` object) data. Will extend linked-in data when compiled with small-icu support.

NODE_NO_WARNINGS=1

When set to `1`, process warnings are silenced.

NODE_OPTIONS=options...

A space-separated list of command-line options. `options...` are interpreted before command-line options, so command-line options will override or compound after anything in `options...`. Node.js will exit with an error if an option that is not allowed in the environment is used, such as `-p` or a script file.

If an option value contains a space, it can be escaped using double quotes:

```
NODE_OPTIONS='--require "./my path/file.js"'
```

A singleton flag passed as a command-line option will override the same flag passed into `NODE_OPTIONS` :

```
# The inspector will be available on port 5555
NODE_OPTIONS='--inspect=localhost:4444' node --inspect=localhost:5555
```

A flag that can be passed multiple times will be treated as if its `NODE_OPTIONS` instances were passed first, and then its command-line instances afterwards:

```
NODE_OPTIONS='--require "./a.js"' node --require "./b.js"
# is equivalent to:
node --require "./a.js" --require "./b.js"
```

Node.js options that are allowed are:

- `--conditions, -C`
- `--diagnostic-dir`
- `--disable-proto`
- `--dns-result-order`
- `--enable-fips`
- `--enable-source-maps`
- `--experimental-abortcontroller`
- `--experimental-import-meta-resolve`
- `--experimental-json-modules`
- `--experimental-loader`
- `--experimental-modules`
- `--experimental-policy`
- `--experimental-specifier-resolution`
- `--experimental-top-level-await`
- `--experimental-vm-modules`
- `--experimental-wasi-unstable-preview1`
- `--experimental-wasm-modules`
- `--force-context-aware`
- `--force-fips`
- `--frozen-intrinsics`
- `--heapsnapshot-near-heap-limit`
- `--heapsnapshot-signal`
- `--http-parser`
- `--icu-data-dir`
- `--input-type`
- `--insecure-http-parser`
- `--inspect-brk`

- `--inspect-port` , `--debug-port`
- `--inspect-publish-uid`
- `--inspect`
- `--max-http-header-size`
- `--napi-modules`
- `--no-deprecation`
- `--no-experimental-repl-await`
- `--no-force-async-hooks-checks`
- `--no-warnings`
- `--node-memory-debug`
- `--openssl-config`
- `--pending-deprecation`
- `--policy-integrity`
- `--preserve-symlinks-main`
- `--preserve-symlinks`
- `--prof-process`
- `--redirect-warnings`
- `--report-compact`
- `--report-dir` , `--report-directory`
- `--report-filename`
- `--report-on-fatalerror`
- `--report-on-signal`
- `--report-signal`
- `--report-uncaught-exception`
- `--require` , `-r`
- `--secure-heap-min`
- `--secure-heap`
- `--throw-deprecation`
- `--title`
- `--tls-cipher-list`
- `--tls-keylog`
- `--tls-max-v1.2`
- `--tls-max-v1.3`
- `--tls-min-v1.0`
- `--tls-min-v1.1`
- `--tls-min-v1.2`
- `--tls-min-v1.3`
- `--trace-atomics-wait`
- `--trace-deprecation`
- `--trace-event-categories`
- `--trace-event-file-pattern`

- `--trace-events-enabled`
- `--trace-exit`
- `--trace-sigint`
- `--trace-sync-io`
- `--trace-tls`
- `--trace-uncaught`
- `--trace-warnings`
- `--track-heap-objects`
- `--unhandled-rejections`
- `--use-bundled-ca`
- `--use-largepages`
- `--use-openssl-ca`
- `--v8-pool-size`
- `--zero-fill-buffers`

V8 options that are allowed are:

- `--abort-on-uncaught-exception`
- `--disallow-code-generation-from-strings`
- `--huge-max-old-generation-size`
- `--interpreted-frames-native-stack`
- `--jitless`
- `--max-old-space-size`
- `--perf-basic-prof-only-functions`
- `--perf-basic-prof`
- `--perf-prof-unwinding-info`
- `--perf-prof`
- `--stack-trace-limit`

`--perf-basic-prof-only-functions`, `--perf-basic-prof`, `--perf-prof-unwinding-info`, and `--perf-prof` are only available on Linux.

`NODE_PATH=path[:...]`

`':'`-separated list of directories prefixed to the module search path.

On Windows, this is a `';'` -separated list instead.

`NODE_PENDING_DEPRECATION=1`

When set to `1`, emit pending deprecation warnings.

Pending deprecations are generally identical to a runtime deprecation with the notable exception that they are turned *off* by default and will not be emitted unless either the `--pending-deprecation` command-line flag, or the `NODE_PENDING_DEPRECATION=1` environment variable, is set. Pending deprecations are used to provide a kind of selective "early warning" mechanism that developers may leverage to detect deprecated API usage.

`NODE_PENDING_PIPE_INSTANCES=instances`

Set the number of pending pipe instance handles when the pipe server is waiting for connections. This setting applies to Windows only.

NODE_PRESERVE_SYMLINKS=1

When set to `1`, instructs the module loader to preserve symbolic links when resolving and caching modules.

NODE_REDIRECT_WARNINGS=file

When set, process warnings will be emitted to the given file instead of printing to stderr. The file will be created if it does not exist, and will be appended to if it does. If an error occurs while attempting to write the warning to the file, the warning will be written to stderr instead. This is equivalent to using the `--redirect-warnings=file` command-line flag.

NODE_REPL_HISTORY=file

Path to the file used to store the persistent REPL history. The default path is `~/.node_repl_history`, which is overridden by this variable. Setting the value to an empty string (`' '` or `' '`) disables persistent REPL history.

NODE_REPL_EXTERNAL_MODULE=file

Path to a Node.js module which will be loaded in place of the built-in REPL. Overriding this value to an empty string (`' '`) will use the built-in REPL.

NODE_SKIP_PLATFORM_CHECK=value

If `value` equals `'1'`, the check for a supported platform is skipped during Node.js startup. Node.js might not execute correctly. Any issues encountered on unsupported platforms will not be fixed.

NODE_TLS_REJECT_UNAUTHORIZED=value

If `value` equals `'0'`, certificate validation is disabled for TLS connections. This makes TLS, and HTTPS by extension, insecure. The use of this environment variable is strongly discouraged.

NODE_V8_COVERAGE=dir

When set, Node.js will begin outputting [V8 JavaScript code coverage](#) and [Source Map](#) data to the directory provided as an argument (coverage information is written as JSON to files with a `coverage` prefix).

`NODE_V8_COVERAGE` will automatically propagate to subprocesses, making it easier to instrument applications that call the `child_process.spawn()` family of functions. `NODE_V8_COVERAGE` can be set to an empty string, to prevent propagation.

Coverage output

Coverage is output as an array of [ScriptCoverage](#) objects on the top-level key `result`:

```
{
  "result": [
    {
      "scriptId": "67",
      "url": "internal/tty.js",
      "functions": []
    }
  ]
}
```

Source map cache

Stability: 1 - Experimental

If found, source map data is appended to the top-level key `source-map-cache` on the JSON coverage object.

`source-map-cache` is an object with keys representing the files source maps were extracted from, and values which include the raw source-map URL (in the key `url`), the parsed Source Map v3 information (in the key `data`), and the line lengths of the source file (in the key `lineLengths`).

```
{
  "result": [
    {
      "scriptId": "68",
      "url": "file:///absolute/path/to/source.js",
      "functions": []
    }
  ],
  "source-map-cache": {
    "file:///absolute/path/to/source.js": {
      "url": "./path-to-map.json",
      "data": {
        "version": 3,
        "sources": [
          "file:///absolute/path/to/original.js"
        ],
        "names": [
          "Foo",
          "console",
          "info"
        ],
        "mappings": "MAAMA,IACJC,YAAaC",
        "sourceRoot": "./"
      },
      "lineLengths": [
        13,
        62,
        38,
        27
      ]
    }
  }
}
```

NO_COLOR=<any>

`NO_COLOR` is an alias for `NODE_DISABLE_COLORS`. The value of the environment variable is arbitrary.

OPENSSL_CONF=file

Load an OpenSSL configuration file on startup. Among other uses, this can be used to enable FIPS-compliant crypto if Node.js is built with `./configure --openssl-fips`.

If the `--openssl-config` command-line option is used, the environment variable is ignored.

SSL_CERT_DIR=dir

If `--use-openssl-ca` is enabled, this overrides and sets OpenSSL's directory containing trusted certificates.

Be aware that unless the child environment is explicitly set, this environment variable will be inherited by any child processes, and if they use OpenSSL, it may cause them to trust the same CAs as node.

SSL_CERT_FILE=file

If `--use-openssl-ca` is enabled, this overrides and sets OpenSSL's file containing trusted certificates.

Be aware that unless the child environment is explicitly set, this environment variable will be inherited by any child processes, and if they use OpenSSL, it may cause them to trust the same CAs as node.

TZ

The `TZ` environment variable is used to specify the timezone configuration.

While the Node.js support for `TZ` will not handle all of the various [ways that TZ is handled in other environments](#), it will support basic [timezone IDs](#) (such as `'Etc/UTC'`, `'Europe/Paris'` or `'America/New_York'`). It may support a few other abbreviations or aliases, but these are strongly discouraged and not guaranteed.

```
$ TZ=Europe/Dublin node -pe "new Date().toString()"
Wed May 12 2021 20:30:48 GMT+0100 (Irish Standard Time)
```

UV_THREADPOOL_SIZE=size

Set the number of threads used in libuv's threadpool to `size` threads.

Asynchronous system APIs are used by Node.js whenever possible, but where they do not exist, libuv's threadpool is used to create asynchronous node APIs based on synchronous system APIs. Node.js APIs that use the threadpool are:

- all `fs` APIs, other than the file watcher APIs and those that are explicitly synchronous
- asynchronous crypto APIs such as `crypto.pbkdf2()`, `crypto.scrypt()`, `crypto.randomBytes()`, `crypto.randomFill()`, `crypto.generateKeyPair()`
- `dns.lookup()`
- all `zlib` APIs, other than those that are explicitly synchronous

Because libuv's threadpool has a fixed size, it means that if for whatever reason any of these APIs takes a long time, other (seemingly unrelated) APIs that run in libuv's threadpool will experience degraded performance. In order to mitigate this issue, one potential solution is to increase the size of libuv's threadpool by setting the `'UV_THREADPOOL_SIZE'` environment variable to a value greater than `4` (its current default value). For more information, see the [libuv threadpool documentation](#).

Useful V8 options

V8 has its own set of CLI options. Any V8 CLI option that is provided to `node` will be passed on to V8 to handle. V8's options have *no stability guarantee*. The V8 team themselves don't consider them to be part of their formal API, and reserve the right to change them at any time. Likewise, they are not covered by the Node.js stability guarantees. Many of the V8 options are of interest only to V8 developers. Despite this, there is a small set of V8 options that are widely applicable to Node.js, and they are documented here:

--max-old-space-size=SIZE (in megabytes)

Sets the max memory size of V8's old memory section. As memory consumption approaches the limit, V8 will spend more time on garbage collection in an effort to free unused memory.

On a machine with 2 GB of memory, consider setting this to 1536 (1.5 GB) to leave some memory for other uses and avoid swapping.

```
$ node --max-old-space-size=1536 index.js
```

Console

Stability: 2 - Stable

Source Code: [lib/console.js](#)

The `console` module provides a simple debugging console that is similar to the JavaScript console mechanism provided by web browsers.

The module exports two specific components:

- A `Console` class with methods such as `console.log()`, `console.error()` and `console.warn()` that can be used to write to any Node.js stream.
- A global `console` instance configured to write to `process.stdout` and `process.stderr`. The global `console` can be used without calling `require('console')`.

Warning: The global console object's methods are neither consistently synchronous like the browser APIs they resemble, nor are they consistently asynchronous like all other Node.js streams. See the [note on process I/O](#) for more information.

Example using the global `console`:

```
console.log('hello world');
// Prints: hello world, to stdout
console.log('hello %s', 'world');
// Prints: hello world, to stdout
console.error(new Error('whoops, something bad happened'));
// Prints error message and stack trace to stderr:
//   Error: Whoops, something bad happened
//     at [eval]:5:15
//     at Script.runInThisContext (node:vm:132:18)
//     at Object.runInThisContext (node:vm:309:38)
//     at node:internal/process/execution:77:19
//     at [eval]-wrapper:6:22
//     at evalScript (node:internal/process/execution:76:60)
//     at node:internal/main/eval_string:23:3

const name = 'Will Robinson';
console.warn(`Danger ${name}! Danger!`);
// Prints: Danger Will Robinson! Danger!, to stderr
```

Example using the `Console` class:

```

const out = getStreamSomehow();
const err = getStreamSomehow();
const myConsole = new console.Console(out, err);

myConsole.log('hello world');
// Prints: hello world, to out
myConsole.log('hello %s', 'world');
// Prints: hello world, to out
myConsole.error(new Error('Whoops, something bad happened'));
// Prints: [Error: Whoops, something bad happened], to err

const name = 'Will Robinson';
myConsole.warn(`Danger ${name}! Danger!`);
// Prints: Danger Will Robinson! Danger!, to err

```

Class: Console

The `Console` class can be used to create a simple logger with configurable output streams and can be accessed using either `require('console').Console` or `console.Console` (or their destructured counterparts):

```
const { Console } = require('console');
```

```
const { Console } = console;
```

`new Console(stdout[, stderr][, ignoreErrors])`

`new Console(options)`

- `options` `<Object>`
 - `stdout` `<stream.Writable>`
 - `stderr` `<stream.Writable>`
 - `ignoreErrors` `<boolean>` Ignore errors when writing to the underlying streams. **Default:** `true`.
 - `colorMode` `<boolean>` | `<string>` Set color support for this `Console` instance. Setting to `true` enables coloring while inspecting values. Setting to `false` disables coloring while inspecting values. Setting to `'auto'` makes color support depend on the value of the `isTTY` property and the value returned by `getColorDepth()` on the respective stream. This option can not be used, if `inspectOptions.colors` is set as well. **Default:** `'auto'`.
 - `inspectOptions` `<Object>` Specifies options that are passed along to `util.inspect()`.
 - `groupIndentation` `<number>` Set group indentation. **Default:** `2`.

Creates a new `Console` with one or two writable stream instances. `stdout` is a writable stream to print log or info output. `stderr` is used for warning or error output. If `stderr` is not provided, `stdout` is used for `stderr`.

```

const output = fs.createWriteStream('./stdout.log');
const errorOutput = fs.createWriteStream('./stderr.log');
// Custom simple logger
const logger = new Console({ stdout: output, stderr: errorOutput });
// use it like console

```

```
const count = 5;
logger.log('count: %d', count);
// In stdout.log: count 5
```

The global `console` is a special `Console` whose output is sent to `process.stdout` and `process.stderr`. It is equivalent to calling:

```
new Console({ stdout: process.stdout, stderr: process.stderr });
```

console.assert(value[, ...message])

- `value` `<any>` The value tested for being truthy.
- `...message` `<any>` All arguments besides `value` are used as error message.

`console.assert()` writes a message if `value` is `false` or omitted. It only writes a message and does not otherwise affect execution. The output always starts with "Assertion failed". If provided, `message` is formatted using `util.format()`.

If `value` is `truthy`, nothing happens.

```
console.assert(true, 'does nothing');

console.assert(false, 'Whoops %s work', 'didn\'t');
// Assertion failed: Whoops didn't work

console.assert();
// Assertion failed
```

console.clear()

When `stdout` is a TTY, calling `console.clear()` will attempt to clear the TTY. When `stdout` is not a TTY, this method does nothing.

The specific operation of `console.clear()` can vary across operating systems and terminal types. For most Linux operating systems, `console.clear()` operates similarly to the `clear` shell command. On Windows, `console.clear()` will clear only the output in the current terminal viewport for the Node.js binary.

console.count([label])

- `label` `<string>` The display label for the counter. **Default:** `'default'`.

Maintains an internal counter specific to `label` and outputs to `stdout` the number of times `console.count()` has been called with the given `label`.

```
> console.count()
default: 1
undefined
> console.count('default')
default: 2
undefined
> console.count('abc')
abc: 1
undefined
> console.count('xyz')
xyz: 1
```

```

undefined
> console.count('abc')
abc: 2
undefined
> console.count()
default: 3
undefined
>

```

console.countReset([label])

- `label` `<string>` The display label for the counter. **Default:** `'default'`.

Resets the internal counter specific to `label`.

```

> console.count('abc');
abc: 1
undefined
> console.countReset('abc');
undefined
> console.count('abc');
abc: 1
undefined
>

```

console.debug(data[, ...args])

- `data` `<any>`
- `...args` `<any>`

The `console.debug()` function is an alias for `console.log()`.

console.dir(obj[, options])

- `obj` `<any>`
- `options` `<Object>`
 - `showHidden` `<boolean>` If `true` then the object's non-enumerable and symbol properties will be shown too. **Default:** `false`.
 - `depth` `<number>` Tells `util.inspect()` how many times to recurse while formatting the object. This is useful for inspecting large complicated objects. To make it recurse indefinitely, pass `null`. **Default:** `2`.
 - `colors` `<boolean>` If `true`, then the output will be styled with ANSI color codes. Colors are customizable; see [customizing util.inspect\(\) colors](#). **Default:** `false`.

Uses `util.inspect()` on `obj` and prints the resulting string to `stdout`. This function bypasses any custom `inspect()` function defined on `obj`.

console.dirxml(...data)

- `...data` `<any>`

This method calls `console.log()` passing it the arguments received. This method does not produce any XML formatting.

console.error([data][, ...args])

- `data` `<any>`
- `...args` `<any>`

Prints to `stderr` with newline. Multiple arguments can be passed, with the first used as the primary message and all additional used as substitution values similar to `printf(3)` (the arguments are all passed to `util.format()`).

```
const code = 5;
console.error('error #%d', code);
// Prints: error #5, to stderr
console.error('error', code);
// Prints: error 5, to stderr
```

If formatting elements (e.g. `%d`) are not found in the first string then `util.inspect()` is called on each argument and the resulting string values are concatenated. See `util.format()` for more information.

`console.group([...label])`

- `...label` `<any>`

Increases indentation of subsequent lines by spaces for `groupIndentation` length.

If one or more `label`s are provided, those are printed first without the additional indentation.

`console.groupCollapsed()`

An alias for `console.group()`.

`console.groupEnd()`

Decreases indentation of subsequent lines by spaces for `groupIndentation` length.

`console.info([data][, ...args])`

- `data` `<any>`
- `...args` `<any>`

The `console.info()` function is an alias for `console.log()`.

`console.log([data][, ...args])`

- `data` `<any>`
- `...args` `<any>`

Prints to `stdout` with newline. Multiple arguments can be passed, with the first used as the primary message and all additional used as substitution values similar to `printf(3)` (the arguments are all passed to `util.format()`).

```
const count = 5;
console.log('count: %d', count);
// Prints: count: 5, to stdout
console.log('count:', count);
// Prints: count: 5, to stdout
```

See `util.format()` for more information.

console.table(tabularData[, properties])

- `tabularData` `<any>`
- `properties` `<string[]>` Alternate properties for constructing the table.

Try to construct a table with the columns of the properties of `tabularData` (or use `properties`) and rows of `tabularData` and log it. Falls back to just logging the argument if it can't be parsed as tabular.

```
// These can't be parsed as tabular data
console.table(Symbol());
// Symbol()

console.table(undefined);
// undefined

console.table([
  { a: 1, b: 'Y' },
  { a: 'Z', b: 2 }
]);
//
// | (index) | a | b |
// |-----|---|---|
// | 0      | 1 | 'Y'|
// | 1      | 'Z'| 2  |
// |-----|---|---|

console.table([
  { a: 1, b: 'Y' },
  { a: 'Z', b: 2 }
], ['a']);
//
// | (index) | a |
// |-----|---|
// | 0      | 1 |
// | 1      | 'Z'|
// |-----|---|
```

console.time([label])

- `label` `<string>` Default: `'default'`

Starts a timer that can be used to compute the duration of an operation. Timers are identified by a unique `label`. Use the same `label` when calling `console.timeEnd()` to stop the timer and output the elapsed time in suitable time units to `stdout`. For example, if the elapsed time is 3869ms, `console.timeEnd()` displays "3.869s".

console.timeEnd([label])

- `label` `<string>` Default: `'default'`

Stops a timer that was previously started by calling `console.time()` and prints the result to `stdout`:

```
console.time('100-elements');
for (let i = 0; i < 100; i++) {}
console.timeEnd('100-elements');
// prints 100-elements: 225.438ms
```

console.timeLog([label][, ...data])

- `label` `<string>` Default: `'default'`

- `...data` `<any>`

For a timer that was previously started by calling `console.time()`, prints the elapsed time and other `data` arguments to `stdout`:

```
console.time('process');
const value = expensiveProcess1(); // Returns 42
console.timeLog('process', value);
// Prints "process: 365.227ms 42".
doExpensiveProcess2(value);
console.timeEnd('process');
```

`console.trace([message][, ...args])`

- `message` `<any>`
- `...args` `<any>`

Prints to `stderr` the string `'Trace: '`, followed by the `util.format()` formatted message and stack trace to the current position in the code.

```
console.trace('Show me');
// Prints: (stack trace will vary based on where trace is called)
// Trace: Show me
//   at repl:2:9
//   at REPLServer.defaultEval (repl.js:248:27)
//   at bound (domain.js:287:14)
//   at REPLServer.runBound [as eval] (domain.js:300:12)
//   at REPLServer.<anonymous> (repl.js:412:12)
//   at emitOne (events.js:82:20)
//   at REPLServer.emit (events.js:169:7)
//   at REPLServer.Interface._onLine (readline.js:210:10)
//   at REPLServer.Interface._line (readline.js:549:8)
//   at REPLServer.Interface._ttyWrite (readline.js:826:14)
```

`console.warn([data][, ...args])`

- `data` `<any>`
- `...args` `<any>`

The `console.warn()` function is an alias for `console.error()`.

Inspector only methods

The following methods are exposed by the V8 engine in the general API but do not display anything unless used in conjunction with the `inspector` (`--inspect` flag).

`console.profile([label])`

- `label` `<string>`

This method does not display anything unless used in the inspector. The `console.profile()` method starts a JavaScript CPU profile with an optional label until `console.profileEnd()` is called. The profile is then added to the **Profile** panel of the inspector.

```

console.profile('MyLabel');
// Some code
console.profileEnd('MyLabel');
// Adds the profile 'MyLabel' to the Profiles panel of the inspector.

```

console.profileEnd([label])

- `label` `<string>`

This method does not display anything unless used in the inspector. Stops the current JavaScript CPU profiling session if one has been started and prints the report to the **Profiles** panel of the inspector. See `console.profile()` for an example.

If this method is called without a label, the most recently started profile is stopped.

console.timeStamp([label])

- `label` `<string>`

This method does not display anything unless used in the inspector. The `console.timeStamp()` method adds an event with the label `'label'` to the **Timeline** panel of the inspector.

Crypto

Stability: 2 - Stable

Source Code: [lib/crypto.js](#)

The `crypto` module provides cryptographic functionality that includes a set of wrappers for OpenSSL's hash, HMAC, cipher, decipher, sign, and verify functions.

```

const { createHmac } = await import('crypto');

const secret = 'abcdefg';
const hash = createHmac('sha256', secret)
  .update('I love cupcakes')
  .digest('hex');

console.log(hash);
// Prints:
//   c0fa1bc00531bd78ef38c628449c5102aeabd49b5dc3a2a516ea6ea959d6658e

const crypto = require('crypto');

const secret = 'abcdefg';
const hash = crypto.createHmac('sha256', secret)
  .update('I love cupcakes')
  .digest('hex');

console.log(hash);
// Prints:
//   c0fa1bc00531bd78ef38c628449c5102aeabd49b5dc3a2a516ea6ea959d6658e

```

Determining if crypto support is unavailable

It is possible for Node.js to be built without including support for the `crypto` module. In such cases, attempting to `import` from `crypto` or calling `require('crypto')` will result in an error being thrown.

When using CommonJS, the error thrown can be caught using `try/catch`:

```
let crypto;
try {
  crypto = require('crypto');
} catch (err) {
  console.log('crypto support is disabled!');
}
```

When using the lexical ESM `import` keyword, the error can only be caught if a handler for `process.on('uncaughtException')` is registered *before* any attempt to load the module is made -- using, for instance, a preload module.

When using ESM, if there is a chance that the code may be run on a build of Node.js where `crypto` support is not enabled, consider using the `import()` function instead of the lexical `import` keyword:

```
let crypto;
try {
  crypto = await import('crypto');
} catch (err) {
  console.log('crypto support is disabled!');
}
```

Class: `Certificate`

SPKAC is a Certificate Signing Request mechanism originally implemented by Netscape and was specified formally as part of [HTML5's keygen element](#).

`<keygen>` is deprecated since [HTML 5.2](#) and new projects should not use this element anymore.

The `crypto` module provides the `Certificate` class for working with SPKAC data. The most common usage is handling output generated by the HTML5 `<keygen>` element. Node.js uses [OpenSSL's SPKAC implementation](#) internally.

Static method: `Certificate.exportChallenge(spkac[, encoding])`

- `spkac` `<string>` | `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>`
- `encoding` `<string>` The `encoding` of the `spkac` string.
- Returns: `<Buffer>` The challenge component of the `spkac` data structure, which includes a public key and a challenge.

```
const { Certificate } = await import('crypto');
const spkac = getSpkacSomehow();
const challenge = Certificate.exportChallenge(spkac);
console.log(challenge.toString('utf8'));
// Prints: the challenge as a UTF8 string
const { Certificate } = require('crypto');
const spkac = getSpkacSomehow();
const challenge = Certificate.exportChallenge(spkac);
console.log(challenge.toString('utf8'));
// Prints: the challenge as a UTF8 string
```

Static method: `Certificate.exportPublicKey(sp kac[, encoding])`

- `sp kac` `<string>` | `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>`
- `encoding` `<string>` The `encoding` of the `sp kac` string.
- Returns: `<Buffer>` The public key component of the `sp kac` data structure, which includes a public key and a challenge.

```
const { Certificate } = await import('crypto');
const sp kac = getSp kacSomehow();
const publicKey = Certificate.exportPublicKey(sp kac);
console.log(publicKey);
// Prints: the public key as <Buffer ...>const { Certificate } = require('crypto');
const sp kac = getSp kacSomehow();
const publicKey = Certificate.exportPublicKey(sp kac);
console.log(publicKey);
// Prints: the public key as <Buffer ...>
```

Static method: `Certificate.verifySp kac(sp kac[, encoding])`

- `sp kac` `<string>` | `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>`
- `encoding` `<string>` The `encoding` of the `sp kac` string.
- Returns: `<boolean>` `true` if the given `sp kac` data structure is valid, `false` otherwise.

```
import { Buffer } from 'buffer';
const { Certificate } = await import('crypto');

const sp kac = getSp kacSomehow();
console.log(Certificate.verifySp kac(Buffer.from(sp kac)));
// Prints: true or falseconst { Certificate } = require('crypto');
const { Buffer } = require('buffer');

const sp kac = getSp kacSomehow();
console.log(Certificate.verifySp kac(Buffer.from(sp kac)));
// Prints: true or false
```

Legacy API

Stability: 0 - Deprecated

As a legacy interface, it is possible to create new instances of the `crypto.Certificate` class as illustrated in the examples below.

`new crypto.Certificate()`

Instances of the `Certificate` class can be created using the `new` keyword or by calling `crypto.Certificate()` as a function:

```
const { Certificate } = await import('crypto');

const cert1 = new Certificate();
const cert2 = Certificate();const { Certificate } = require('crypto');
```

```
const cert1 = new Certificate();
const cert2 = Certificate();
```

certificate.exportChallenge(sp kac[, encoding])

- **sp kac** `<string> | <ArrayBuffer> | <Buffer> | <TypedArray> | <DataView>`
- **encoding** `<string>` The **encoding** of the **sp kac** string.
- Returns: `<Buffer>` The challenge component of the **sp kac** data structure, which includes a public key and a challenge.

```
const { Certificate } = await import('crypto');
const cert = Certificate();
const sp kac = getSp kacSomehow();
const challenge = cert.exportChallenge(sp kac);
console.log(challenge.toString('utf8'));
// Prints: the challenge as a UTF8 string
const { Certificate } = require('crypto');
const cert = Certificate();
const sp kac = getSp kacSomehow();
const challenge = cert.exportChallenge(sp kac);
console.log(challenge.toString('utf8'));
// Prints: the challenge as a UTF8 string
```

certificate.exportPublicKey(sp kac[, encoding])

- **sp kac** `<string> | <ArrayBuffer> | <Buffer> | <TypedArray> | <DataView>`
- **encoding** `<string>` The **encoding** of the **sp kac** string.
- Returns: `<Buffer>` The public key component of the **sp kac** data structure, which includes a public key and a challenge.

```
const { Certificate } = await import('crypto');
const cert = Certificate();
const sp kac = getSp kacSomehow();
const publicKey = cert.exportPublicKey(sp kac);
console.log(publicKey);
// Prints: the public key as <Buffer ...>
const { Certificate } = require('crypto');
const cert = Certificate();
const sp kac = getSp kacSomehow();
const publicKey = cert.exportPublicKey(sp kac);
console.log(publicKey);
// Prints: the public key as <Buffer ...>
```

certificate.verifySp kac(sp kac[, encoding])

- **sp kac** `<string> | <ArrayBuffer> | <Buffer> | <TypedArray> | <DataView>`
- **encoding** `<string>` The **encoding** of the **sp kac** string.
- Returns: `<boolean>` **true** if the given **sp kac** data structure is valid, **false** otherwise.

```
import { Buffer } from 'buffer';
const { Certificate } = await import('crypto');
```

```

const cert = Certificate();
const spkac = getSpkacSomehow();
console.log(cert.verifySpkac(Buffer.from(spkac)));
// Prints: true or false
const { Certificate } = require('crypto');
const { Buffer } = require('buffer');

const cert = Certificate();
const spkac = getSpkacSomehow();
console.log(cert.verifySpkac(Buffer.from(spkac)));
// Prints: true or false

```

Class: Cipher

- Extends: `<stream.Transform>`

Instances of the `Cipher` class are used to encrypt data. The class can be used in one of two ways:

- As a `stream` that is both readable and writable, where plain unencrypted data is written to produce encrypted data on the readable side, or
- Using the `cipher.update()` and `cipher.final()` methods to produce the encrypted data.

The `crypto.createCipher()` or `crypto.createCipheriv()` methods are used to create `Cipher` instances. `Cipher` objects are not to be created directly using the `new` keyword.

Example: Using `Cipher` objects as streams:

```

const {
  script,
  randomFill,
  createCipheriv
} = await import('crypto');

const algorithm = 'aes-192-cbc';
const password = 'Password used to generate key';

// First, we'll generate the key. The key length is dependent on the algorithm.
// In this case for aes192, it is 24 bytes (192 bits).
script(password, 'salt', 24, (err, key) => {
  if (err) throw err;
  // Then, we'll generate a random initialization vector
  randomFill(new Uint8Array(16), (err, iv) => {
    if (err) throw err;

    // Once we have the key and iv, we can create and use the cipher...
    const cipher = createCipheriv(algorithm, key, iv);

    let encrypted = '';
    cipher.setEncoding('hex');

    cipher.on('data', (chunk) => encrypted += chunk);
    cipher.on('end', () => console.log(encrypted));
  });
});

```



```

    cipher.write('some clear text data');
    cipher.end();
  });
});const {
  script,
  randomFill,
  createCipheriv
} = require('crypto');

const algorithm = 'aes-192-cbc';
const password = 'Password used to generate key';

// First, we'll generate the key. The key length is dependent on the algorithm.
// In this case for aes192, it is 24 bytes (192 bits).
script(password, 'salt', 24, (err, key) => {
  if (err) throw err;
  // Then, we'll generate a random initialization vector
  randomFill(new Uint8Array(16), (err, iv) => {
    if (err) throw err;

    // Once we have the key and iv, we can create and use the cipher...
    const cipher = createCipheriv(algorithm, key, iv);

    let encrypted = '';
    cipher.setEncoding('hex');

    cipher.on('data', (chunk) => encrypted += chunk);
    cipher.on('end', () => console.log(encrypted));

    cipher.write('some clear text data');
    cipher.end();
  });
});

```

Example: Using `Cipher` and piped streams:

```

import {
  createReadStream,
  createWriteStream,
} from 'fs';

import {
  pipeline
} from 'stream';

const {
  script,
  randomFill,
  createCipheriv

```

```

} = await import('crypto');

const algorithm = 'aes-192-cbc';
const password = 'Password used to generate key';

// First, we'll generate the key. The key length is dependent on the algorithm.
// In this case for aes192, it is 24 bytes (192 bits).
scrypt(password, 'salt', 24, (err, key) => {
  if (err) throw err;
  // Then, we'll generate a random initialization vector
  randomFill(new Uint8Array(16), (err, iv) => {
    if (err) throw err;

    const cipher = createCipheriv(algorithm, key, iv);

    const input = createReadStream('test.js');
    const output = createWriteStream('test.enc');

    pipeline(input, cipher, output, (err) => {
      if (err) throw err;
    });
  });
});const {
  createReadStream,
  createWriteStream,
} = require('fs');

const {
  pipeline
} = require('stream');

const {
  scrypt,
  randomFill,
  createCipheriv,
} = require('crypto');

const algorithm = 'aes-192-cbc';
const password = 'Password used to generate key';

// First, we'll generate the key. The key length is dependent on the algorithm.
// In this case for aes192, it is 24 bytes (192 bits).
scrypt(password, 'salt', 24, (err, key) => {
  if (err) throw err;
  // Then, we'll generate a random initialization vector
  randomFill(new Uint8Array(16), (err, iv) => {
    if (err) throw err;

    const cipher = createCipheriv(algorithm, key, iv);

    const input = createReadStream('test.js');

```

```

const output = createWriteStream('test.enc');

pipeline(input, cipher, output, (err) => {
  if (err) throw err;
});
});
});

```

Example: Using the `cipher.update()` and `cipher.final()` methods:

```

const {
  script,
  randomFill,
  createCipheriv
} = await import('crypto');

const algorithm = 'aes-192-cbc';
const password = 'Password used to generate key';

// First, we'll generate the key. The key length is dependent on the algorithm.
// In this case for aes192, it is 24 bytes (192 bits).
script(password, 'salt', 24, (err, key) => {
  if (err) throw err;
  // Then, we'll generate a random initialization vector
  randomFill(new Uint8Array(16), (err, iv) => {
    if (err) throw err;

    const cipher = createCipheriv(algorithm, key, iv);

    let encrypted = cipher.update('some clear text data', 'utf8', 'hex');
    encrypted += cipher.final('hex');
    console.log(encrypted);
  });
});const {
  script,
  randomFill,
  createCipheriv,
} = require('crypto');

const algorithm = 'aes-192-cbc';
const password = 'Password used to generate key';

// First, we'll generate the key. The key length is dependent on the algorithm.
// In this case for aes192, it is 24 bytes (192 bits).
script(password, 'salt', 24, (err, key) => {
  if (err) throw err;
  // Then, we'll generate a random initialization vector
  randomFill(new Uint8Array(16), (err, iv) => {
    if (err) throw err;

```

```
const cipher = createCipheriv(algorithm, key, iv);

let encrypted = cipher.update('some clear text data', 'utf8', 'hex');
encrypted += cipher.final('hex');
console.log(encrypted);
});
});
```

`cipher.final([outputEncoding])`

- `outputEncoding` `<string>` The `encoding` of the return value.
- Returns: `<Buffer>` | `<string>` Any remaining enciphered contents. If `outputEncoding` is specified, a string is returned. If an `outputEncoding` is not provided, a `Buffer` is returned.

Once the `cipher.final()` method has been called, the `Cipher` object can no longer be used to encrypt data. Attempts to call `cipher.final()` more than once will result in an error being thrown.

`cipher.getAuthTag()`

- Returns: `<Buffer>` When using an authenticated encryption mode (`GCM`, `CCM` and `OCB` are currently supported), the `cipher.getAuthTag()` method returns a `Buffer` containing the *authentication tag* that has been computed from the given data.

The `cipher.getAuthTag()` method should only be called after encryption has been completed using the `cipher.final()` method.

`cipher.setAAD(buffer[, options])`

- `buffer` `<string>` | `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>`
- `options` `<Object>` `stream.transform` options
 - `plaintextLength` `<number>`
 - `encoding` `<string>` The string encoding to use when `buffer` is a string.
- Returns: `<Cipher>` for method chaining.

When using an authenticated encryption mode (`GCM`, `CCM` and `OCB` are currently supported), the `cipher.setAAD()` method sets the value used for the *additional authenticated data* (AAD) input parameter.

The `plaintextLength` option is optional for `GCM` and `OCB`. When using `CCM`, the `plaintextLength` option must be specified and its value must match the length of the plaintext in bytes. See [CCM mode](#).

The `cipher.setAAD()` method must be called before `cipher.update()`.

`cipher.setAutoPadding([autoPadding])`

- `autoPadding` `<boolean>` **Default:** `true`
- Returns: `<Cipher>` for method chaining.

When using block encryption algorithms, the `Cipher` class will automatically add padding to the input data to the appropriate block size. To disable the default padding call `cipher.setAutoPadding(false)`.

When `autoPadding` is `false`, the length of the entire input data must be a multiple of the cipher's block size or `cipher.final()` will throw an error. Disabling automatic padding is useful for non-standard padding, for instance using `0x0` instead of PKCS padding.

The `cipher.setAutoPadding()` method must be called before `cipher.final()`.

`cipher.update(data[, inputEncoding][, outputEncoding])`

- `data` `<string>` | `<Buffer>` | `<TypedArray>` | `<DataView>`
- `inputEncoding` `<string>` The `encoding` of the data.
- `outputEncoding` `<string>` The `encoding` of the return value.
- Returns: `<Buffer>` | `<string>`

Updates the cipher with `data`. If the `inputEncoding` argument is given, the `data` argument is a string using the specified encoding. If the `inputEncoding` argument is not given, `data` must be a `Buffer`, `TypedArray`, or `DataView`. If `data` is a `Buffer`, `TypedArray`, or `DataView`, then `inputEncoding` is ignored.

The `outputEncoding` specifies the output format of the enciphered data. If the `outputEncoding` is specified, a string using the specified encoding is returned. If no `outputEncoding` is provided, a `Buffer` is returned.

The `cipher.update()` method can be called multiple times with new data until `cipher.final()` is called. Calling `cipher.update()` after `cipher.final()` will result in an error being thrown.

Class: Decipher

- Extends: `<stream.Transform>`

Instances of the `Decipher` class are used to decrypt data. The class can be used in one of two ways:

- As a `stream` that is both readable and writable, where plain encrypted data is written to produce unencrypted data on the readable side, or
- Using the `decipher.update()` and `decipher.final()` methods to produce the unencrypted data.

The `crypto.createDecipher()` or `crypto.createDecipheriv()` methods are used to create `Decipher` instances. `Decipher` objects are not to be created directly using the `new` keyword.

Example: Using `Decipher` objects as streams:

```
import { Buffer } from 'buffer';
const {
  scriptSync,
  createDecipheriv
} = await import('crypto');

const algorithm = 'aes-192-cbc';
const password = 'Password used to generate key';
// Key length is dependent on the algorithm. In this case for aes192, it is
// 24 bytes (192 bits).
// Use the async `crypto.script()` instead.
const key = scriptSync(password, 'salt', 24);
// The IV is usually passed along with the ciphertext.
const iv = Buffer.alloc(16, 0); // Initialization vector.

const decipher = createDecipheriv(algorithm, key, iv);

let decrypted = '';
decipher.on('readable', () => {
  while (null !== (chunk = decipher.read())) {
    decrypted += chunk.toString('utf8');
  }
});
```

```

decipher.on('end', () => {
  console.log(decrypted);
  // Prints: some clear text data
});

// Encrypted with same algorithm, key and iv.
const encrypted =
  'e5f79c5915c02171eec6b212d5520d44480993d7d622a7c4c2da32f6efda0ffa';
decipher.write(encrypted, 'hex');
decipher.end();const {
  scriptSync,
  createDecipheriv,
} = require('crypto');
const { Buffer } = require('buffer');

const algorithm = 'aes-192-cbc';
const password = 'Password used to generate key';
// Key length is dependent on the algorithm. In this case for aes192, it is
// 24 bytes (192 bits).
// Use the async `crypto.decrypt()` instead.
const key = scriptSync(password, 'salt', 24);
// The IV is usually passed along with the ciphertext.
const iv = Buffer.alloc(16, 0); // Initialization vector.

const decipher = createDecipheriv(algorithm, key, iv);

let decrypted = '';
decipher.on('readable', () => {
  while (null !== (chunk = decipher.read())) {
    decrypted += chunk.toString('utf8');
  }
});
decipher.on('end', () => {
  console.log(decrypted);
  // Prints: some clear text data
});

// Encrypted with same algorithm, key and iv.
const encrypted =
  'e5f79c5915c02171eec6b212d5520d44480993d7d622a7c4c2da32f6efda0ffa';
decipher.write(encrypted, 'hex');
decipher.end();

```

Example: Using `Decipher` and piped streams:

```

import {
  createReadStream,
  createWriteStream,
} from 'fs';
import { Buffer } from 'buffer';

```

```

const {
  scryptSync,
  createDecipheriv
} = await import('crypto');

const algorithm = 'aes-192-cbc';
const password = 'Password used to generate key';
// Use the async `crypto.scrypt()` instead.
const key = scryptSync(password, 'salt', 24);
// The IV is usually passed along with the ciphertext.
const iv = Buffer.alloc(16, 0); // Initialization vector.

const decipher = createDecipheriv(algorithm, key, iv);

const input = createReadStream('test.enc');
const output = createWriteStream('test.js');

input.pipe(decipher).pipe(output);const {
  createReadStream,
  createWriteStream,
} = require('fs');
const {
  scryptSync,
  createDecipheriv,
} = require('crypto');
const { Buffer } = require('buffer');

const algorithm = 'aes-192-cbc';
const password = 'Password used to generate key';
// Use the async `crypto.scrypt()` instead.
const key = scryptSync(password, 'salt', 24);
// The IV is usually passed along with the ciphertext.
const iv = Buffer.alloc(16, 0); // Initialization vector.

const decipher = createDecipheriv(algorithm, key, iv);

const input = createReadStream('test.enc');
const output = createWriteStream('test.js');

input.pipe(decipher).pipe(output);

```

Example: Using the `decipher.update()` and `decipher.final()` methods:

```

import { Buffer } from 'buffer';
const {
  scryptSync,
  createDecipheriv
} = await import('crypto');

const algorithm = 'aes-192-cbc';

```

```

const password = 'Password used to generate key';
// Use the async `crypto.scrypt()` instead.
const key = scryptSync(password, 'salt', 24);
// The IV is usually passed along with the ciphertext.
const iv = Buffer.alloc(16, 0); // Initialization vector.

const decipher = createDecipheriv(algorithm, key, iv);

// Encrypted using same algorithm, key and iv.
const encrypted =
  'e5f79c5915c02171eec6b212d5520d44480993d7d622a7c4c2da32f6efda0ffa';
let decrypted = decipher.update(encrypted, 'hex', 'utf8');
decrypted += decipher.final('utf8');
console.log(decrypted);
// Prints: some clear text data
const {
  scryptSync,
  createDecipheriv,
} = require('crypto');
const { Buffer } = require('buffer');

const algorithm = 'aes-192-cbc';
const password = 'Password used to generate key';
// Use the async `crypto.scrypt()` instead.
const key = scryptSync(password, 'salt', 24);
// The IV is usually passed along with the ciphertext.
const iv = Buffer.alloc(16, 0); // Initialization vector.

const decipher = createDecipheriv(algorithm, key, iv);

// Encrypted using same algorithm, key and iv.
const encrypted =
  'e5f79c5915c02171eec6b212d5520d44480993d7d622a7c4c2da32f6efda0ffa';
let decrypted = decipher.update(encrypted, 'hex', 'utf8');
decrypted += decipher.final('utf8');
console.log(decrypted);
// Prints: some clear text data

```

decipher.final([outputEncoding])

- `outputEncoding` `<string>` The `encoding` of the return value.
- Returns: `<Buffer>` | `<string>` Any remaining deciphered contents. If `outputEncoding` is specified, a string is returned. If an `outputEncoding` is not provided, a `Buffer` is returned.

Once the `decipher.final()` method has been called, the `Decipher` object can no longer be used to decrypt data. Attempts to call `decipher.final()` more than once will result in an error being thrown.

decipher.setAAD(buffer[, options])

- `buffer` `<string>` | `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>`
- `options` `<Object>` `stream.transform` options
 - `plaintextLength` `<number>`

- `encoding` `<string>` String encoding to use when `buffer` is a string.
- Returns: `<Decipher>` for method chaining.

When using an authenticated encryption mode (`GCM` , `CCM` and `OCB` are currently supported), the `decipher.setAAD()` method sets the value used for the *additional authenticated data* (AAD) input parameter.

The `options` argument is optional for `GCM`. When using `CCM`, the `plaintextLength` option must be specified and its value must match the length of the ciphertext in bytes. See [CCM mode](#) .

The `decipher.setAAD()` method must be called before `decipher.update()` .

When passing a string as the `buffer` , please consider [caveats when using strings as inputs to cryptographic APIs](#) .

decipher.setAuthTag(buffer[, encoding])

- `buffer` `<string>` | `<Buffer>` | `<ArrayBuffer>` | `<TypedArray>` | `<DataView>`
- `encoding` `<string>` String encoding to use when `buffer` is a string.
- Returns: `<Decipher>` for method chaining.

When using an authenticated encryption mode (`GCM` , `CCM` and `OCB` are currently supported), the `decipher.setAuthTag()` method is used to pass in the received *authentication tag*. If no tag is provided, or if the cipher text has been tampered with, `decipher.final()` will throw, indicating that the cipher text should be discarded due to failed authentication. If the tag length is invalid according to [NIST SP 800-38D](#) or does not match the value of the `authTagLength` option, `decipher.setAuthTag()` will throw an error.

The `decipher.setAuthTag()` method must be called before `decipher.update()` for `CCM` mode or before `decipher.final()` for `GCM` and `OCB` modes. `decipher.setAuthTag()` can only be called once.

When passing a string as the authentication tag, please consider [caveats when using strings as inputs to cryptographic APIs](#) .

decipher.setAutoPadding([autoPadding])

- `autoPadding` `<boolean>` **Default:** `true`
- Returns: `<Decipher>` for method chaining.

When data has been encrypted without standard block padding, calling `decipher.setAutoPadding(false)` will disable automatic padding to prevent `decipher.final()` from checking for and removing padding.

Turning auto padding off will only work if the input data's length is a multiple of the ciphers block size.

The `decipher.setAutoPadding()` method must be called before `decipher.final()` .

decipher.update(data[, inputEncoding][, outputEncoding])

- `data` `<string>` | `<Buffer>` | `<TypedArray>` | `<DataView>`
- `inputEncoding` `<string>` The *encoding* of the `data` string.
- `outputEncoding` `<string>` The *encoding* of the return value.
- Returns: `<Buffer>` | `<string>`

Updates the decipher with `data` . If the `inputEncoding` argument is given, the `data` argument is a string using the specified encoding. If the `inputEncoding` argument is not given, `data` must be a `Buffer` . If `data` is a `Buffer` then `inputEncoding` is ignored.

The `outputEncoding` specifies the output format of the enciphered data. If the `outputEncoding` is specified, a string using the specified encoding is returned. If no `outputEncoding` is provided, a `Buffer` is returned.

The `decipher.update()` method can be called multiple times with new data until `decipher.final()` is called. Calling `decipher.update()` after `decipher.final()` will result in an error being thrown.

Class: DiffieHellman

The `DiffieHellman` class is a utility for creating Diffie-Hellman key exchanges.

Instances of the `DiffieHellman` class can be created using the `crypto.createDiffieHellman()` function.

```
import assert from 'assert';

const {
  createDiffieHellman
} = await import('crypto');

// Generate Alice's keys...
const alice = createDiffieHellman(2048);
const aliceKey = alice.generateKeys();

// Generate Bob's keys...
const bob = createDiffieHellman(alice.getPrime(), alice.getGenerator());
const bobKey = bob.generateKeys();

// Exchange and generate the secret...
const aliceSecret = alice.computeSecret(bobKey);
const bobSecret = bob.computeSecret(aliceKey);

// OK
assert.strictEqual(aliceSecret.toString('hex'), bobSecret.toString('hex'));const assert = require('assert');

const {
  createDiffieHellman,
} = require('crypto');

// Generate Alice's keys...
const alice = createDiffieHellman(2048);
const aliceKey = alice.generateKeys();

// Generate Bob's keys...
const bob = createDiffieHellman(alice.getPrime(), alice.getGenerator());
const bobKey = bob.generateKeys();

// Exchange and generate the secret...
const aliceSecret = alice.computeSecret(bobKey);
const bobSecret = bob.computeSecret(aliceKey);

// OK
assert.strictEqual(aliceSecret.toString('hex'), bobSecret.toString('hex'));
```

`diffieHellman.computeSecret(otherPublicKey[, inputEncoding][, outputEncoding])`

- `otherPublicKey` `<string>` | `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>`
- `inputEncoding` `<string>` The `encoding` of an `otherPublicKey` string.

- `outputEncoding` `<string>` The `encoding` of the return value.
- Returns: `<Buffer>` | `<string>`

Computes the shared secret using `otherPublicKey` as the other party's public key and returns the computed shared secret. The supplied key is interpreted using the specified `inputEncoding`, and secret is encoded using specified `outputEncoding`. If the `inputEncoding` is not provided, `otherPublicKey` is expected to be a `Buffer`, `TypedArray`, or `DataView`.

If `outputEncoding` is given a string is returned; otherwise, a `Buffer` is returned.

`diffieHellman.generateKeys([encoding])`

- `encoding` `<string>` The `encoding` of the return value.
- Returns: `<Buffer>` | `<string>`

Generates private and public Diffie-Hellman key values, and returns the public key in the specified `encoding`. This key should be transferred to the other party. If `encoding` is provided a string is returned; otherwise a `Buffer` is returned.

`diffieHellman.getGenerator([encoding])`

- `encoding` `<string>` The `encoding` of the return value.
- Returns: `<Buffer>` | `<string>`

Returns the Diffie-Hellman generator in the specified `encoding`. If `encoding` is provided a string is returned; otherwise a `Buffer` is returned.

`diffieHellman.getPrime([encoding])`

- `encoding` `<string>` The `encoding` of the return value.
- Returns: `<Buffer>` | `<string>`

Returns the Diffie-Hellman prime in the specified `encoding`. If `encoding` is provided a string is returned; otherwise a `Buffer` is returned.

`diffieHellman.getPrivateKey([encoding])`

- `encoding` `<string>` The `encoding` of the return value.
- Returns: `<Buffer>` | `<string>`

Returns the Diffie-Hellman private key in the specified `encoding`. If `encoding` is provided a string is returned; otherwise a `Buffer` is returned.

`diffieHellman.getPublicKey([encoding])`

- `encoding` `<string>` The `encoding` of the return value.
- Returns: `<Buffer>` | `<string>`

Returns the Diffie-Hellman public key in the specified `encoding`. If `encoding` is provided a string is returned; otherwise a `Buffer` is returned.

`diffieHellman.setPrivateKey(privateKey[, encoding])`

- `privateKey` `<string>` | `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>`
- `encoding` `<string>` The `encoding` of the `privateKey` string.

Sets the Diffie-Hellman private key. If the `encoding` argument is provided, `privateKey` is expected to be a string. If no `encoding` is provided, `privateKey` is expected to be a `Buffer`, `TypedArray`, or `DataView`.

`diffieHellman.setPublicKey(publicKey[, encoding])`

- `publicKey` `<string>` | `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>`
- `encoding` `<string>` The `encoding` of the `publicKey` string.

Sets the Diffie-Hellman public key. If the `encoding` argument is provided, `publicKey` is expected to be a string. If no `encoding` is provided, `publicKey` is expected to be a `Buffer`, `TypedArray`, or `DataView`.

`diffieHellman.verifyError`

A bit field containing any warnings and/or errors resulting from a check performed during initialization of the `DiffieHellman` object.

The following values are valid for this property (as defined in `constants` module):

- `DH_CHECK_P_NOT_SAFE_PRIME`
- `DH_CHECK_P_NOT_PRIME`
- `DH_UNABLE_TO_CHECK_GENERATOR`
- `DH_NOT_SUITABLE_GENERATOR`

Class: `DiffieHellmanGroup`

The `DiffieHellmanGroup` class takes a well-known modp group as its argument. It works the same as `DiffieHellman`, except that it does not allow changing its keys after creation. In other words, it does not implement `setPublicKey()` or `setPrivateKey()` methods.

```
const { createDiffieHellmanGroup } = await import('crypto');
const dh = createDiffieHellmanGroup('modp1');const { createDiffieHellmanGroup } = require('crypto');
const dh = createDiffieHellmanGroup('modp1');
```

The name (e.g. `'modp1'`) is taken from [RFC 2412](#) (modp1 and 2) and [RFC 3526](#):

```
$ perl -ne 'print "$1\n" if /^(modp\d+)/' src/node_crypto_groups.h
modp1 # 768 bits
modp2 # 1024 bits
modp5 # 1536 bits
modp14 # 2048 bits
modp15 # etc.
modp16
modp17
modp18
```

Class: `ECDH`

The `ECDH` class is a utility for creating Elliptic Curve Diffie-Hellman (ECDH) key exchanges.

Instances of the `ECDH` class can be created using the `crypto.createECDH()` function.

```
import assert from 'assert';

const {
  createECDH
} = await import('crypto');
```

```

// Generate Alice's keys...
const alice = createECDH('secp521r1');
const aliceKey = alice.generateKeys();

// Generate Bob's keys...
const bob = createECDH('secp521r1');
const bobKey = bob.generateKeys();

// Exchange and generate the secret...
const aliceSecret = alice.computeSecret(bobKey);
const bobSecret = bob.computeSecret(aliceKey);

assert.strictEqual(aliceSecret.toString('hex'), bobSecret.toString('hex'));
// OK
const assert = require('assert');

const {
  createECDH,
} = require('crypto');

// Generate Alice's keys...
const alice = createECDH('secp521r1');
const aliceKey = alice.generateKeys();

// Generate Bob's keys...
const bob = createECDH('secp521r1');
const bobKey = bob.generateKeys();

// Exchange and generate the secret...
const aliceSecret = alice.computeSecret(bobKey);
const bobSecret = bob.computeSecret(aliceKey);

assert.strictEqual(aliceSecret.toString('hex'), bobSecret.toString('hex'));
// OK

```

Static method: `ECDH.convertKey(key, curve[, inputEncoding[, outputEncoding[, format]])`

- `key` `<string> | <ArrayBuffer> | <Buffer> | <TypedArray> | <DataView>`
- `curve` `<string>`
- `inputEncoding` `<string>` The `encoding` of the `key` string.
- `outputEncoding` `<string>` The `encoding` of the return value.
- `format` `<string>` **Default:** `'uncompressed'`
- Returns: `<Buffer> | <string>`

Converts the EC Diffie-Hellman public key specified by `key` and `curve` to the format specified by `format`. The `format` argument specifies point encoding and can be `'compressed'`, `'uncompressed'` or `'hybrid'`. The supplied key is interpreted using the specified `inputEncoding`, and the returned key is encoded using the specified `outputEncoding`.

Use `crypto.getCurves()` to obtain a list of available curve names. On recent OpenSSL releases, `openssl ecparam -list_curves` will also display the name and description of each available elliptic curve.

If `format` is not specified the point will be returned in `'uncompressed'` format.

If the `inputEncoding` is not provided, `key` is expected to be a `Buffer`, `TypedArray`, or `DataView`.

Example (uncompressing a key):

```
const {
  createECDH,
  ECDH
} = await import('crypto');

const ecdh = createECDH('secp256k1');
ecdh.generateKeys();

const compressedKey = ecdh.getPublicKey('hex', 'compressed');

const uncompressedKey = ECDH.convertKey(compressedKey,
                                         'secp256k1',
                                         'hex',
                                         'hex',
                                         'uncompressed');

// The converted key and the uncompressed public key should be the same
console.log(uncompressedKey === ecdh.getPublicKey('hex'));const {
  createECDH,
  ECDH,
} = require('crypto');

const ecdh = createECDH('secp256k1');
ecdh.generateKeys();

const compressedKey = ecdh.getPublicKey('hex', 'compressed');

const uncompressedKey = ECDH.convertKey(compressedKey,
                                         'secp256k1',
                                         'hex',
                                         'hex',
                                         'uncompressed');

// The converted key and the uncompressed public key should be the same
console.log(uncompressedKey === ecdh.getPublicKey('hex'));
```

`ecdh.computeSecret(otherPublicKey[, inputEncoding][, outputEncoding])`

- `otherPublicKey` `<string> | <ArrayBuffer> | <Buffer> | <TypedArray> | <DataView>`
- `inputEncoding` `<string>` The `encoding` of the `otherPublicKey` string.
- `outputEncoding` `<string>` The `encoding` of the return value.
- Returns: `<Buffer> | <string>`

Computes the shared secret using `otherPublicKey` as the other party's public key and returns the computed shared secret. The supplied key is interpreted using specified `inputEncoding`, and the returned secret is encoded using the specified `outputEncoding`. If the

`inputEncoding` is not provided, `otherPublicKey` is expected to be a `Buffer`, `TypedArray`, or `DataView`.

If `outputEncoding` is given a string will be returned; otherwise a `Buffer` is returned.

`ecdh.computeSecret` will throw an `ERR_CRYPTO_ECDH_INVALID_PUBLIC_KEY` error when `otherPublicKey` lies outside of the elliptic curve. Since `otherPublicKey` is usually supplied from a remote user over an insecure network, be sure to handle this exception accordingly.

`ecdh.generateKeys([encoding[, format]])`

- `encoding` `<string>` The `encoding` of the return value.
- `format` `<string>` **Default:** `'uncompressed'`
- Returns: `<Buffer>` | `<string>`

Generates private and public EC Diffie-Hellman key values, and returns the public key in the specified `format` and `encoding`. This key should be transferred to the other party.

The `format` argument specifies point encoding and can be `'compressed'` or `'uncompressed'`. If `format` is not specified, the point will be returned in `'uncompressed'` format.

If `encoding` is provided a string is returned; otherwise a `Buffer` is returned.

`ecdh.getPrivateKey([encoding])`

- `encoding` `<string>` The `encoding` of the return value.
- Returns: `<Buffer>` | `<string>` The EC Diffie-Hellman in the specified `encoding`.

If `encoding` is specified, a string is returned; otherwise a `Buffer` is returned.

`ecdh.getPublicKey([encoding][, format])`

- `encoding` `<string>` The `encoding` of the return value.
- `format` `<string>` **Default:** `'uncompressed'`
- Returns: `<Buffer>` | `<string>` The EC Diffie-Hellman public key in the specified `encoding` and `format`.

The `format` argument specifies point encoding and can be `'compressed'` or `'uncompressed'`. If `format` is not specified the point will be returned in `'uncompressed'` format.

If `encoding` is specified, a string is returned; otherwise a `Buffer` is returned.

`ecdh.setPrivateKey(privateKey[, encoding])`

- `privateKey` `<string>` | `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>`
- `encoding` `<string>` The `encoding` of the `privateKey` string.

Sets the EC Diffie-Hellman private key. If `encoding` is provided, `privateKey` is expected to be a string; otherwise `privateKey` is expected to be a `Buffer`, `TypedArray`, or `DataView`.

If `privateKey` is not valid for the curve specified when the `ECDH` object was created, an error is thrown. Upon setting the private key, the associated public point (key) is also generated and set in the `ECDH` object.

`ecdh.setPublicKey(publicKey[, encoding])`

Stability: 0 - Deprecated

- `publicKey` `<string>` | `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>`

- `encoding <string>` The `encoding` of the `publicKey` string.

Sets the EC Diffie-Hellman public key. If `encoding` is provided `publicKey` is expected to be a string; otherwise a `Buffer`, `TypedArray`, or `DataView` is expected.

There is not normally a reason to call this method because `ECDH` only requires a private key and the other party's public key to compute the shared secret. Typically either `ecdh.generateKeys()` or `ecdh.setPrivateKey()` will be called. The `ecdh.setPrivateKey()` method attempts to generate the public point/key associated with the private key being set.

Example (obtaining a shared secret):

```
const {
  createECDH,
  createHash
} = await import('crypto');

const alice = createECDH('secp256k1');
const bob = createECDH('secp256k1');

// This is a shortcut way of specifying one of Alice's previous private
// keys. It would be unwise to use such a predictable private key in a real
// application.
alice.setPrivateKey(
  createHash('sha256').update('alice', 'utf8').digest()
);

// Bob uses a newly generated cryptographically strong
// pseudorandom key pair
bob.generateKeys();

const aliceSecret = alice.computeSecret(bob.getPublicKey(), null, 'hex');
const bobSecret = bob.computeSecret(alice.getPublicKey(), null, 'hex');

// aliceSecret and bobSecret should be the same shared secret value
console.log(aliceSecret === bobSecret);const {
  createECDH,
  createHash,
} = require('crypto');

const alice = createECDH('secp256k1');
const bob = createECDH('secp256k1');

// This is a shortcut way of specifying one of Alice's previous private
// keys. It would be unwise to use such a predictable private key in a real
// application.
alice.setPrivateKey(
  createHash('sha256').update('alice', 'utf8').digest()
);

// Bob uses a newly generated cryptographically strong
// pseudorandom key pair
bob.generateKeys();
```



```
const aliceSecret = alice.computeSecret(bob.getPublicKey(), null, 'hex');
const bobSecret = bob.computeSecret(alice.getPublicKey(), null, 'hex');

// aliceSecret and bobSecret should be the same shared secret value
console.log(aliceSecret === bobSecret);
```

Class: Hash

- Extends: `<stream.Transform>`

The `Hash` class is a utility for creating hash digests of data. It can be used in one of two ways:

- As a `stream` that is both readable and writable, where data is written to produce a computed hash digest on the readable side, or
- Using the `hash.update()` and `hash.digest()` methods to produce the computed hash.

The `crypto.createHash()` method is used to create `Hash` instances. `Hash` objects are not to be created directly using the `new` keyword.

Example: Using `Hash` objects as streams:

```
const {
  createHash
} = await import('crypto');

const hash = createHash('sha256');

hash.on('readable', () => {
  // Only one element is going to be produced by the
  // hash stream.
  const data = hash.read();
  if (data) {
    console.log(data.toString('hex'));
    // Prints:
    //   6a2da20943931e9834fc12cfe5bb47bbd9ae43489a30726962b576f4e3993e50
  }
});

hash.write('some data to hash');
hash.end();const {
  createHash,
} = require('crypto');

const hash = createHash('sha256');

hash.on('readable', () => {
  // Only one element is going to be produced by the
  // hash stream.
  const data = hash.read();
  if (data) {
    console.log(data.toString('hex'));
    // Prints:
    //   6a2da20943931e9834fc12cfe5bb47bbd9ae43489a30726962b576f4e3993e50
```

```

    }
  });

  hash.write('some data to hash');
  hash.end();

```

Example: Using `Hash` and piped streams:

```

import { createReadStream } from 'fs';
import { stdout } from 'process';
const { createHash } = await import('crypto');

const hash = createHash('sha256');

const input = createReadStream('test.js');
input.pipe(hash).setEncoding('hex').pipe(stdout);const { createReadStream } = require('fs');
const { createHash } = require('crypto');
const { stdout } = require('process');

const hash = createHash('sha256');

const input = createReadStream('test.js');
input.pipe(hash).setEncoding('hex').pipe(stdout);

```

Example: Using the `hash.update()` and `hash.digest()` methods:

```

const {
  createHash
} = await import('crypto');

const hash = createHash('sha256');

hash.update('some data to hash');
console.log(hash.digest('hex'));
// Prints:
//   6a2da20943931e9834fc12cfe5bb47bbd9ae43489a30726962b576f4e3993e50const {
  createHash,
} = require('crypto');

const hash = createHash('sha256');

hash.update('some data to hash');
console.log(hash.digest('hex'));
// Prints:
//   6a2da20943931e9834fc12cfe5bb47bbd9ae43489a30726962b576f4e3993e50

```

hash.copy([options])

- `options` `<Object>` `stream.transform` options

- Returns: `<Hash>`

Creates a new `Hash` object that contains a deep copy of the internal state of the current `Hash` object.

The optional `options` argument controls stream behavior. For XOF hash functions such as `'shake256'`, the `outputLength` option can be used to specify the desired output length in bytes.

An error is thrown when an attempt is made to copy the `Hash` object after its `hash.digest()` method has been called.

```
// Calculate a rolling hash.
const {
  createHash
} = await import('crypto');

const hash = createHash('sha256');

hash.update('one');
console.log(hash.copy().digest('hex'));

hash.update('two');
console.log(hash.copy().digest('hex'));

hash.update('three');
console.log(hash.copy().digest('hex'));

// Etc.// Calculate a rolling hash.
const {
  createHash,
} = require('crypto');

const hash = createHash('sha256');

hash.update('one');
console.log(hash.copy().digest('hex'));

hash.update('two');
console.log(hash.copy().digest('hex'));

hash.update('three');
console.log(hash.copy().digest('hex'));

// Etc.
```

hash.digest([encoding])

- `encoding` `<string>` The `encoding` of the return value.
- Returns: `<Buffer>` | `<string>`

Calculates the digest of all of the data passed to be hashed (using the `hash.update()` method). If `encoding` is provided a string will be returned; otherwise a `Buffer` is returned.

The `Hash` object can not be used again after `hash.digest()` method has been called. Multiple calls will cause an error to be thrown.

hash.update(data[, inputEncoding])

- `data` `<string>` | `<Buffer>` | `<TypedArray>` | `<DataView>`
- `inputEncoding` `<string>` The `encoding` of the `data` string.

Updates the hash content with the given `data`, the encoding of which is given in `inputEncoding`. If `encoding` is not provided, and the `data` is a string, an encoding of `'utf8'` is enforced. If `data` is a `Buffer`, `TypedArray`, or `DataView`, then `inputEncoding` is ignored.

This can be called many times with new data as it is streamed.

Class: Hmac

- Extends: `<stream.Transform>`

The `Hmac` class is a utility for creating cryptographic HMAC digests. It can be used in one of two ways:

- As a `stream` that is both readable and writable, where data is written to produce a computed HMAC digest on the readable side, or
- Using the `hmac.update()` and `hmac.digest()` methods to produce the computed HMAC digest.

The `crypto.createHmac()` method is used to create `Hmac` instances. `Hmac` objects are not to be created directly using the `new` keyword.

Example: Using `Hmac` objects as streams:

```
const {
  createHmac
} = await import('crypto');

const hmac = createHmac('sha256', 'a secret');

hmac.on('readable', () => {
  // Only one element is going to be produced by the
  // hash stream.
  const data = hmac.read();
  if (data) {
    console.log(data.toString('hex'));
    // Prints:
    //   7fd04df92f636fd450bc841c9418e5825c17f33ad9c87c518115a45971f7f77e
  }
});

hmac.write('some data to hash');
hmac.end();const {
  createHmac,
} = require('crypto');

const hmac = createHmac('sha256', 'a secret');

hmac.on('readable', () => {
  // Only one element is going to be produced by the
  // hash stream.
  const data = hmac.read();
  if (data) {
    console.log(data.toString('hex'));
```

```

    // Prints:
    //    7fd04df92f636fd450bc841c9418e5825c17f33ad9c87c518115a45971f7f77e
  }
});

hmac.write('some data to hash');
hmac.end();

```

Example: Using `Hmac` and piped streams:

```

import { createReadStream } from 'fs';
import { stdout } from 'process';
const {
  createHmac
} = await import('crypto');

const hmac = createHmac('sha256', 'a secret');

const input = createReadStream('test.js');
input.pipe(hmac).pipe(stdout);const {
  createReadStream,
} = require('fs');
const {
  createHmac,
} = require('crypto');
const { stdout } = require('process');

const hmac = createHmac('sha256', 'a secret');

const input = createReadStream('test.js');
input.pipe(hmac).pipe(stdout);

```

Example: Using the `hmac.update()` and `hmac.digest()` methods:

```

const {
  createHmac
} = await import('crypto');

const hmac = createHmac('sha256', 'a secret');

hmac.update('some data to hash');
console.log(hmac.digest('hex'));
// Prints:
//    7fd04df92f636fd450bc841c9418e5825c17f33ad9c87c518115a45971f7f77econst {
  createHmac,
} = require('crypto');

const hmac = createHmac('sha256', 'a secret');

hmac.update('some data to hash');

```

```
console.log(hmac.digest('hex'));  
// Prints:  
// 7fd04df92f636fd450bc841c9418e5825c17f33ad9c87c518115a45971f7f77e
```

hmac.digest([encoding])

- `encoding` `<string>` The `encoding` of the return value.
- Returns: `<Buffer>` | `<string>`

Calculates the HMAC digest of all of the data passed using `hmac.update()`. If `encoding` is provided a string is returned; otherwise a `Buffer` is returned;

The `Hmac` object can not be used again after `hmac.digest()` has been called. Multiple calls to `hmac.digest()` will result in an error being thrown.

hmac.update(data[, inputEncoding])

- `data` `<string>` | `<Buffer>` | `<TypedArray>` | `<DataView>`
- `inputEncoding` `<string>` The `encoding` of the data string.

Updates the `Hmac` content with the given `data`, the encoding of which is given in `inputEncoding`. If `encoding` is not provided, and the `data` is a string, an encoding of `'utf8'` is enforced. If `data` is a `Buffer`, `TypedArray`, or `DataView`, then `inputEncoding` is ignored.

This can be called many times with new data as it is streamed.

Class: KeyObject

Node.js uses a `KeyObject` class to represent a symmetric or asymmetric key, and each kind of key exposes different functions. The `crypto.createSecretKey()`, `crypto.createPublicKey()` and `crypto.createPrivateKey()` methods are used to create `KeyObject` instances. `KeyObject` objects are not to be created directly using the `new` keyword.

Most applications should consider using the new `KeyObject` API instead of passing keys as strings or `Buffer`s due to improved security features.

`KeyObject` instances can be passed to other threads via `postMessage()`. The receiver obtains a cloned `KeyObject`, and the `KeyObject` does not need to be listed in the `transferList` argument.

Static method: KeyObject.from(key)

- `key` `<CryptoKey>`
- Returns: `<KeyObject>`

Example: Converting a `CryptoKey` instance to a `KeyObject`:

```
const { webcrypto, KeyObject } = await import('crypto');  
const { subtle } = webcrypto;  
  
const key = await subtle.generateKey({  
  name: 'HMAC',  
  hash: 'SHA-256',  
  length: 256  
}, true, ['sign', 'verify']);
```

```

const keyObject = KeyObject.from(key);
console.log(keyObject.symmetricKeySize);
// Prints: 32 (symmetric key size in bytes)
const {
  webcrypto: {
    subtle,
  },
  KeyObject,
} = require('crypto');

(async function() {
  const key = await subtle.generateKey({
    name: 'HMAC',
    hash: 'SHA-256',
    length: 256
  }, true, ['sign', 'verify']);

  const keyObject = KeyObject.from(key);
  console.log(keyObject.symmetricKeySize);
  // Prints: 32 (symmetric key size in bytes)
})();

```

keyObject.asymmetricKeyDetails

- `<Object>`
 - `modulusLength` : `<number>` Key size in bits (RSA, DSA).
 - `publicExponent` : `<bigint>` Public exponent (RSA).
 - `divisorLength` : `<number>` Size of `q` in bits (DSA).
 - `namedCurve` : `<string>` Name of the curve (EC).

This property exists only on asymmetric keys. Depending on the type of the key, this object contains information about the key. None of the information obtained through this property can be used to uniquely identify a key or to compromise the security of the key.

RSA-PSS parameters, DH, or any future key type details might be exposed via this API using additional attributes.

keyObject.asymmetricKeyType

- `<string>`

For asymmetric keys, this property represents the type of the key. Supported key types are:

- `'rsa'` (OID 1.2.840.113549.1.1.1)
- `'rsa-pss'` (OID 1.2.840.113549.1.1.10)
- `'dsa'` (OID 1.2.840.10040.4.1)
- `'ec'` (OID 1.2.840.10045.2.1)
- `'x25519'` (OID 1.3.101.110)
- `'x448'` (OID 1.3.101.111)
- `'ed25519'` (OID 1.3.101.112)
- `'ed448'` (OID 1.3.101.113)
- `'dh'` (OID 1.2.840.113549.1.3.1)

This property is `undefined` for unrecognized `KeyObject` types and symmetric keys.

keyObject.export([options])

- `options`: `<Object>`
- Returns: `<string> | <Buffer> | <Object>`

For symmetric keys, the following encoding options can be used:

- `format`: `<string>` Must be `'buffer'` (default) or `'jwk'`.

For public keys, the following encoding options can be used:

- `type`: `<string>` Must be one of `'pkcs1'` (RSA only) or `'spki'`.
- `format`: `<string>` Must be `'pem'`, `'der'`, or `'jwk'`.

For private keys, the following encoding options can be used:

- `type`: `<string>` Must be one of `'pkcs1'` (RSA only), `'pkcs8'` or `'sec1'` (EC only).
- `format`: `<string>` Must be `'pem'`, `'der'`, or `'jwk'`.
- `cipher`: `<string>` If specified, the private key will be encrypted with the given `cipher` and `passphrase` using PKCS#5 v2.0 password based encryption.
- `passphrase`: `<string> | <Buffer>` The passphrase to use for encryption, see `cipher`.

The result type depends on the selected encoding format, when PEM the result is a string, when DER it will be a buffer containing the data encoded as DER, when `JWK` it will be an object.

When `JWK` encoding format was selected, all other encoding options are ignored.

PKCS#1, SEC1, and PKCS#8 type keys can be encrypted by using a combination of the `cipher` and `format` options. The PKCS#8 `type` can be used with any `format` to encrypt any key algorithm (RSA, EC, or DH) by specifying a `cipher`. PKCS#1 and SEC1 can only be encrypted by specifying a `cipher` when the PEM `format` is used. For maximum compatibility, use PKCS#8 for encrypted private keys. Since PKCS#8 defines its own encryption mechanism, PEM-level encryption is not supported when encrypting a PKCS#8 key. See [RFC 5208](#) for PKCS#8 encryption and [RFC 1421](#) for PKCS#1 and SEC1 encryption.

keyObject.symmetricKeySize

- `<number>`

For secret keys, this property represents the size of the key in bytes. This property is `undefined` for asymmetric keys.

keyObject.type

- `<string>`

Depending on the type of this `KeyObject`, this property is either `'secret'` for secret (symmetric) keys, `'public'` for public (asymmetric) keys or `'private'` for private (asymmetric) keys.

Class: Sign

- Extends: `<stream.Writable>`

The `Sign` class is a utility for generating signatures. It can be used in one of two ways:

- As a writable `stream`, where data to be signed is written and the `sign.sign()` method is used to generate and return the signature, or
- Using the `sign.update()` and `sign.sign()` methods to produce the signature.

The `crypto.createSign()` method is used to create `Sign` instances. The argument is the string name of the hash function to use. `Sign` objects are not to be created directly using the `new` keyword.

Example: Using `Sign` and `Verify` objects as streams:

```
const {
  generateKeyPairSync,
  createSign,
  createVerify
} = await import('crypto');

const { privateKey, publicKey } = generateKeyPairSync('ec', {
  namedCurve: 'sect239k1'
});

const sign = createSign('SHA256');
sign.write('some data to sign');
sign.end();
const signature = sign.sign(privateKey, 'hex');

const verify = createVerify('SHA256');
verify.write('some data to sign');
verify.end();
console.log(verify.verify(publicKey, signature, 'hex'));
// Prints: trueconst {
  generateKeyPairSync,
  createSign,
  createVerify,
} = require('crypto');

const { privateKey, publicKey } = generateKeyPairSync('ec', {
  namedCurve: 'sect239k1'
});

const sign = createSign('SHA256');
sign.write('some data to sign');
sign.end();
const signature = sign.sign(privateKey, 'hex');

const verify = createVerify('SHA256');
verify.write('some data to sign');
verify.end();
console.log(verify.verify(publicKey, signature, 'hex'));
// Prints: true
```

Example: Using the `sign.update()` and `verify.update()` methods:

```
const {
  generateKeyPairSync,
  createSign,
  createVerify
} = await import('crypto');
```

```

const { privateKey, publicKey } = generateKeyPairSync('rsa', {
  modulusLength: 2048,
});

const sign = createSign('SHA256');
sign.update('some data to sign');
sign.end();
const signature = sign.sign(privateKey);

const verify = createVerify('SHA256');
verify.update('some data to sign');
verify.end();
console.log(verify.verify(publicKey, signature));
// Prints: trueconst {
  generateKeyPairSync,
  createSign,
  createVerify,
} = require('crypto');

const { privateKey, publicKey } = generateKeyPairSync('rsa', {
  modulusLength: 2048,
});

const sign = createSign('SHA256');
sign.update('some data to sign');
sign.end();
const signature = sign.sign(privateKey);

const verify = createVerify('SHA256');
verify.update('some data to sign');
verify.end();
console.log(verify.verify(publicKey, signature));
// Prints: true

```

sign.sign(privateKey[, outputEncoding])

- `privateKey` `<Object>` | `<string>` | `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>` | `<KeyObject>` | `<CryptoKey>`
 - `dsaEncoding` `<string>`
 - `padding` `<integer>`
 - `saltLength` `<integer>`
- `outputEncoding` `<string>` The `encoding` of the return value.
- Returns: `<Buffer>` | `<string>`

Calculates the signature on all the data passed through using either `sign.update()` or `sign.write()`.

If `privateKey` is not a `KeyObject`, this function behaves as if `privateKey` had been passed to `crypto.createPrivateKey()`. If it is an object, the following additional properties can be passed:

- `dsaEncoding` `<string>` For DSA and ECDSA, this option specifies the format of the generated signature. It can be one of the following:
 - `'der'` (default): DER-encoded ASN.1 signature structure encoding `(r, s)`.

- `'ieee-p1363'`: Signature format `r || s` as proposed in IEEE-P1363.
- `padding` `<integer>` Optional padding value for RSA, one of the following:
 - `crypto.constants.RSA_PKCS1_PADDING` (default)
 - `crypto.constants.RSA_PKCS1_PSS_PADDING`
`RSA_PKCS1_PSS_PADDING` will use MGF1 with the same hash function used to sign the message as specified in section 3.1 of [RFC 4055](#), unless an MGF1 hash function has been specified as part of the key in compliance with section 3.3 of [RFC 4055](#).
- `saltLength` `<integer>` Salt length for when padding is `RSA_PKCS1_PSS_PADDING`. The special value `crypto.constants.RSA_PSS_SALTLEN_DIGEST` sets the salt length to the digest size, `crypto.constants.RSA_PSS_SALTLEN_MAX_SIGN` (default) sets it to the maximum permissible value.

If `outputEncoding` is provided a string is returned; otherwise a `Buffer` is returned.

The `Sign` object can not be again used after `sign.sign()` method has been called. Multiple calls to `sign.sign()` will result in an error being thrown.

`sign.update(data[, inputEncoding])`

- `data` `<string>` | `<Buffer>` | `<TypedArray>` | `<DataView>`
- `inputEncoding` `<string>` The `encoding` of the `data` string.

Updates the `Sign` content with the given `data`, the encoding of which is given in `inputEncoding`. If `encoding` is not provided, and the `data` is a string, an encoding of `'utf8'` is enforced. If `data` is a `Buffer`, `TypedArray`, or `DataView`, then `inputEncoding` is ignored.

This can be called many times with new data as it is streamed.

Class: `Verify`

- Extends: `<stream.Writable>`

The `Verify` class is a utility for verifying signatures. It can be used in one of two ways:

- As a writable `stream` where written data is used to validate against the supplied signature, or
- Using the `verify.update()` and `verify.verify()` methods to verify the signature.

The `crypto.createVerify()` method is used to create `Verify` instances. `Verify` objects are not to be created directly using the `new` keyword.

See `Sign` for examples.

`verify.update(data[, inputEncoding])`

- `data` `<string>` | `<Buffer>` | `<TypedArray>` | `<DataView>`
- `inputEncoding` `<string>` The `encoding` of the `data` string.

Updates the `Verify` content with the given `data`, the encoding of which is given in `inputEncoding`. If `inputEncoding` is not provided, and the `data` is a string, an encoding of `'utf8'` is enforced. If `data` is a `Buffer`, `TypedArray`, or `DataView`, then `inputEncoding` is ignored.

This can be called many times with new data as it is streamed.

`verify.verify(object, signature[, signatureEncoding])`

- `object` `<Object>` | `<string>` | `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>` | `<KeyObject>` | `<CryptoKey>`
 - `dsaEncoding` `<string>`
 - `padding` `<integer>`

- `saltLength` `<integer>`
- `signature` `<string> | <ArrayBuffer> | <Buffer> | <TypedArray> | <DataView>`
- `signatureEncoding` `<string>` The encoding of the signature string.
- Returns: `<boolean>` `true` or `false` depending on the validity of the signature for the data and public key.

Verifies the provided data using the given `object` and `signature`.

If `object` is not a `KeyObject`, this function behaves as if `object` had been passed to `crypto.createPublicKey()`. If it is an object, the following additional properties can be passed:

- `dsaEncoding` `<string>` For DSA and ECDSA, this option specifies the format of the signature. It can be one of the following:
 - `'der'` (default): DER-encoded ASN.1 signature structure encoding (`r`, `s`).
 - `'ieee-p1363'`: Signature format `r || s` as proposed in IEEE-P1363.
- `padding` `<integer>` Optional padding value for RSA, one of the following:
 - `crypto.constants.RSA_PKCS1_PADDING` (default)
 - `crypto.constants.RSA_PKCS1_PSS_PADDING`
`RSA_PKCS1_PSS_PADDING` will use MGF1 with the same hash function used to verify the message as specified in section 3.1 of [RFC 4055](#), unless an MGF1 hash function has been specified as part of the key in compliance with section 3.3 of [RFC 4055](#).
- `saltLength` `<integer>` Salt length for when padding is `RSA_PKCS1_PSS_PADDING`. The special value `crypto.constants.RSA_PSS_SALTLEN_DIGEST` sets the salt length to the digest size, `crypto.constants.RSA_PSS_SALTLEN_AUTO` (default) causes it to be determined automatically.

The `signature` argument is the previously calculated signature for the data, in the `signatureEncoding`. If a `signatureEncoding` is specified, the `signature` is expected to be a string; otherwise `signature` is expected to be a `Buffer`, `TypedArray`, or `DataView`.

The `verify` object can not be used again after `verify.verify()` has been called. Multiple calls to `verify.verify()` will result in an error being thrown.

Because public keys can be derived from private keys, a private key may be passed instead of a public key.

Class: X509Certificate

Encapsulates an X509 certificate and provides read-only access to its information.

```
const { X509Certificate } = await import('crypto');

const x509 = new X509Certificate('{... pem encoded cert ...}');

console.log(x509.subject);const { X509Certificate } = require('crypto');

const x509 = new X509Certificate('{... pem encoded cert ...}');

console.log(x509.subject);
```

new X509Certificate(buffer)

- `buffer` `<string> | <TypedArray> | <Buffer> | <DataView>` A PEM or DER encoded X509 Certificate.

- Type: `<boolean>` Will be `true` if this is a Certificate Authority (ca) certificate.

x509.checkEmail(email[, options])

- `email` `<string>`
- `options` `<Object>`
 - `subject` `<string>` 'always' or 'never'. Default: 'always'.
 - `wildcards` `<boolean>` Default: `true`.
 - `partialWildcards` `<boolean>` Default: `true`.
 - `multiLabelWildcards` `<boolean>` Default: `false`.
 - `singleLabelSubdomains` `<boolean>` Default: `false`.
- Returns: `<string>` | `<undefined>` Returns `email` if the certificate matches, `undefined` if it does not.

Checks whether the certificate matches the given email address.

x509.checkHost(name[, options])

- `name` `<string>`
- `options` `<Object>`
 - `subject` `<string>` 'always' or 'never'. Default: 'always'.
 - `wildcards` `<boolean>` Default: `true`.
 - `partialWildcards` `<boolean>` Default: `true`.
 - `multiLabelWildcards` `<boolean>` Default: `false`.
 - `singleLabelSubdomains` `<boolean>` Default: `false`.
- Returns: `<string>` | `<undefined>` Returns `name` if the certificate matches, `undefined` if it does not.

Checks whether the certificate matches the given host name.

x509.checkIP(ip[, options])

- `ip` `<string>`
- `options` `<Object>`
 - `subject` `<string>` 'always' or 'never'. Default: 'always'.
 - `wildcards` `<boolean>` Default: `true`.
 - `partialWildcards` `<boolean>` Default: `true`.
 - `multiLabelWildcards` `<boolean>` Default: `false`.
 - `singleLabelSubdomains` `<boolean>` Default: `false`.
- Returns: `<string>` | `<undefined>` Returns `ip` if the certificate matches, `undefined` if it does not.

Checks whether the certificate matches the given IP address (IPv4 or IPv6).

x509.checkIssued(otherCert)

- `otherCert` `<X509Certificate>`
- Returns: `<boolean>`

Checks whether this certificate was issued by the given `otherCert`.

x509.checkPrivateKey(privateKey)

- `privateKey` `<KeyObject>` A private key.

- Returns: `<boolean>`

Checks whether the public key for this certificate is consistent with the given private key.

x509.fingerprint

- Type: `<string>`

The SHA-1 fingerprint of this certificate.

x509.fingerprint256

- Type: `<string>`

The SHA-256 fingerprint of this certificate.

x509.infoAccess

- Type: `<string>`

The information access content of this certificate.

x509.issuer

- Type: `<string>`

The issuer identification included in this certificate.

x509.issuerCertificate

- Type: `<X509Certificate>`

The issuer certificate or `undefined` if the issuer certificate is not available.

x509.keyUsage

- Type: `<string[]>`

An array detailing the key usages for this certificate.

x509.publicKey

- Type: `<KeyObject>`

The public key `<KeyObject>` for this certificate.

x509.raw

- Type: `<Buffer>`

A `Buffer` containing the DER encoding of this certificate.

x509.serialNumber

- Type: `<string>`

The serial number of this certificate.

x509.subject

- Type: `<string>`

The complete subject of this certificate.

x509.subjectAltName

- Type: `<string>`

The subject alternative name specified for this certificate.

x509.toJSON()

- Type: `<string>`

There is no standard JSON encoding for X509 certificates. The `toJSON()` method returns a string containing the PEM encoded certificate.

x509.toLegacyObject()

- Type: `<Object>`

Returns information about this certificate using the legacy `certificate object` encoding.

x509.toString()

- Type: `<string>`

Returns the PEM-encoded certificate.

x509.validFrom

- Type: `<string>`

The date/time from which this certificate is considered valid.

x509.validTo

- Type: `<string>`

The date/time until which this certificate is considered valid.

x509.verify(publicKey)

- `publicKey` `<KeyObject>` A public key.
- Returns: `<boolean>`

Verifies that this certificate was signed by the given public key. Does not perform any other validation checks on the certificate.

crypto module methods and properties

crypto.constants

- Returns: `<Object>` An object containing commonly used constants for crypto and security related operations. The specific constants currently defined are described in [Crypto constants](#).

crypto.DEFAULT_ENCODING

Stability: 0 - Deprecated

The default encoding to use for functions that can take either strings or `buffers`. The default value is `'buffer'`, which makes methods default to `Buffer` objects.

The `crypto.DEFAULT_ENCODING` mechanism is provided for backward compatibility with legacy programs that expect `'latin1'` to be the default encoding.

New applications should expect the default to be `'buffer'`.

This property is deprecated.

crypto.fips

Stability: 0 - Deprecated

Property for checking and controlling whether a FIPS compliant crypto provider is currently in use. Setting to true requires a FIPS build of Node.js.

This property is deprecated. Please use `crypto.setFips()` and `crypto.getFips()` instead.

crypto.checkPrime(candidate[, options, [callback]])

- `candidate` `<ArrayBuffer> | <SharedArrayBuffer> | <TypedArray> | <Buffer> | <DataView> | <bigint>` A possible prime encoded as a sequence of big endian octets of arbitrary length.
- `options` `<Object>`
 - `checks` `<number>` The number of Miller-Rabin probabilistic primality iterations to perform. When the value is `0` (zero), a number of checks is used that yields a false positive rate of at most 2^{-64} for random input. Care must be used when selecting a number of checks. Refer to the OpenSSL documentation for the `BN_is_prime_ex` function `nchecks` options for more details. **Default:** `0`
- `callback` `<Function>`
 - `err` `<Error>` Set to an `<Error>` object if an error occurred during check.
 - `result` `<boolean>` `true` if the candidate is a prime with an error probability less than `0.25 ** options.checks`.

Checks the primality of the `candidate`.

crypto.checkPrimeSync(candidate[, options])

- `candidate` `<ArrayBuffer> | <SharedArrayBuffer> | <TypedArray> | <Buffer> | <DataView> | <bigint>` A possible prime encoded as a sequence of big endian octets of arbitrary length.
- `options` `<Object>`
 - `checks` `<number>` The number of Miller-Rabin probabilistic primality iterations to perform. When the value is `0` (zero), a number of checks is used that yields a false positive rate of at most 2^{-64} for random input. Care must be used when selecting a number of checks. Refer to the OpenSSL documentation for the `BN_is_prime_ex` function `nchecks` options for more details. **Default:** `0`
- Returns: `<boolean>` `true` if the candidate is a prime with an error probability less than `0.25 ** options.checks`.

Checks the primality of the `candidate`.

crypto.createCipher(algorithm, password[, options])

Stability: 0 - Deprecated: Use `crypto.createCipheriv()` instead.

- `algorithm` `<string>`
- `password` `<string> | <ArrayBuffer> | <Buffer> | <TypedArray> | <DataView>`
- `options` `<Object>` `stream.transform` options
- Returns: `<Cipher>`

Creates and returns a `Cipher` object that uses the given `algorithm` and `password`.

The `options` argument controls stream behavior and is optional except when a cipher in CCM or OCB mode is used (e.g. `'aes-128-ccm'`). In that case, the `authTagLength` option is required and specifies the length of the authentication tag in bytes, see [CCM mode](#). In GCM mode, the `authTagLength` option is not required but can be used to set the length of the authentication tag that will be returned by `getAuthTag()` and defaults to 16 bytes.

The `algorithm` is dependent on OpenSSL, examples are `'aes192'`, etc. On recent OpenSSL releases, `openssl list -cipher -algorithms` (`openssl list-cipher-algorithms` for older versions of OpenSSL) will display the available cipher algorithms.

The `password` is used to derive the cipher key and initialization vector (IV). The value must be either a `'latin1'` encoded string, a `Buffer`, a `TypedArray`, or a `DataView`.

The implementation of `crypto.createCipher()` derives keys using the OpenSSL function `EVP_BytesToKey` with the digest algorithm set to MD5, one iteration, and no salt. The lack of salt allows dictionary attacks as the same password always creates the same key. The low iteration count and non-cryptographically secure hash algorithm allow passwords to be tested very rapidly.

In line with OpenSSL's recommendation to use a more modern algorithm instead of `EVP_BytesToKey` it is recommended that developers derive a key and IV on their own using `crypto.scrypt()` and to use `crypto.createCipheriv()` to create the `Cipher` object. Users should not use ciphers with counter mode (e.g. CTR, GCM, or CCM) in `crypto.createCipher()`. A warning is emitted when they are used in order to avoid the risk of IV reuse that causes vulnerabilities. For the case when IV is reused in GCM, see [Nonce-Disrespecting Adversaries](#) for details.

`crypto.createCipheriv(algorithm, key, iv[, options])`

- `algorithm` `<string>`
- `key` `<string>` | `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>` | `<KeyObject>` | `<CryptoKey>`
- `iv` `<string>` | `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>` | `<null>`
- `options` `<Object>` `stream.transformOptions`
- Returns: `<Cipher>`

Creates and returns a `Cipher` object, with the given `algorithm`, `key` and initialization vector (`iv`).

The `options` argument controls stream behavior and is optional except when a cipher in CCM or OCB mode is used (e.g. `'aes-128-ccm'`). In that case, the `authTagLength` option is required and specifies the length of the authentication tag in bytes, see [CCM mode](#). In GCM mode, the `authTagLength` option is not required but can be used to set the length of the authentication tag that will be returned by `getAuthTag()` and defaults to 16 bytes.

The `algorithm` is dependent on OpenSSL, examples are `'aes192'`, etc. On recent OpenSSL releases, `openssl list -cipher -algorithms` (`openssl list-cipher-algorithms` for older versions of OpenSSL) will display the available cipher algorithms.

The `key` is the raw key used by the `algorithm` and `iv` is an `initialization vector`. Both arguments must be `'utf8'` encoded strings, `Buffers`, `TypedArray`, or `DataView`s. The `key` may optionally be a `KeyObject` of type `secret`. If the cipher does not need an initialization vector, `iv` may be `null`.

When passing strings for `key` or `iv`, please consider [caveats when using strings as inputs to cryptographic APIs](#).

Initialization vectors should be unpredictable and unique; ideally, they will be cryptographically random. They do not have to be secret: IVs are typically just added to ciphertext messages unencrypted. It may sound contradictory that something has to be unpredictable and unique, but does not have to be secret; remember that an attacker must not be able to predict ahead of time what a given IV will be.

`crypto.createDecipher(algorithm, password[, options])`

Stability: 0 - Deprecated: Use `crypto.createDecipheriv()` instead.

- `algorithm` `<string>`
- `password` `<string>` | `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>`
- `options` `<Object>` `stream.transformOptions`
- Returns: `<Decipher>`

Creates and returns a `Decipher` object that uses the given `algorithm` and `password` (key).

The `options` argument controls stream behavior and is optional except when a cipher in CCM or OCB mode is used (e.g. `'aes-128-ccm'`). In that case, the `authTagLength` option is required and specifies the length of the authentication tag in bytes, see [CCM mode](#).

The implementation of `crypto.createDecipher()` derives keys using the OpenSSL function `EVP_BytesToKey` with the digest algorithm set to MD5, one iteration, and no salt. The lack of salt allows dictionary attacks as the same password always creates the same key. The low iteration count and non-cryptographically secure hash algorithm allow passwords to be tested very rapidly.

In line with OpenSSL's recommendation to use a more modern algorithm instead of `EVP_BytesToKey` it is recommended that developers derive a key and IV on their own using `crypto.scrypt()` and to use `crypto.createDecipheriv()` to create the `Decipher` object.

`crypto.createDecipheriv(algorithm, key, iv[, options])`

- `algorithm` `<string>`
- `key` `<string>` | `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>` | `<KeyObject>` | `<CryptoKey>`
- `iv` `<string>` | `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>` | `<null>`
- `options` `<Object>` `stream.transformOptions`
- Returns: `<Decipher>`

Creates and returns a `Decipher` object that uses the given `algorithm`, `key` and initialization vector (`iv`).

The `options` argument controls stream behavior and is optional except when a cipher in CCM or OCB mode is used (e.g. `'aes-128-ccm'`). In that case, the `authTagLength` option is required and specifies the length of the authentication tag in bytes, see [CCM mode](#). In GCM mode, the `authTagLength` option is not required but can be used to restrict accepted authentication tags to those with the specified length.

The `algorithm` is dependent on OpenSSL, examples are `'aes192'`, etc. On recent OpenSSL releases, `openssl list -cipher-algorithms` (`openssl list-cipher-algorithms` for older versions of OpenSSL) will display the available cipher algorithms.

The `key` is the raw key used by the `algorithm` and `iv` is an [initialization vector](#). Both arguments must be `'utf8'` encoded strings, `Buffers`, `TypedArray`, or `DataView`s. The `key` may optionally be a `KeyObject` of type `secret`. If the cipher does not need an initialization vector, `iv` may be `null`.

When passing strings for `key` or `iv`, please consider [caveats when using strings as inputs to cryptographic APIs](#).

Initialization vectors should be unpredictable and unique; ideally, they will be cryptographically random. They do not have to be secret: IVs are typically just added to ciphertext messages unencrypted. It may sound contradictory that something has to be unpredictable and unique, but does not have to be secret; remember that an attacker must not be able to predict ahead of time what a given IV will be.

`crypto.createDiffieHellman(prime[, primeEncoding][, generator][, generatorEncoding])`

- `prime` `<string>` | `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>`
- `primeEncoding` `<string>` The [encoding](#) of the `prime` string.
- `generator` `<number>` | `<string>` | `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>` **Default:** 2
- `generatorEncoding` `<string>` The [encoding](#) of the `generator` string.

- Returns: `<DiffieHellman>`

Creates a `DiffieHellman` key exchange object using the supplied `prime` and an optional specific `generator` .

The `generator` argument can be a number, string, or `Buffer` . If `generator` is not specified, the value `2` is used.

If `primeEncoding` is specified, `prime` is expected to be a string; otherwise a `Buffer` , `TypedArray` , or `DataView` is expected.

If `generatorEncoding` is specified, `generator` is expected to be a string; otherwise a number, `Buffer` , `TypedArray` , or `DataView` is expected.

`crypto.createDiffieHellman(primeLength[, generator])`

- `primeLength` `<number>`
- `generator` `<number>` Default: `2`
- Returns: `<DiffieHellman>`

Creates a `DiffieHellman` key exchange object and generates a prime of `primeLength` bits using an optional specific numeric `generator` . If `generator` is not specified, the value `2` is used.

`crypto.createDiffieHellmanGroup(name)`

- `name` `<string>`
- Returns: `<DiffieHellmanGroup>`

An alias for `crypto.getDiffieHellman()`

`crypto.createECDH(curveName)`

- `curveName` `<string>`
- Returns: `<ECDH>`

Creates an Elliptic Curve Diffie-Hellman (`ECDH`) key exchange object using a predefined curve specified by the `curveName` string. Use `crypto.getCurves()` to obtain a list of available curve names. On recent OpenSSL releases, `openssl ecparam -list_curves` will also display the name and description of each available elliptic curve.

`crypto.createHash(algorithm[, options])`

- `algorithm` `<string>`
- `options` `<Object>` `stream.transformOptions`
- Returns: `<Hash>`

Creates and returns a `Hash` object that can be used to generate hash digests using the given `algorithm` . Optional `options` argument controls stream behavior. For XOF hash functions such as `'shake256'` , the `outputLength` option can be used to specify the desired output length in bytes.

The `algorithm` is dependent on the available algorithms supported by the version of OpenSSL on the platform. Examples are `'sha256'` , `'sha512'` , etc. On recent releases of OpenSSL, `openssl list -digest-algorithms` (`openssl list-message-digest-algorithms` for older versions of OpenSSL) will display the available digest algorithms.

Example: generating the sha256 sum of a file

```
import {
  createReadStream
} from 'fs';
import { argv } from 'process';
```

```

const {
  createHash
} = await import('crypto');

const filename = argv[2];

const hash = createHash('sha256');

const input = createReadStream(filename);
input.on('readable', () => {
  // Only one element is going to be produced by the
  // hash stream.
  const data = input.read();
  if (data)
    hash.update(data);
  else {
    console.log(`${hash.digest('hex')} ${filename}`);
  }
});const {
  createReadStream,
} = require('fs');
const {
  createHash,
} = require('crypto');
const { argv } = require('process');

const filename = argv[2];

const hash = createHash('sha256');

const input = createReadStream(filename);
input.on('readable', () => {
  // Only one element is going to be produced by the
  // hash stream.
  const data = input.read();
  if (data)
    hash.update(data);
  else {
    console.log(`${hash.digest('hex')} ${filename}`);
  }
});

```

crypto.createHmac(algorithm, key[, options])

- **algorithm** `<string>`
- **key** `<string> | <ArrayBuffer> | <Buffer> | <TypedArray> | <DataView> | <KeyObject> | <CryptoKey>`
- **options** `<Object>` `stream.transform` options
 - **encoding** `<string>` The string encoding to use when **key** is a string.
- Returns: `<Hmac>`

Creates and returns an `Hmac` object that uses the given `algorithm` and `key`. Optional `options` argument controls stream behavior.

The `algorithm` is dependent on the available algorithms supported by the version of OpenSSL on the platform. Examples are `'sha256'`, `'sha512'`, etc. On recent releases of OpenSSL, `openssl list -digest -algorithms` (`openssl list-message-digest-algorithms` for older versions of OpenSSL) will display the available digest algorithms.

The `key` is the HMAC key used to generate the cryptographic HMAC hash. If it is a `KeyObject`, its type must be `secret`.

Example: generating the sha256 HMAC of a file

```
import {
  createReadStream
} from 'fs';
import { argv } from 'process';
const {
  createHmac
} = await import('crypto');

const filename = argv[2];

const hmac = createHmac('sha256', 'a secret');

const input = createReadStream(filename);
input.on('readable', () => {
  // Only one element is going to be produced by the
  // hash stream.
  const data = input.read();
  if (data)
    hmac.update(data);
  else {
    console.log(`${hmac.digest('hex')} ${filename}`);
  }
});const {
  createReadStream,
} = require('fs');
const {
  createHmac,
} = require('crypto');
const { argv } = require('process');

const filename = argv[2];

const hmac = createHmac('sha256', 'a secret');

const input = createReadStream(filename);
input.on('readable', () => {
  // Only one element is going to be produced by the
  // hash stream.
  const data = input.read();
  if (data)
    hmac.update(data);
  else {
    console.log(`${hmac.digest('hex')} ${filename}`);
  }
});
```

```
}  
});
```

crypto.createPrivateKey(key)

- `key` `<Object> | <string> | <ArrayBuffer> | <Buffer> | <TypedArray> | <DataView>`
 - `key`: `<string> | <ArrayBuffer> | <Buffer> | <TypedArray> | <DataView> | <Object>` The key material, either in PEM, DER, or JWK format.
 - `format`: `<string>` Must be `'pem'`, `'der'`, or `'jwk'`. **Default:** `'pem'`.
 - `type`: `<string>` Must be `'pkcs1'`, `'pkcs8'` or `'sec1'`. This option is required only if the `format` is `'der'` and ignored otherwise.
 - `passphrase`: `<string> | <Buffer>` The passphrase to use for decryption.
 - `encoding`: `<string>` The string encoding to use when `key` is a string.
- Returns: `<KeyObject>`

Creates and returns a new key object containing a private key. If `key` is a string or `Buffer`, `format` is assumed to be `'pem'`; otherwise, `key` must be an object with the properties described above.

If the private key is encrypted, a `passphrase` must be specified. The length of the passphrase is limited to 1024 bytes.

crypto.createPublicKey(key)

- `key` `<Object> | <string> | <ArrayBuffer> | <Buffer> | <TypedArray> | <DataView>`
 - `key`: `<string> | <ArrayBuffer> | <Buffer> | <TypedArray> | <DataView> | <Object>` The key material, either in PEM, DER, or JWK format.
 - `format`: `<string>` Must be `'pem'`, `'der'`, or `'jwk'`. **Default:** `'pem'`.
 - `type`: `<string>` Must be `'pkcs1'` or `'spki'`. This option is required only if the `format` is `'der'` and ignored otherwise.
 - `encoding` `<string>` The string encoding to use when `key` is a string.
- Returns: `<KeyObject>`

Creates and returns a new key object containing a public key. If `key` is a string or `Buffer`, `format` is assumed to be `'pem'`; if `key` is a `KeyObject` with type `'private'`, the public key is derived from the given private key; otherwise, `key` must be an object with the properties described above.

If the format is `'pem'`, the `'key'` may also be an X.509 certificate.

Because public keys can be derived from private keys, a private key may be passed instead of a public key. In that case, this function behaves as if `crypto.createPrivateKey()` had been called, except that the type of the returned `KeyObject` will be `'public'` and that the private key cannot be extracted from the returned `KeyObject`. Similarly, if a `KeyObject` with type `'private'` is given, a new `KeyObject` with type `'public'` will be returned and it will be impossible to extract the private key from the returned object.

crypto.createSecretKey(key[, encoding])

- `key` `<string> | <ArrayBuffer> | <Buffer> | <TypedArray> | <DataView>`
- `encoding` `<string>` The string encoding when `key` is a string.
- Returns: `<KeyObject>`

Creates and returns a new key object containing a secret key for symmetric encryption or `Hmac`.

crypto.createSign(algorithm[, options])

- `algorithm` `<string>`
- `options` `<Object>` `stream.Writable` options

- Returns: `<Sign>`

Creates and returns a `Sign` object that uses the given `algorithm`. Use `crypto.getHashes()` to obtain the names of the available digest algorithms. Optional `options` argument controls the `stream.Writable` behavior.

In some cases, a `Sign` instance can be created using the name of a signature algorithm, such as `'RSA-SHA256'`, instead of a digest algorithm. This will use the corresponding digest algorithm. This does not work for all signature algorithms, such as `'ecdsa-with-SHA256'`, so it is best to always use digest algorithm names.

`crypto.createVerify(algorithm[, options])`

- `algorithm` `<string>`
- `options` `<Object>` `stream.Writable` options
- Returns: `<Verify>`

Creates and returns a `Verify` object that uses the given algorithm. Use `crypto.getHashes()` to obtain an array of names of the available signing algorithms. Optional `options` argument controls the `stream.Writable` behavior.

In some cases, a `Verify` instance can be created using the name of a signature algorithm, such as `'RSA-SHA256'`, instead of a digest algorithm. This will use the corresponding digest algorithm. This does not work for all signature algorithms, such as `'ecdsa-with-SHA256'`, so it is best to always use digest algorithm names.

`crypto.diffieHellman(options)`

- `options`: `<Object>`
 - `privateKey`: `<KeyObject>`
 - `publicKey`: `<KeyObject>`
- Returns: `<Buffer>`

Computes the Diffie-Hellman secret based on a `privateKey` and a `publicKey`. Both keys must have the same `asymmetricKeyType`, which must be one of `'dh'` (for Diffie-Hellman), `'ec'` (for ECDH), `'x448'`, or `'x25519'` (for ECDH-ES).

`crypto.generateKey(type, options, callback)`

- `type`: `<string>` The intended use of the generated secret key. Currently accepted values are `'hmac'` and `'aes'`.
- `options`: `<Object>`
 - `length`: `<number>` The bit length of the key to generate. This must be a value greater than 0.
 - If `type` is `'hmac'`, the minimum is 1, and the maximum length is $2^{31}-1$. If the value is not a multiple of 8, the generated key will be truncated to `Math.floor(length / 8)`.
 - If `type` is `'aes'`, the length must be one of 128, 192, or 256.
- `callback`: `<Function>`
 - `err`: `<Error>`
 - `key`: `<KeyObject>`

Asynchronously generates a new random secret key of the given `length`. The `type` will determine which validations will be performed on the `length`.

```
const {
  generateKey
} = await import('crypto');

generateKey('hmac', { length: 64 }, (err, key) => {
  if (err) throw err;
```

```

    console.log(key.export().toString('hex')); // 46e.....620
  });const {
    generateKey,
  } = require('crypto');

  generateKey('hmac', { length: 64 }, (err, key) => {
    if (err) throw err;
    console.log(key.export().toString('hex')); // 46e.....620
  });

```

crypto.generateKeyPair(type, options, callback)

- **type**: `<string>` Must be `'rsa'`, `'dsa'`, `'ec'`, `'ed25519'`, `'ed448'`, `'x25519'`, `'x448'`, or `'dh'`.
- **options**: `<Object>`
 - **modulusLength**: `<number>` Key size in bits (RSA, DSA).
 - **publicExponent**: `<number>` Public exponent (RSA). **Default**: `0x10001`.
 - **divisorLength**: `<number>` Size of `q` in bits (DSA).
 - **namedCurve**: `<string>` Name of the curve to use (EC).
 - **prime**: `<Buffer>` The prime parameter (DH).
 - **primeLength**: `<number>` Prime length in bits (DH).
 - **generator**: `<number>` Custom generator (DH). **Default**: `2`.
 - **groupName**: `<string>` Diffie-Hellman group name (DH). See `crypto.getDiffieHellman()`.
 - **publicKeyEncoding**: `<Object>` See `keyObject.export()`.
 - **privateKeyEncoding**: `<Object>` See `keyObject.export()`.
- **callback**: `<Function>`
 - **err**: `<Error>`
 - **publicKey**: `<string>` | `<Buffer>` | `<KeyObject>`
 - **privateKey**: `<string>` | `<Buffer>` | `<KeyObject>`

Generates a new asymmetric key pair of the given `type`. RSA, DSA, EC, Ed25519, Ed448, X25519, X448, and DH are currently supported.

If a `publicKeyEncoding` or `privateKeyEncoding` was specified, this function behaves as if `keyObject.export()` had been called on its result. Otherwise, the respective part of the key is returned as a `KeyObject`.

It is recommended to encode public keys as `'spki'` and private keys as `'pkcs8'` with encryption for long-term storage:

```

const {
  generateKeyPair
} = await import('crypto');

generateKeyPair('rsa', {
  modulusLength: 4096,
  publicKeyEncoding: {
    type: 'spki',
    format: 'pem'
  },
},
privateKeyEncoding: {
  type: 'pkcs8',

```



```

    format: 'pem',
    cipher: 'aes-256-cbc',
    passphrase: 'top secret'
  }
}, (err, publicKey, privateKey) => {
  // Handle errors and use the generated key pair.
});const {
  generateKeyPair,
} = require('crypto');

generateKeyPair('rsa', {
  modulusLength: 4096,
  publicKeyEncoding: {
    type: 'spki',
    format: 'pem'
  },
  privateKeyEncoding: {
    type: 'pkcs8',
    format: 'pem',
    cipher: 'aes-256-cbc',
    passphrase: 'top secret'
  }
}, (err, publicKey, privateKey) => {
  // Handle errors and use the generated key pair.
});

```

On completion, `callback` will be called with `err` set to `undefined` and `publicKey` / `privateKey` representing the generated key pair.

If this method is invoked as its `util.promisify()` ed version, it returns a `Promise` for an `Object` with `publicKey` and `privateKey` properties.

crypto.generateKeyPairSync(type, options)

- `type`: `<string>` Must be `'rsa'`, `'dsa'`, `'ec'`, `'ed25519'`, `'ed448'`, `'x25519'`, `'x448'`, or `'dh'`.
- `options`: `<Object>`
 - `modulusLength`: `<number>` Key size in bits (RSA, DSA).
 - `publicExponent`: `<number>` Public exponent (RSA). **Default:** `0x10001`.
 - `divisorLength`: `<number>` Size of `q` in bits (DSA).
 - `namedCurve`: `<string>` Name of the curve to use (EC).
 - `prime`: `<Buffer>` The prime parameter (DH).
 - `primeLength`: `<number>` Prime length in bits (DH).
 - `generator`: `<number>` Custom generator (DH). **Default:** `2`.
 - `groupName`: `<string>` Diffie-Hellman group name (DH). See `crypto.getDiffieHellman()`.
 - `publicKeyEncoding`: `<Object>` See `keyObject.export()`.
 - `privateKeyEncoding`: `<Object>` See `keyObject.export()`.
- Returns: `<Object>`
 - `publicKey`: `<string>` | `<Buffer>` | `<KeyObject>`
 - `privateKey`: `<string>` | `<Buffer>` | `<KeyObject>`

Generates a new asymmetric key pair of the given `type`. RSA, DSA, EC, Ed25519, Ed448, X25519, X448, and DH are currently supported.

If a `publicKeyEncoding` or `privateKeyEncoding` was specified, this function behaves as if `keyObject.export()` had been called on its result. Otherwise, the respective part of the key is returned as a `KeyObject`.

When encoding public keys, it is recommended to use `'spki'`. When encoding private keys, it is recommended to use `'pkcs8'` with a strong passphrase, and to keep the passphrase confidential.

```
const {
  generateKeyPairSync
} = await import('crypto');

const {
  publicKey,
  privateKey,
} = generateKeyPairSync('rsa', {
  modulusLength: 4096,
  publicKeyEncoding: {
    type: 'spki',
    format: 'pem'
  },
  privateKeyEncoding: {
    type: 'pkcs8',
    format: 'pem',
    cipher: 'aes-256-cbc',
    passphrase: 'top secret'
  }
});const {
  generateKeyPairSync,
} = require('crypto');

const {
  publicKey,
  privateKey,
} = generateKeyPairSync('rsa', {
  modulusLength: 4096,
  publicKeyEncoding: {
    type: 'spki',
    format: 'pem'
  },
  privateKeyEncoding: {
    type: 'pkcs8',
    format: 'pem',
    cipher: 'aes-256-cbc',
    passphrase: 'top secret'
  }
});
```

The return value `{ publicKey, privateKey }` represents the generated key pair. When PEM encoding was selected, the respective key will be a string, otherwise it will be a buffer containing the data encoded as DER.

crypto.generateKeySync(type, options)

- `type`: `<string>` The intended use of the generated secret key. Currently accepted values are `'hmac'` and `'aes'`.
- `options`: `<Object>`
 - `length`: `<number>` The bit length of the key to generate.
 - If `type` is `'hmac'`, the minimum is 1, and the maximum length is $2^{31}-1$. If the value is not a multiple of 8, the generated key will be truncated to `Math.floor(length / 8)`.
 - If `type` is `'aes'`, the length must be one of 128, 192, or 256.
- Returns: `<KeyObject>`

Synchronously generates a new random secret key of the given `length`. The `type` will determine which validations will be performed on the `length`.

```
const {
  generateKeySync
} = await import('crypto');

const key = generateKeySync('hmac', 64);
console.log(key.export().toString('hex')); // e89.....41e

const {
  generateKeySync,
} = require('crypto');

const key = generateKeySync('hmac', 64);
console.log(key.export().toString('hex')); // e89.....41e
```

crypto.generatePrime(size[, options[, callback]])

- `size` `<number>` The size (in bits) of the prime to generate.
- `options` `<Object>`
 - `add` `<ArrayBuffer>` | `<SharedArrayBuffer>` | `<TypedArray>` | `<Buffer>` | `<DataView>` | `<bigint>`
 - `rem` `<ArrayBuffer>` | `<SharedArrayBuffer>` | `<TypedArray>` | `<Buffer>` | `<DataView>` | `<bigint>`
 - `safe` `<boolean>` **Default:** `false`.
 - `bigint` `<boolean>` When `true`, the generated prime is returned as a `bigint`.
- `callback` `<Function>`
 - `err` `<Error>`
 - `prime` `<ArrayBuffer>` | `<bigint>`

Generates a pseudorandom prime of `size` bits.

If `options.safe` is `true`, the prime will be a safe prime -- that is, $(\text{prime} - 1) / 2$ will also be a prime.

The `options.add` and `options.rem` parameters can be used to enforce additional requirements, e.g., for Diffie-Hellman:

- If `options.add` and `options.rem` are both set, the prime will satisfy the condition that `prime % add = rem`.
- If only `options.add` is set and `options.safe` is not `true`, the prime will satisfy the condition that `prime % add = 1`.
- If only `options.add` is set and `options.safe` is set to `true`, the prime will instead satisfy the condition that `prime % add = 3`. This is necessary because `prime % add = 1` for `options.add > 2` would contradict the condition enforced by `options.safe`.
- `options.rem` is ignored if `options.add` is not given.

Both `options.add` and `options.rem` must be encoded as big-endian sequences if given as an `ArrayBuffer`, `SharedArrayBuffer`, `TypedArray`, `Buffer`, or `DataView`.

By default, the prime is encoded as a big-endian sequence of octets in an `<ArrayBuffer>`. If the `bigint` option is `true`, then a `<bigint>` is provided.

`crypto.generatePrimeSync(size[, options])`

- `size` `<number>` The size (in bits) of the prime to generate.
- `options` `<Object>`
 - `add` `<ArrayBuffer>` | `<SharedArrayBuffer>` | `<TypedArray>` | `<Buffer>` | `<DataView>` | `<bigint>`
 - `rem` `<ArrayBuffer>` | `<SharedArrayBuffer>` | `<TypedArray>` | `<Buffer>` | `<DataView>` | `<bigint>`
 - `safe` `<boolean>` **Default:** `false`.
 - `bigint` `<boolean>` When `true`, the generated prime is returned as a `bigint`.
- Returns: `<ArrayBuffer>` | `<bigint>`

Generates a pseudorandom prime of `size` bits.

If `options.safe` is `true`, the prime will be a safe prime -- that is, $(\text{prime} - 1) / 2$ will also be a prime.

The `options.add` and `options.rem` parameters can be used to enforce additional requirements, e.g., for Diffie-Hellman:

- If `options.add` and `options.rem` are both set, the prime will satisfy the condition that $\text{prime} \% \text{add} = \text{rem}$.
- If only `options.add` is set and `options.safe` is not `true`, the prime will satisfy the condition that $\text{prime} \% \text{add} = 1$.
- If only `options.add` is set and `options.safe` is set to `true`, the prime will instead satisfy the condition that $\text{prime} \% \text{add} = 3$. This is necessary because $\text{prime} \% \text{add} = 1$ for `options.add > 2` would contradict the condition enforced by `options.safe`.
- `options.rem` is ignored if `options.add` is not given.

Both `options.add` and `options.rem` must be encoded as big-endian sequences if given as an `ArrayBuffer`, `SharedArrayBuffer`, `TypedArray`, `Buffer`, or `DataView`.

By default, the prime is encoded as a big-endian sequence of octets in an `<ArrayBuffer>`. If the `bigint` option is `true`, then a `<bigint>` is provided.

`crypto.getCipherInfo(nameOrNid[, options])`

- `nameOrNid`: `<string>` | `<number>` The name or nid of the cipher to query.
- `options`: `<Object>`
 - `keyLength`: `<number>` A test key length.
 - `ivLength`: `<number>` A test IV length.
- Returns: `<Object>`
 - `name` `<string>` The name of the cipher
 - `nid` `<number>` The nid of the cipher
 - `blockSize` `<number>` The block size of the cipher in bytes. This property is omitted when `mode` is `'stream'`.
 - `ivLength` `<number>` The expected or default initialization vector length in bytes. This property is omitted if the cipher does not use an initialization vector.
 - `keyLength` `<number>` The expected or default key length in bytes.
 - `mode` `<string>` The cipher mode. One of `'cbc'`, `'ccm'`, `'cfb'`, `'ctr'`, `'ecb'`, `'gcm'`, `'ocb'`, `'ofb'`, `'stream'`, `'wrap'`, `'xts'`.

Returns information about a given cipher.

Some ciphers accept variable length keys and initialization vectors. By default, the `crypto.getCipherInfo()` method will return the default values for these ciphers. To test if a given key length or iv length is acceptable for given cipher, use the `keyLength` and `ivLength` options. If the given values are unacceptable, `undefined` will be returned.

crypto.getCiphers()

- Returns: `<string[]>` An array with the names of the supported cipher algorithms.

```
const {
  getCiphers
} = await import('crypto');

console.log(getCiphers()); // ['aes-128-cbc', 'aes-128-ccm', ...]const {
  getCiphers,
} = require('crypto');

console.log(getCiphers()); // ['aes-128-cbc', 'aes-128-ccm', ...]
```

crypto.getCurves()

- Returns: `<string[]>` An array with the names of the supported elliptic curves.

```
const {
  getCurves
} = await import('crypto');

console.log(getCurves()); // ['Oakley-EC2N-3', 'Oakley-EC2N-4', ...]const {
  getCurves,
} = require('crypto');

console.log(getCurves()); // ['Oakley-EC2N-3', 'Oakley-EC2N-4', ...]
```

crypto.getDiffieHellman(groupName)

- `groupName` `<string>`
- Returns: `<DiffieHellmanGroup>`

Creates a predefined `DiffieHellmanGroup` key exchange object. The supported groups are: `'modp1'`, `'modp2'`, `'modp5'` (defined in [RFC 2412](#), but see [Caveats](#)) and `'modp14'`, `'modp15'`, `'modp16'`, `'modp17'`, `'modp18'` (defined in [RFC 3526](#)). The returned object mimics the interface of objects created by `crypto.createDiffieHellman()`, but will not allow changing the keys (with `diffieHellman.setPublicKey()`, for example). The advantage of using this method is that the parties do not have to generate nor exchange a group modulus beforehand, saving both processor and communication time.

Example (obtaining a shared secret):

```
const {
  getDiffieHellman
} = await import('crypto');
const alice = getDiffieHellman('modp14');
const bob = getDiffieHellman('modp14');

alice.generateKeys();
bob.generateKeys();

const aliceSecret = alice.computeSecret(bob.getPublicKey(), null, 'hex');
```

```

const bobSecret = bob.computeSecret(alice.getPublicKey(), null, 'hex');

/* aliceSecret and bobSecret should be the same */
console.log(aliceSecret === bobSecret);const {
  getDiffieHellman,
} = require('crypto');

const alice = getDiffieHellman('modp14');
const bob = getDiffieHellman('modp14');

alice.generateKeys();
bob.generateKeys();

const aliceSecret = alice.computeSecret(bob.getPublicKey(), null, 'hex');
const bobSecret = bob.computeSecret(alice.getPublicKey(), null, 'hex');

/* aliceSecret and bobSecret should be the same */
console.log(aliceSecret === bobSecret);

```

crypto.getFips()

- Returns: `<number>` `1` if and only if a FIPS compliant crypto provider is currently in use, `0` otherwise. A future semver-major release may change the return type of this API to a `<boolean>`.

crypto.getHashes()

- Returns: `<string[]>` An array of the names of the supported hash algorithms, such as `'RSA-SHA256'`. Hash algorithms are also called "digest" algorithms.

```

const {
  getHashes
} = await import('crypto');

console.log(getHashes()); // ['DSA', 'DSA-SHA', 'DSA-SHA1', ...]const {
  getHashes,
} = require('crypto');

console.log(getHashes()); // ['DSA', 'DSA-SHA', 'DSA-SHA1', ...]

```

crypto.hkdf(digest, ikm, salt, info, keylen, callback)

- `digest` `<string>` The digest algorithm to use.
- `ikm` `<string>` | `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>` | `<KeyObject>` The input keying material. It must be at least one byte in length.
- `salt` `<string>` | `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>` The salt value. Must be provided but can be zero-length.
- `info` `<string>` | `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>` Additional info value. Must be provided but can be zero-length, and cannot be more than 1024 bytes.
- `keylen` `<number>` The length of the key to generate. Must be greater than 0. The maximum allowable value is `255` times the number of bytes produced by the selected digest function (e.g. `sha512` generates 64-byte hashes, making the maximum HKDF output 16320 bytes).
- `callback` `<Function>`

- `err` `<Error>`
- `derivedKey` `<ArrayBuffer>`

HKDF is a simple key derivation function defined in RFC 5869. The given `ikm`, `salt` and `info` are used with the `digest` to derive a key of `keylen` bytes.

The supplied `callback` function is called with two arguments: `err` and `derivedKey`. If an error occurs while deriving the key, `err` will be set; otherwise `err` will be `null`. The successfully generated `derivedKey` will be passed to the callback as an `<ArrayBuffer>`. An error will be thrown if any of the input arguments specify invalid values or types.

```
import { Buffer } from 'buffer';
const {
  hkdf
} = await import('crypto');

hkdf('sha512', 'key', 'salt', 'info', 64, (err, derivedKey) => {
  if (err) throw err;
  console.log(Buffer.from(derivedKey).toString('hex')); // '24156e2...5391653'
});const {
  hkdf,
} = require('crypto');
const { Buffer } = require('buffer');

hkdf('sha512', 'key', 'salt', 'info', 64, (err, derivedKey) => {
  if (err) throw err;
  console.log(Buffer.from(derivedKey).toString('hex')); // '24156e2...5391653'
});
```

`crypto.hkdfSync(digest, ikm, salt, info, keylen)`

- `digest` `<string>` The digest algorithm to use.
- `ikm` `<string>` | `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>` | `<KeyObject>` The input keying material. It must be at least one byte in length.
- `salt` `<string>` | `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>` The salt value. Must be provided but can be zero-length.
- `info` `<string>` | `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>` Additional info value. Must be provided but can be zero-length, and cannot be more than 1024 bytes.
- `keylen` `<number>` The length of the key to generate. Must be greater than 0. The maximum allowable value is 255 times the number of bytes produced by the selected digest function (e.g. `sha512` generates 64-byte hashes, making the maximum HKDF output 16320 bytes).
- Returns: `<ArrayBuffer>`

Provides a synchronous HKDF key derivation function as defined in RFC 5869. The given `ikm`, `salt` and `info` are used with the `digest` to derive a key of `keylen` bytes.

The successfully generated `derivedKey` will be returned as an `<ArrayBuffer>`.

An error will be thrown if any of the input arguments specify invalid values or types, or if the derived key cannot be generated.

```
import { Buffer } from 'buffer';
const {
  hkdfSync
} = await import('crypto');
```

```
const derivedKey = hkdfSync('sha512', 'key', 'salt', 'info', 64);
console.log(Buffer.from(derivedKey).toString('hex')); // '24156e2...5391653'const {
  hkdfSync,
} = require('crypto');
const { Buffer } = require('buffer');

const derivedKey = hkdfSync('sha512', 'key', 'salt', 'info', 64);
console.log(Buffer.from(derivedKey).toString('hex')); // '24156e2...5391653'
```

crypto.pbkdf2(password, salt, iterations, keylen, digest, callback)

- `password` `<string>` | `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>`
- `salt` `<string>` | `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>`
- `iterations` `<number>`
- `keylen` `<number>`
- `digest` `<string>`
- `callback` `<Function>`
 - `err` `<Error>`
 - `derivedKey` `<Buffer>`

Provides an asynchronous Password-Based Key Derivation Function 2 (PBKDF2) implementation. A selected HMAC digest algorithm specified by `digest` is applied to derive a key of the requested byte length (`keylen`) from the `password` , `salt` and `iterations` .

The supplied `callback` function is called with two arguments: `err` and `derivedKey` . If an error occurs while deriving the key, `err` will be set; otherwise `err` will be `null` . By default, the successfully generated `derivedKey` will be passed to the callback as a `Buffer` . An error will be thrown if any of the input arguments specify invalid values or types.

If `digest` is `null` , `'sha1'` will be used. This behavior is deprecated, please specify a `digest` explicitly.

The `iterations` argument must be a number set as high as possible. The higher the number of iterations, the more secure the derived key will be, but will take a longer amount of time to complete.

The `salt` should be as unique as possible. It is recommended that a salt is random and at least 16 bytes long. See [NIST SP 800-132](#) for details.

When passing strings for `password` or `salt` , please consider [caveats when using strings as inputs to cryptographic APIs](#) .

```
const {
  pbkdf2
} = await import('crypto');

pbkdf2('secret', 'salt', 100000, 64, 'sha512', (err, derivedKey) => {
  if (err) throw err;
  console.log(derivedKey.toString('hex')); // '3745e48...08d59ae'
});const {
  pbkdf2,
} = require('crypto');

pbkdf2('secret', 'salt', 100000, 64, 'sha512', (err, derivedKey) => {
  if (err) throw err;
```



```
console.log(derivedKey.toString('hex')); // '3745e48...08d59ae'
});
```

The `crypto.DEFAULT_ENCODING` property can be used to change the way the `derivedKey` is passed to the callback. This property, however, has been deprecated and use should be avoided.

```
import crypto from 'crypto';
crypto.DEFAULT_ENCODING = 'hex';
crypto.pbkdf2('secret', 'salt', 100000, 512, 'sha512', (err, derivedKey) => {
  if (err) throw err;
  console.log(derivedKey); // '3745e48...aa39b34'
});const crypto = require('crypto');
crypto.DEFAULT_ENCODING = 'hex';
crypto.pbkdf2('secret', 'salt', 100000, 512, 'sha512', (err, derivedKey) => {
  if (err) throw err;
  console.log(derivedKey); // '3745e48...aa39b34'
});
```

An array of supported digest functions can be retrieved using `crypto.getHashes()`.

This API uses libuv's threadpool, which can have surprising and negative performance implications for some applications; see the `UV_THREADPOOL_SIZE` documentation for more information.

`crypto.pbkdf2Sync(password, salt, iterations, keylen, digest)`

- `password` `<string>` | `<Buffer>` | `<TypedArray>` | `<DataView>`
- `salt` `<string>` | `<Buffer>` | `<TypedArray>` | `<DataView>`
- `iterations` `<number>`
- `keylen` `<number>`
- `digest` `<string>`
- Returns: `<Buffer>`

Provides a synchronous Password-Based Key Derivation Function 2 (PBKDF2) implementation. A selected HMAC digest algorithm specified by `digest` is applied to derive a key of the requested byte length (`keylen`) from the `password` , `salt` and `iterations` .

If an error occurs an `Error` will be thrown, otherwise the derived key will be returned as a `Buffer` .

If `digest` is `null` , `'sha1'` will be used. This behavior is deprecated, please specify a `digest` explicitly.

The `iterations` argument must be a number set as high as possible. The higher the number of iterations, the more secure the derived key will be, but will take a longer amount of time to complete.

The `salt` should be as unique as possible. It is recommended that a salt is random and at least 16 bytes long. See [NIST SP 800-132](#) for details.

When passing strings for `password` or `salt` , please consider [caveats when using strings as inputs to cryptographic APIs](#) .

```
const {
  pbkdf2Sync
} = await import('crypto');

const key = pbkdf2Sync('secret', 'salt', 100000, 64, 'sha512');
```

```
console.log(key.toString('hex')); // '3745e48...08d59ae'const {
  pbkdf2Sync,
} = require('crypto');

const key = pbkdf2Sync('secret', 'salt', 100000, 64, 'sha512');
console.log(key.toString('hex')); // '3745e48...08d59ae'
```

The `crypto.DEFAULT_ENCODING` property may be used to change the way the `derivedKey` is returned. This property, however, is deprecated and use should be avoided.

```
import crypto from 'crypto';
crypto.DEFAULT_ENCODING = 'hex';
const key = crypto.pbkdf2Sync('secret', 'salt', 100000, 512, 'sha512');
console.log(key); // '3745e48...aa39b34'const crypto = require('crypto');
crypto.DEFAULT_ENCODING = 'hex';
const key = crypto.pbkdf2Sync('secret', 'salt', 100000, 512, 'sha512');
console.log(key); // '3745e48...aa39b34'
```

An array of supported digest functions can be retrieved using `crypto.getHashes()`.

crypto.privateDecrypt(privateKey, buffer)

- `privateKey` `<Object> | <string> | <ArrayBuffer> | <Buffer> | <TypedArray> | <DataView> | <KeyObject> | <CryptoKey>`
 - `oaepHash` `<string>` The hash function to use for OAEP padding and MGF1. **Default:** `'sha1'`
 - `oaepLabel` `<string> | <ArrayBuffer> | <Buffer> | <TypedArray> | <DataView>` The label to use for OAEP padding. If not specified, no label is used.
 - `padding` `<crypto.constants>` An optional padding value defined in `crypto.constants`, which may be: `crypto.constants.RSA_NO_PADDING`, `crypto.constants.RSA_PKCS1_PADDING`, or `crypto.constants.RSA_PKCS1_OAEP_PADDING`.
- `buffer` `<string> | <ArrayBuffer> | <Buffer> | <TypedArray> | <DataView>`
- Returns: `<Buffer>` A new `Buffer` with the decrypted content.

Decrypts `buffer` with `privateKey`. `buffer` was previously encrypted using the corresponding public key, for example using `crypto.publicEncrypt()`.

If `privateKey` is not a `KeyObject`, this function behaves as if `privateKey` had been passed to `crypto.createPrivateKey()`. If it is an object, the `padding` property can be passed. Otherwise, this function uses `RSA_PKCS1_OAEP_PADDING`.

crypto.privateEncrypt(privateKey, buffer)

- `privateKey` `<Object> | <string> | <ArrayBuffer> | <Buffer> | <TypedArray> | <DataView> | <KeyObject> | <CryptoKey>`
 - `key` `<string> | <ArrayBuffer> | <Buffer> | <TypedArray> | <DataView> | <KeyObject> | <CryptoKey>` A PEM encoded private key.
 - `passphrase` `<string> | <ArrayBuffer> | <Buffer> | <TypedArray> | <DataView>` An optional passphrase for the private key.
 - `padding` `<crypto.constants>` An optional padding value defined in `crypto.constants`, which may be: `crypto.constants.RSA_NO_PADDING` or `crypto.constants.RSA_PKCS1_PADDING`.
 - `encoding` `<string>` The string encoding to use when `buffer`, `key`, or `passphrase` are strings.
- `buffer` `<string> | <ArrayBuffer> | <Buffer> | <TypedArray> | <DataView>`
- Returns: `<Buffer>` A new `Buffer` with the encrypted content.

Encrypts `buffer` with `privateKey`. The returned data can be decrypted using the corresponding public key, for example using `crypto.publicDecrypt()`.

If `privateKey` is not a `KeyObject`, this function behaves as if `privateKey` had been passed to `crypto.createPrivateKey()`. If it is an object, the `padding` property can be passed. Otherwise, this function uses `RSA_PKCS1_PADDING`.

`crypto.publicDecrypt(key, buffer)`

- `key` `<Object>` | `<string>` | `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>` | `<KeyObject>` | `<CryptoKey>`
 - `passphrase` `<string>` | `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>` An optional passphrase for the private key.
 - `padding` `<crypto.constants>` An optional padding value defined in `crypto.constants`, which may be: `crypto.constants.RSA_NO_PADDING` or `crypto.constants.RSA_PKCS1_PADDING`.
 - `encoding` `<string>` The string encoding to use when `buffer`, `key`, or `passphrase` are strings.
- `buffer` `<string>` | `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>`
- Returns: `<Buffer>` A new `Buffer` with the decrypted content.

Decrypts `buffer` with `key`. `buffer` was previously encrypted using the corresponding private key, for example using `crypto.privateEncrypt()`.

If `key` is not a `KeyObject`, this function behaves as if `key` had been passed to `crypto.createPublicKey()`. If it is an object, the `padding` property can be passed. Otherwise, this function uses `RSA_PKCS1_PADDING`.

Because RSA public keys can be derived from private keys, a private key may be passed instead of a public key.

`crypto.publicEncrypt(key, buffer)`

- `key` `<Object>` | `<string>` | `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>` | `<KeyObject>` | `<CryptoKey>`
 - `key` `<string>` | `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>` | `<KeyObject>` | `<CryptoKey>` A PEM encoded public or private key, `<KeyObject>`, or `<CryptoKey>`.
 - `oaepHash` `<string>` The hash function to use for OAEP padding and MGF1. **Default:** `'sha1'`
 - `oaepLabel` `<string>` | `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>` The label to use for OAEP padding. If not specified, no label is used.
 - `passphrase` `<string>` | `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>` An optional passphrase for the private key.
 - `padding` `<crypto.constants>` An optional padding value defined in `crypto.constants`, which may be: `crypto.constants.RSA_NO_PADDING`, `crypto.constants.RSA_PKCS1_PADDING`, or `crypto.constants.RSA_PKCS1_OAEP_PADDING`.
 - `encoding` `<string>` The string encoding to use when `buffer`, `key`, `oaepLabel`, or `passphrase` are strings.
- `buffer` `<string>` | `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>`
- Returns: `<Buffer>` A new `Buffer` with the encrypted content.

Encrypts the content of `buffer` with `key` and returns a new `Buffer` with encrypted content. The returned data can be decrypted using the corresponding private key, for example using `crypto.privateDecrypt()`.

If `key` is not a `KeyObject`, this function behaves as if `key` had been passed to `crypto.createPublicKey()`. If it is an object, the `padding` property can be passed. Otherwise, this function uses `RSA_PKCS1_OAEP_PADDING`.

Because RSA public keys can be derived from private keys, a private key may be passed instead of a public key.

`crypto.randomBytes(size[, callback])`

- `size` `<number>` The number of bytes to generate. The `size` must not be larger than `231 - 1`.
- `callback` `<Function>`
 - `err` `<Error>`

- `buf` `<Buffer>`
- Returns: `<Buffer>` if the `callback` function is not provided.

Generates cryptographically strong pseudorandom data. The `size` argument is a number indicating the number of bytes to generate.

If a `callback` function is provided, the bytes are generated asynchronously and the `callback` function is invoked with two arguments: `err` and `buf`. If an error occurs, `err` will be an `Error` object; otherwise it is `null`. The `buf` argument is a `Buffer` containing the generated bytes.

```
// Asynchronous
const {
  randomBytes
} = await import('crypto');

randomBytes(256, (err, buf) => {
  if (err) throw err;
  console.log(`${buf.length} bytes of random data: ${buf.toString('hex')}`);
}); // Asynchronous

const {
  randomBytes,
} = require('crypto');

randomBytes(256, (err, buf) => {
  if (err) throw err;
  console.log(`${buf.length} bytes of random data: ${buf.toString('hex')}`);
});
```

If the `callback` function is not provided, the random bytes are generated synchronously and returned as a `Buffer`. An error will be thrown if there is a problem generating the bytes.

```
// Synchronous
const {
  randomBytes
} = await import('crypto');

const buf = randomBytes(256);
console.log(
  `${buf.length} bytes of random data: ${buf.toString('hex')}`); // Synchronous

const {
  randomBytes,
} = require('crypto');

const buf = randomBytes(256);
console.log(
  `${buf.length} bytes of random data: ${buf.toString('hex')}`);
```

The `crypto.randomBytes()` method will not complete until there is sufficient entropy available. This should normally never take longer than a few milliseconds. The only time when generating the random bytes may conceivably block for a longer period of time is right after boot, when the whole system is still low on entropy.

This API uses libuv's threadpool, which can have surprising and negative performance implications for some applications; see the [UV_THREADPOOL_SIZE](#) documentation for more information.

The asynchronous version of `crypto.randomBytes()` is carried out in a single threadpool request. To minimize threadpool task length variation, partition large `randomBytes` requests when doing so as part of fulfilling a client request.

`crypto.randomFillSync(buffer[, offset][, size])`

- `buffer` `<ArrayBuffer> | <Buffer> | <TypedArray> | <DataView>` Must be supplied. The size of the provided `buffer` must not be larger than `2**31 - 1`.
- `offset` `<number>` **Default:** `0`
- `size` `<number>` **Default:** `buffer.length - offset`. The `size` must not be larger than `2**31 - 1`.
- Returns: `<ArrayBuffer> | <Buffer> | <TypedArray> | <DataView>` The object passed as `buffer` argument.

Synchronous version of `crypto.randomFill()`.

```
import { Buffer } from 'buffer';
const { randomFillSync } = await import('crypto');

const buf = Buffer.alloc(10);
console.log(randomFillSync(buf).toString('hex'));

randomFillSync(buf, 5);
console.log(buf.toString('hex'));

// The above is equivalent to the following:
randomFillSync(buf, 5, 5);
console.log(buf.toString('hex'));const { randomFillSync } = require('crypto');
const { Buffer } = require('buffer');

const buf = Buffer.alloc(10);
console.log(randomFillSync(buf).toString('hex'));

randomFillSync(buf, 5);
console.log(buf.toString('hex'));

// The above is equivalent to the following:
randomFillSync(buf, 5, 5);
console.log(buf.toString('hex'));
```

Any `ArrayBuffer`, `TypedArray` or `DataView` instance may be passed as `buffer`.

```
import { Buffer } from 'buffer';
const { randomFillSync } = await import('crypto');

const a = new Uint32Array(10);
console.log(Buffer.from(randomFillSync(a).buffer,
    a.byteOffset, a.byteLength).toString('hex'));

const b = new DataView(new ArrayBuffer(10));
console.log(Buffer.from(randomFillSync(b).buffer,
```

```

        b.byteOffset, b.byteLength).toString('hex'));

const c = new ArrayBuffer(10);
console.log(Buffer.from(randomFillSync(c)).toString('hex'));const { randomFillSync } = require('crypto');
const { Buffer } = require('buffer');

const a = new Uint32Array(10);
console.log(Buffer.from(randomFillSync(a).buffer,
        a.byteOffset, a.byteLength).toString('hex'));

const b = new DataView(new ArrayBuffer(10));
console.log(Buffer.from(randomFillSync(b).buffer,
        b.byteOffset, b.byteLength).toString('hex'));

const c = new ArrayBuffer(10);
console.log(Buffer.from(randomFillSync(c)).toString('hex'));

```

crypto.randomFill(buffer[, offset][, size], callback)

- **buffer** `<ArrayBuffer> | <Buffer> | <TypedArray> | <DataView>` Must be supplied. The size of the provided **buffer** must not be larger than `2**31 - 1`.
- **offset** `<number>` **Default:** 0
- **size** `<number>` **Default:** `buffer.length - offset`. The **size** must not be larger than `2**31 - 1`.
- **callback** `<Function>` `function(err, buf) {}`.

This function is similar to `crypto.randomBytes()` but requires the first argument to be a `Buffer` that will be filled. It also requires that a callback is passed in.

If the **callback** function is not provided, an error will be thrown.

```

import { Buffer } from 'buffer';
const { randomFill } = await import('crypto');

const buf = Buffer.alloc(10);
randomFill(buf, (err, buf) => {
  if (err) throw err;
  console.log(buf.toString('hex'));
});

randomFill(buf, 5, (err, buf) => {
  if (err) throw err;
  console.log(buf.toString('hex'));
});

// The above is equivalent to the following:
randomFill(buf, 5, 5, (err, buf) => {
  if (err) throw err;
  console.log(buf.toString('hex'));
});const { randomFill } = require('crypto');
const { Buffer } = require('buffer');

```

```

const buf = Buffer.alloc(10);
randomFill(buf, (err, buf) => {
  if (err) throw err;
  console.log(buf.toString('hex'));
});

randomFill(buf, 5, (err, buf) => {
  if (err) throw err;
  console.log(buf.toString('hex'));
});

// The above is equivalent to the following:
randomFill(buf, 5, 5, (err, buf) => {
  if (err) throw err;
  console.log(buf.toString('hex'));
});

```

Any `ArrayBuffer`, `TypedArray`, or `DataView` instance may be passed as `buffer`.

While this includes instances of `Float32Array` and `Float64Array`, this function should not be used to generate random floating-point numbers. The result may contain `+Infinity`, `-Infinity`, and `NaN`, and even if the array contains finite numbers only, they are not drawn from a uniform random distribution and have no meaningful lower or upper bounds.

```

import { Buffer } from 'buffer';
const { randomFill } = await import('crypto');

const a = new Uint32Array(10);
randomFill(a, (err, buf) => {
  if (err) throw err;
  console.log(Buffer.from(buf.buffer, buf.byteOffset, buf.byteLength)
    .toString('hex'));
});

const b = new DataView(new ArrayBuffer(10));
randomFill(b, (err, buf) => {
  if (err) throw err;
  console.log(Buffer.from(buf.buffer, buf.byteOffset, buf.byteLength)
    .toString('hex'));
});

const c = new ArrayBuffer(10);
randomFill(c, (err, buf) => {
  if (err) throw err;
  console.log(Buffer.from(buf).toString('hex'));
});const { randomFill } = require('crypto');
const { Buffer } = require('buffer');

const a = new Uint32Array(10);
randomFill(a, (err, buf) => {
  if (err) throw err;
  console.log(Buffer.from(buf.buffer, buf.byteOffset, buf.byteLength)

```

```

        .toString('hex'));
    });

    const b = new DataView(new ArrayBuffer(10));
    randomFill(b, (err, buf) => {
        if (err) throw err;
        console.log(Buffer.from(buf.buffer, buf.byteOffset, buf.byteLength)
            .toString('hex'));
    });

    const c = new ArrayBuffer(10);
    randomFill(c, (err, buf) => {
        if (err) throw err;
        console.log(Buffer.from(buf).toString('hex'));
    });

```

This API uses libuv's threadpool, which can have surprising and negative performance implications for some applications; see the [UV_THREADPOOL_SIZE](#) documentation for more information.

The asynchronous version of `crypto.randomFill()` is carried out in a single threadpool request. To minimize threadpool task length variation, partition large `randomFill` requests when doing so as part of fulfilling a client request.

`crypto.randomInt([min,]max[, callback])`

- `min` `<integer>` Start of random range (inclusive). **Default:** `0`.
- `max` `<integer>` End of random range (exclusive).
- `callback` `<Function>` `function(err, n) {}`.

Return a random integer `n` such that `min <= n < max`. This implementation avoids [modulo bias](#).

The range `(max - min)` must be less than 2^{48} . `min` and `max` must be [safe integers](#).

If the `callback` function is not provided, the random integer is generated synchronously.

```

// Asynchronous
const {
    randomInt
} = await import('crypto');

randomInt(3, (err, n) => {
    if (err) throw err;
    console.log(`Random number chosen from (0, 1, 2): ${n}`);
}); // Asynchronous

const {
    randomInt,
} = require('crypto');

randomInt(3, (err, n) => {
    if (err) throw err;
    console.log(`Random number chosen from (0, 1, 2): ${n}`);
});

```



```
// Synchronous
const {
  randomInt
} = await import('crypto');

const n = randomInt(3);
console.log(`Random number chosen from (0, 1, 2): ${n}`); // Synchronous
const {
  randomInt,
} = require('crypto');

const n = randomInt(3);
console.log(`Random number chosen from (0, 1, 2): ${n}`);
```

```
// With `min` argument
const {
  randomInt
} = await import('crypto');

const n = randomInt(1, 7);
console.log(`The dice rolled: ${n}`); // With `min` argument
const {
  randomInt,
} = require('crypto');

const n = randomInt(1, 7);
console.log(`The dice rolled: ${n}`);
```

crypto.randomUUID([options])

- **options** `<Object>`
 - **disableEntropyCache** `<boolean>` By default, to improve performance, Node.js generates and caches enough random data to generate up to 128 random UUIDs. To generate a UUID without using the cache, set **disableEntropyCache** to `true`. **Default:** `false`.
- Returns: `<string>`

Generates a random [RFC 4122](#) version 4 UUID. The UUID is generated using a cryptographic pseudorandom number generator.

crypto.scrypt(password, salt, keylen[, options], callback)

- **password** `<string> | <ArrayBuffer> | <Buffer> | <TypedArray> | <DataView>`
- **salt** `<string> | <ArrayBuffer> | <Buffer> | <TypedArray> | <DataView>`
- **keylen** `<number>`
- **options** `<Object>`
 - **cost** `<number>` CPU/memory cost parameter. Must be a power of two greater than one. **Default:** 16384.
 - **blockSize** `<number>` Block size parameter. **Default:** 8.
 - **parallelization** `<number>` Parallelization parameter. **Default:** 1.
 - **N** `<number>` Alias for **cost**. Only one of both may be specified.
 - **r** `<number>` Alias for **blockSize**. Only one of both may be specified.

- `p` `<number>` Alias for `parallelization`. Only one of both may be specified.
- `maxmem` `<number>` Memory upper bound. It is an error when (approximately) `128 * N * r > maxmem`. **Default:** `32 * 1024 * 1024`.
- `callback` `<Function>`
 - `err` `<Error>`
 - `derivedKey` `<Buffer>`

Provides an asynchronous `script` implementation. `Script` is a password-based key derivation function that is designed to be expensive computationally and memory-wise in order to make brute-force attacks unrewarding.

The `salt` should be as unique as possible. It is recommended that a salt is random and at least 16 bytes long. See [NIST SP 800-132](#) for details.

When passing strings for `password` or `salt`, please consider [caveats when using strings as inputs to cryptographic APIs](#).

The `callback` function is called with two arguments: `err` and `derivedKey`. `err` is an exception object when key derivation fails, otherwise `err` is `null`. `derivedKey` is passed to the callback as a `Buffer`.

An exception is thrown when any of the input arguments specify invalid values or types.

```
const {
  script
} = await import('crypto');

// Using the factory defaults.
script('password', 'salt', 64, (err, derivedKey) => {
  if (err) throw err;
  console.log(derivedKey.toString('hex')); // '3745e48...08d59ae'
});

// Using a custom N parameter. Must be a power of two.
script('password', 'salt', 64, { N: 1024 }, (err, derivedKey) => {
  if (err) throw err;
  console.log(derivedKey.toString('hex')); // '3745e48...aa39b34'
});const {
  script,
} = require('crypto');

// Using the factory defaults.
script('password', 'salt', 64, (err, derivedKey) => {
  if (err) throw err;
  console.log(derivedKey.toString('hex')); // '3745e48...08d59ae'
});

// Using a custom N parameter. Must be a power of two.
script('password', 'salt', 64, { N: 1024 }, (err, derivedKey) => {
  if (err) throw err;
  console.log(derivedKey.toString('hex')); // '3745e48...aa39b34'
});
```

`crypto.scriptSync(password, salt, keylen[, options])`

- `password` `<string>` | `<Buffer>` | `<TypedArray>` | `<DataView>`
- `salt` `<string>` | `<Buffer>` | `<TypedArray>` | `<DataView>`

- `keylen <number>`
- `options <Object>`
 - `cost <number>` CPU/memory cost parameter. Must be a power of two greater than one. **Default:** 16384 .
 - `blockSize <number>` Block size parameter. **Default:** 8 .
 - `parallelization <number>` Parallelization parameter. **Default:** 1 .
 - `N <number>` Alias for `cost` . Only one of both may be specified.
 - `r <number>` Alias for `blockSize` . Only one of both may be specified.
 - `p <number>` Alias for `parallelization` . Only one of both may be specified.
 - `maxmem <number>` Memory upper bound. It is an error when (approximately) $128 * N * r > \text{maxmem}$. **Default:** $32 * 1024 * 1024$.
- Returns: `<Buffer>`

Provides a synchronous `scrypt` implementation. Scrypt is a password-based key derivation function that is designed to be expensive computationally and memory-wise in order to make brute-force attacks unrewarding.

The `salt` should be as unique as possible. It is recommended that a salt is random and at least 16 bytes long. See [NIST SP 800-132](#) for details.

When passing strings for `password` or `salt` , please consider [caveats when using strings as inputs to cryptographic APIs](#) .

An exception is thrown when key derivation fails, otherwise the derived key is returned as a `Buffer` .

An exception is thrown when any of the input arguments specify invalid values or types.

```
const {
  scryptSync
} = await import('crypto');
// Using the factory defaults.

const key1 = scryptSync('password', 'salt', 64);
console.log(key1.toString('hex')); // '3745e48...08d59ae'
// Using a custom N parameter. Must be a power of two.
const key2 = scryptSync('password', 'salt', 64, { N: 1024 });
console.log(key2.toString('hex')); // '3745e48...aa39b34'const {
  scryptSync,
} = require('crypto');
// Using the factory defaults.

const key1 = scryptSync('password', 'salt', 64);
console.log(key1.toString('hex')); // '3745e48...08d59ae'
// Using a custom N parameter. Must be a power of two.
const key2 = scryptSync('password', 'salt', 64, { N: 1024 });
console.log(key2.toString('hex')); // '3745e48...aa39b34'
```

crypto.secureHeapUsed()

- Returns: `<Object>`
 - `total <number>` The total allocated secure heap size as specified using the `--secure-heap=n` command-line flag.
 - `min <number>` The minimum allocation from the secure heap as specified using the `--secure-heap-min` command-line flag.
 - `used <number>` The total number of bytes currently allocated from the secure heap.

- `utilization` `<number>` The calculated ratio of `used` to `total` allocated bytes.

`crypto.setEngine(engine[, flags])`

- `engine` `<string>`
- `flags` `<crypto.constants>` **Default:** `crypto.constants.ENGINE_METHOD_ALL`

Load and set the `engine` for some or all OpenSSL functions (selected by flags).

`engine` could be either an id or a path to the engine's shared library.

The optional `flags` argument uses `ENGINE_METHOD_ALL` by default. The `flags` is a bit field taking one of or a mix of the following flags (defined in `crypto.constants`):

- `crypto.constants.ENGINE_METHOD_RSA`
- `crypto.constants.ENGINE_METHOD_DSA`
- `crypto.constants.ENGINE_METHOD_DH`
- `crypto.constants.ENGINE_METHOD_RAND`
- `crypto.constants.ENGINE_METHOD_EC`
- `crypto.constants.ENGINE_METHOD_CIPHERS`
- `crypto.constants.ENGINE_METHOD_DIGESTS`
- `crypto.constants.ENGINE_METHOD_PKEY_METHS`
- `crypto.constants.ENGINE_METHOD_PKEY_ASN1_METHS`
- `crypto.constants.ENGINE_METHOD_ALL`
- `crypto.constants.ENGINE_METHOD_NONE`

The flags below are deprecated in OpenSSL-1.1.0.

- `crypto.constants.ENGINE_METHOD_ECDH`
- `crypto.constants.ENGINE_METHOD_ECDSA`
- `crypto.constants.ENGINE_METHOD_STORE`

`crypto.setFips(bool)`

- `bool` `<boolean>` `true` to enable FIPS mode.

Enables the FIPS compliant crypto provider in a FIPS-enabled Node.js build. Throws an error if FIPS mode is not available.

`crypto.sign(algorithm, data, key[, callback])`

- `algorithm` `<string>` | `<null>` | `<undefined>`
- `data` `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>`
- `key` `<Object>` | `<string>` | `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>` | `<KeyObject>` | `<CryptoKey>`
- `callback` `<Function>`
 - `err` `<Error>`
 - `signature` `<Buffer>`
- Returns: `<Buffer>` if the `callback` function is not provided.

Calculates and returns the signature for `data` using the given private key and algorithm. If `algorithm` is `null` or `undefined`, then the algorithm is dependent upon the key type (especially Ed25519 and Ed448).

If `key` is not a `KeyObject`, this function behaves as if `key` had been passed to `crypto.createPrivateKey()`. If it is an object, the following additional properties can be passed:

- `dsaEncoding` `<string>` For DSA and ECDSA, this option specifies the format of the generated signature. It can be one of the following:
 - `'der'` (default): DER-encoded ASN.1 signature structure encoding `(r, s)`.
 - `'ieee-p1363'`: Signature format `r || s` as proposed in IEEE-P1363.
- `padding` `<integer>` Optional padding value for RSA, one of the following:
 - `crypto.constants.RSA_PKCS1_PADDING` (default)
 - `crypto.constants.RSA_PKCS1_PSS_PADDING`
`RSA_PKCS1_PSS_PADDING` will use MGF1 with the same hash function used to sign the message as specified in section 3.1 of [RFC 4055](#).
- `saltLength` `<integer>` Salt length for when padding is `RSA_PKCS1_PSS_PADDING`. The special value `crypto.constants.RSA_PSS_SALTLEN_DIGEST` sets the salt length to the digest size, `crypto.constants.RSA_PSS_SALTLEN_MAX_SIGN` (default) sets it to the maximum permissible value.

If the `callback` function is provided this function uses libuv's threadpool.

`crypto.timingSafeEqual(a, b)`

- `a` `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>`
- `b` `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>`
- Returns: `<boolean>`

This function is based on a constant-time algorithm. Returns true if `a` is equal to `b`, without leaking timing information that would allow an attacker to guess one of the values. This is suitable for comparing HMAC digests or secret values like authentication cookies or [capability urls](#).

`a` and `b` must both be `Buffer` s, `TypedArray` s, or `DataView` s, and they must have the same byte length.

If at least one of `a` and `b` is a `TypedArray` with more than one byte per entry, such as `Uint16Array`, the result will be computed using the platform byte order.

Use of `crypto.timingSafeEqual` does not guarantee that the *surrounding* code is timing-safe. Care should be taken to ensure that the surrounding code does not introduce timing vulnerabilities.

`crypto.verify(algorithm, data, key, signature[, callback])`

- `algorithm` `<string>` | `<null>` | `<undefined>`
- `data` `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>`
- `key` `<Object>` | `<string>` | `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>` | `<KeyObject>` | `<CryptoKey>`
- `signature` `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>`
- `callback` `<Function>`
 - `err` `<Error>`
 - `result` `<boolean>`
- Returns: `<boolean>` `true` or `false` depending on the validity of the signature for the data and public key if the `callback` function is not provided.

Verifies the given signature for `data` using the given key and algorithm. If `algorithm` is `null` or `undefined`, then the algorithm is dependent upon the key type (especially Ed25519 and Ed448).

If `key` is not a `KeyObject`, this function behaves as if `key` had been passed to `crypto.createPublicKey()`. If it is an object, the following additional properties can be passed:

- `dsaEncoding` `<string>` For DSA and ECDSA, this option specifies the format of the signature. It can be one of the following:
 - `'der'` (default): DER-encoded ASN.1 signature structure encoding `(r, s)`.
 - `'ieee-p1363'`: Signature format `r || s` as proposed in IEEE-P1363.
- `padding` `<integer>` Optional padding value for RSA, one of the following:
 - `crypto.constants.RSA_PKCS1_PADDING` (default)
 - `crypto.constants.RSA_PKCS1_PSS_PADDING`
`RSA_PKCS1_PSS_PADDING` will use MGF1 with the same hash function used to sign the message as specified in section 3.1 of [RFC 4055](#).
- `saltLength` `<integer>` Salt length for when padding is `RSA_PKCS1_PSS_PADDING`. The special value `crypto.constants.RSA_PSS_SALTLEN_DIGEST` sets the salt length to the digest size, `crypto.constants.RSA_PSS_SALTLEN_MAX_SIGN` (default) sets it to the maximum permissible value.

The `signature` argument is the previously calculated signature for the `data`.

Because public keys can be derived from private keys, a private key or a public key may be passed for `key`.

If the `callback` function is provided this function uses libuv's threadpool.

crypto.webcrypto

Type: `<Crypto>` An implementation of the Web Crypto API standard.

See the [Web Crypto API documentation](#) for details.

Notes

Using strings as inputs to cryptographic APIs

For historical reasons, many cryptographic APIs provided by Node.js accept strings as inputs where the underlying cryptographic algorithm works on byte sequences. These instances include plaintexts, ciphertexts, symmetric keys, initialization vectors, passphrases, salts, authentication tags, and additional authenticated data.

When passing strings to cryptographic APIs, consider the following factors.

- Not all byte sequences are valid UTF-8 strings. Therefore, when a byte sequence of length `n` is derived from a string, its entropy is generally lower than the entropy of a random or pseudorandom `n` byte sequence. For example, no UTF-8 string will result in the byte sequence `c0 af`. Secret keys should almost exclusively be random or pseudorandom byte sequences.
- Similarly, when converting random or pseudorandom byte sequences to UTF-8 strings, subsequences that do not represent valid code points may be replaced by the Unicode replacement character (`U+FFFD`). The byte representation of the resulting Unicode string may, therefore, not be equal to the byte sequence that the string was created from.

```
const original = [0xc0, 0xaf];
const bytesAsString = Buffer.from(original).toString('utf8');
const stringAsBytes = Buffer.from(bytesAsString, 'utf8');
console.log(stringAsBytes);
// Prints '<Buffer ef bf bd ef bf bd>'.
```

The outputs of ciphers, hash functions, signature algorithms, and key derivation functions are pseudorandom byte sequences and should not be used as Unicode strings.

- When strings are obtained from user input, some Unicode characters can be represented in multiple equivalent ways that result in different byte sequences. For example, when passing a user passphrase to a key derivation function, such as PBKDF2 or scrypt, the result of the key derivation function depends on whether the string uses composed or decomposed characters. Node.js does not normalize character representations. Developers should consider using `String.prototype.normalize()` on user inputs before passing them to cryptographic APIs.

Legacy streams API (prior to Node.js 0.10)

The `Crypto` module was added to Node.js before there was the concept of a unified Stream API, and before there were `Buffer` objects for handling binary data. As such, the many of the `crypto` defined classes have methods not typically found on other Node.js classes that implement the `streams` API (e.g. `update()`, `final()`, or `digest()`). Also, many methods accepted and returned `'latin1'` encoded strings by default rather than `Buffer`s. This default was changed after Node.js v0.8 to use `Buffer` objects by default instead.

Recent ECDH changes

Usage of `ECDH` with non-dynamically generated key pairs has been simplified. Now, `ecdh.setPrivateKey()` can be called with a preselected private key and the associated public point (key) will be computed and stored in the object. This allows code to only store and provide the private part of the EC key pair. `ecdh.setPrivateKey()` now also validates that the private key is valid for the selected curve.

The `ecdh.setPublicKey()` method is now deprecated as its inclusion in the API is not useful. Either a previously stored private key should be set, which automatically generates the associated public key, or `ecdh.generateKeys()` should be called. The main drawback of using `ecdh.setPublicKey()` is that it can be used to put the ECDH key pair into an inconsistent state.

Support for weak or compromised algorithms

The `crypto` module still supports some algorithms which are already compromised and are not currently recommended for use. The API also allows the use of ciphers and hashes with a small key size that are too weak for safe use.

Users should take full responsibility for selecting the crypto algorithm and key size according to their security requirements.

Based on the recommendations of [NIST SP 800-131A](#) :

- MD5 and SHA-1 are no longer acceptable where collision resistance is required such as digital signatures.
- The key used with RSA, DSA, and DH algorithms is recommended to have at least 2048 bits and that of the curve of ECDSA and ECDH at least 224 bits, to be safe to use for several years.
- The DH groups of `modp1`, `modp2` and `modp5` have a key size smaller than 2048 bits and are not recommended.

See the reference for other recommendations and details.

CCM mode

CCM is one of the supported [AEAD algorithms](#). Applications which use this mode must adhere to certain restrictions when using the cipher API:

- The authentication tag length must be specified during cipher creation by setting the `authTagLength` option and must be one of 4, 6, 8, 10, 12, 14 or 16 bytes.
- The length of the initialization vector (nonce) `N` must be between 7 and 13 bytes ($7 \leq N \leq 13$).
- The length of the plaintext is limited to $2^{31} - (8 * (15 - N))$ bytes.
- When decrypting, the authentication tag must be set via `setAuthTag()` before calling `update()`. Otherwise, decryption will fail and `final()` will throw an error in compliance with section 2.6 of [RFC 3610](#).
- Using stream methods such as `write(data)`, `end(data)` or `pipe()` in CCM mode might fail as CCM cannot handle more than one chunk of data per instance.
- When passing additional authenticated data (AAD), the length of the actual message in bytes must be passed to `setAAD()` via the `plaintextLength` option. Many crypto libraries include the authentication tag in the ciphertext, which means that they produce

ciphertexts of the length `plaintextLength + authTagLength`. Node.js does not include the authentication tag, so the ciphertext length is always `plaintextLength`. This is not necessary if no AAD is used.

- As CCM processes the whole message at once, `update()` must be called exactly once.
- Even though calling `update()` is sufficient to encrypt/decrypt the message, applications *must* call `final()` to compute or verify the authentication tag.

```
import { Buffer } from 'buffer';
const {
  createCipheriv,
  createDecipheriv,
  randomBytes
} = await import('crypto');

const key = 'keykeykeykeykeykeykeykey';
const nonce = randomBytes(12);

const aad = Buffer.from('0123456789', 'hex');

const cipher = createCipheriv('aes-192-ccm', key, nonce, {
  authTagLength: 16
});
const plaintext = 'Hello world';
cipher.setAAD(aad, {
  plaintextLength: Buffer.byteLength(plaintext)
});
const ciphertext = cipher.update(plaintext, 'utf8');
cipher.final();
const tag = cipher.getAuthTag();

// Now transmit { ciphertext, nonce, tag }.

const decipher = createDecipheriv('aes-192-ccm', key, nonce, {
  authTagLength: 16
});
decipher.setAuthTag(tag);
decipher.setAAD(aad, {
  plaintextLength: ciphertext.length
});
const receivedPlaintext = decipher.update(ciphertext, null, 'utf8');

try {
  decipher.final();
} catch (err) {
  console.error('Authentication failed!');
  return;
}

console.log(receivedPlaintext);const {
  createCipheriv,
  createDecipheriv,
  randomBytes,
```



```

} = require('crypto');
const { Buffer } = require('buffer');

const key = 'keykeykeykeykeykeykeykey';
const nonce = randomBytes(12);

const aad = Buffer.from('0123456789', 'hex');

const cipher = createCipheriv('aes-192-ccm', key, nonce, {
  authTagLength: 16
});
const plaintext = 'Hello world';
cipher.setAAD(aad, {
  plaintextLength: Buffer.byteLength(plaintext)
});
const ciphertext = cipher.update(plaintext, 'utf8');
cipher.final();
const tag = cipher.getAuthTag();

// Now transmit { ciphertext, nonce, tag }.

const decipher = createDecipheriv('aes-192-ccm', key, nonce, {
  authTagLength: 16
});
decipher.setAuthTag(tag);
decipher.setAAD(aad, {
  plaintextLength: ciphertext.length
});
const receivedPlaintext = decipher.update(ciphertext, null, 'utf8');

try {
  decipher.final();
} catch (err) {
  console.error('Authentication failed!');
  return;
}

console.log(receivedPlaintext);

```

Crypto constants

The following constants exported by `crypto.constants` apply to various uses of the `crypto`, `tls`, and `https` modules and are generally specific to OpenSSL.

OpenSSL options

Constant	Description
<code>SSL_OP_ALL</code>	Applies multiple bug workarounds within OpenSSL. See https://www.openssl.org/docs/man1.0.2/ssl/SSL_CTX_set_options.html for detail.
<code>SSL_OP_ALLOW_NO_DHE_KEY</code>	Instructs OpenSSL to allow a non-[EC]DHE-based key exchange mode for TLS v1.3

X	
SSL_OP_ALLOW_UNSAFE_LEGACY_RENEGOTIATION	Allows legacy insecure renegotiation between OpenSSL and unpatched clients or servers. See https://www.openssl.org/docs/man1.0.2/ssl/SSL_CTX_set_options.html .
SSL_OP_CIPHER_SERVER_PREFERENCE	Attempts to use the server's preferences instead of the client's when selecting a cipher. Behavior depends on protocol version. See https://www.openssl.org/docs/man1.0.2/ssl/SSL_CTX_set_options.html .
SSL_OP_CISCO_ANYCONNECT	Instructs OpenSSL to use Cisco's "speshul" version of DTLS_BAD_VER.
SSL_OP_COOKIE_EXCHANGE	Instructs OpenSSL to turn on cookie exchange.
SSL_OP_CRYPTOPRO_TLSEXT_BUG	Instructs OpenSSL to add server-hello extension from an early version of the cryptopro draft.
SSL_OP_DONT_INSERT_EMPTY_FRAGMENTS	Instructs OpenSSL to disable a SSL 3.0/TLS 1.0 vulnerability workaround added in OpenSSL 0.9.6d.
SSL_OP_EPHEMERAL_RSA	Instructs OpenSSL to always use the tmp_rsa key when performing RSA operations.
SSL_OP_LEGACY_SERVER_CONNECT	Allows initial connection to servers that do not support RI.
SSL_OP_MICROSOFT_BIG_SSLV3_BUFFER	
SSL_OP_MICROSOFT_SESS_ID_BUG	
SSL_OP_MSIE_SSLV2_RSA_PADDING	Instructs OpenSSL to disable the workaround for a man-in-the-middle protocol-version vulnerability in the SSL 2.0 server implementation.
SSL_OP_NETSCAPE_CA_DN_BUG	
SSL_OP_NETSCAPE_CHALLENGE_BUG	
SSL_OP_NETSCAPE_DEMO_CIPHER_CHANGE_BUG	
SSL_OP_NETSCAPE_REUSE_CIPHER_CHANGE_BUG	
SSL_OP_NO_COMPRESSION	Instructs OpenSSL to disable support for SSL/TLS compression.
SSL_OP_NO_ENCRYPT_THEN_MAC	Instructs OpenSSL to disable encrypt-then-MAC.
SSL_OP_NO_QUERY_MTU	
SSL_OP_NO_RENEGOTIATION	Instructs OpenSSL to disable renegotiation.
SSL_OP_NO_SESSION_RESUMPTION_ON_RENEGOTIATION	Instructs OpenSSL to always start a new session when performing renegotiation.
SSL_OP_NO_SSLv2	Instructs OpenSSL to turn off SSL v2

SSL_OP_NO_SSLv3	Instructs OpenSSL to turn off SSL v3
SSL_OP_NO_TICKET	Instructs OpenSSL to disable use of RFC4507bis tickets.
SSL_OP_NO_TLSv1	Instructs OpenSSL to turn off TLS v1
SSL_OP_NO_TLSv1_1	Instructs OpenSSL to turn off TLS v1.1
SSL_OP_NO_TLSv1_2	Instructs OpenSSL to turn off TLS v1.2
SSL_OP_NO_TLSv1_3	Instructs OpenSSL to turn off TLS v1.3
SSL_OP_PKCS1_CHECK_1	
SSL_OP_PKCS1_CHECK_2	
SSL_OP_PRIORITIZE_CHACHA	Instructs OpenSSL server to prioritize ChaCha20Poly1305 when client does. This option has no effect if <code>SSL_OP_CIPHER_SERVER_PREFERENCE</code> is not enabled.
SSL_OP_SINGLE_DH_USE	Instructs OpenSSL to always create a new key when using temporary/ephemeral DH parameters.
SSL_OP_SINGLE_ECDH_USE	Instructs OpenSSL to always create a new key when using temporary/ephemeral ECDH parameters.
SSL_OP_SSLEAY_080_CLIENT_DH_BUG	
SSL_OP_SSLREF2_REUSE_CERT_TYPE_BUG	
SSL_OP_TLS_BLOCK_PADDING_BUG	
SSL_OP_TLS_D5_BUG	
SSL_OP_TLS_ROLLBACK_BUG	Instructs OpenSSL to disable version rollback attack detection.

OpenSSL engine constants

Constant	Description
ENGINE_METHOD_RSA	Limit engine usage to RSA
ENGINE_METHOD_DSA	Limit engine usage to DSA
ENGINE_METHOD_DH	Limit engine usage to DH
ENGINE_METHOD_RAND	Limit engine usage to RAND
ENGINE_METHOD_EC	Limit engine usage to EC
ENGINE_METHOD_CIPHERS	Limit engine usage to CIPHERS
ENGINE_METHOD_DIGESTS	Limit engine usage to DIGESTS
ENGINE_METHOD_PKEY_METHODS	Limit engine usage to PKEY_METHODS
ENGINE_METHOD_PKEY_ASN1_METHODS	Limit engine usage to PKEY_ASN1_METHODS
ENGINE_METHOD_ALL	

ENGINE_METHOD_NONE	
--------------------	--

Other OpenSSL constants

See the [list of SSL OP Flags](#) for details.

Constant	Description
DH_CHECK_P_NOT_SAFE_PRIME	
DH_CHECK_P_NOT_PRIME	
DH_UNABLE_TO_CHECK_GENERATOR	
DH_NOT_SUITABLE_GENERATOR	
ALPN_ENABLED	
RSA_PKCS1_PADDING	
RSA_SSLV23_PADDING	
RSA_NO_PADDING	
RSA_PKCS1_OAEP_PADDING	
RSA_X931_PADDING	
RSA_PKCS1_PSS_PADDING	
RSA_PSS_SALTLEN_DIGEST	Sets the salt length for RSA_PKCS1_PSS_PADDING to the digest size when signing or verifying.
RSA_PSS_SALTLEN_MAX_SIGN	Sets the salt length for RSA_PKCS1_PSS_PADDING to the maximum permissible value when signing data.
RSA_PSS_SALTLEN_AUTO	Causes the salt length for RSA_PKCS1_PSS_PADDING to be determined automatically when verifying a signature.
POINT_CONVERSION_COMPRESSED	
POINT_CONVERSION_UNCOMPRESSED	
POINT_CONVERSION_HYBRID	

Node.js crypto constants

Constant	Description
defaultCoreCipherList	Specifies the built-in default cipher list used by Node.js.
defaultCipherList	Specifies the active default cipher list used by the current Node.js process.

Debugger

Node.js includes a command-line debugging utility. To use it, start Node.js with the `inspect` argument followed by the path to the script to debug.

```
$ node inspect myscript.js
< Debugger listening on ws://127.0.0.1:9229/621111f9-ffcb-4e82-b718-48a145fa5db8
< For help, see: https://nodejs.org/en/docs/inspector
<
< Debugger attached.
<
ok
Break on start in myscript.js:2
  1 // myscript.js
> 2 global.x = 5;
  3 setTimeout(() => {
  4   debugger;
debug>
```

The Node.js debugger client is not a full-featured debugger, but simple step and inspection are possible.

Inserting the statement `debugger;` into the source code of a script will enable a breakpoint at that position in the code:

```
// myscript.js
global.x = 5;
setTimeout(() => {
  debugger;
  console.log('world');
}, 1000);
console.log('hello');
```

Once the debugger is run, a breakpoint will occur at line 3:

```
$ node inspect myscript.js
< Debugger listening on ws://127.0.0.1:9229/621111f9-ffcb-4e82-b718-48a145fa5db8
< For help, see: https://nodejs.org/en/docs/inspector
<
< Debugger attached.
<
ok
Break on start in myscript.js:2
  1 // myscript.js
> 2 global.x = 5;
  3 setTimeout(() => {
  4   debugger;
debug> cont
< hello
<
```

```

break in myscript.js:4
  2 global.x = 5;
  3 setTimeout(() => {
> 4   debugger;
  5   console.log('world');
  6 }, 1000);
debug> next
break in myscript.js:5
  3 setTimeout(() => {
  4   debugger;
> 5   console.log('world');
  6 }, 1000);
  7 console.log('hello');
debug> repl
Press Ctrl+C to leave debug repl
> x
5
> 2 + 2
4
debug> next
< world
<
break in myscript.js:6
  4   debugger;
  5   console.log('world');
> 6 }, 1000);
  7 console.log('hello');
  8
debug> .exit
$

```

The `repl` command allows code to be evaluated remotely. The `next` command steps to the next line. Type `help` to see what other commands are available.

Pressing `enter` without typing a command will repeat the previous debugger command.

Watchers

It is possible to watch expression and variable values while debugging. On every breakpoint, each expression from the watchers list will be evaluated in the current context and displayed immediately before the breakpoint's source code listing.

To begin watching an expression, type `watch('my_expression')`. The command `watchers` will print the active watchers. To remove a watcher, type `unwatch('my_expression')`.

Command reference

Stepping

- `cont`, `c`: Continue execution
- `next`, `n`: Step next

- `step, s`: Step in
- `out, o`: Step out
- `pause`: Pause running code (like pause button in Developer Tools)

Breakpoints

- `setBreakpoint()`, `sb()`: Set breakpoint on current line
- `setBreakpoint(line)`, `sb(line)`: Set breakpoint on specific line
- `setBreakpoint('fn()', sb(...))`: Set breakpoint on a first statement in function's body
- `setBreakpoint('script.js', 1)`, `sb(...)`: Set breakpoint on first line of `script.js`
- `setBreakpoint('script.js', 1, 'num < 4')`, `sb(...)`: Set conditional breakpoint on first line of `script.js` that only breaks when `num < 4` evaluates to `true`
- `clearBreakpoint('script.js', 1)`, `cb(...)`: Clear breakpoint in `script.js` on line 1

It is also possible to set a breakpoint in a file (module) that is not loaded yet:

```
$ node inspect main.js
< Debugger listening on ws://127.0.0.1:9229/48a5b28a-550c-471b-b5e1-d13dd7165df9
< For help, see: https://nodejs.org/en/docs/inspector
<
< Debugger attached.
<
ok
Break on start in main.js:1
> 1 const mod = require('./mod.js');
   2 mod.hello();
   3 mod.hello();
debug> setBreakpoint('mod.js', 22)
Warning: script 'mod.js' was not loaded yet.
debug> c
break in mod.js:22
 20 // USE OR OTHER DEALINGS IN THE SOFTWARE.
 21
>22 exports.hello = function() {
 23   return 'hello from module';
 24 };
debug>
```

It is also possible to set a conditional breakpoint that only breaks when a given expression evaluates to `true`:

```
$ node inspect main.js
< Debugger listening on ws://127.0.0.1:9229/ce24daa8-3816-44d4-b8ab-8273c8a66d35
< For help, see: https://nodejs.org/en/docs/inspector
< Debugger attached.
Break on start in main.js:7
   5 }
   6
> 7 addOne(10);
   8 addOne(-1);
   9
```

```

debug> setBreakpoint('main.js', 4, 'num < 0')
  1 'use strict';
  2
  3 function addOne(num) {
> 4   return num + 1;
  5 }
  6
  7 addOne(10);
  8 addOne(-1);
  9
debug> cont
break in main.js:4
  2
  3 function addOne(num) {
> 4   return num + 1;
  5 }
  6
debug> exec('num')
-1
debug>

```

Information

- `backtrace`, `bt` : Print backtrace of current execution frame
- `list(5)` : List scripts source code with 5 line context (5 lines before and after)
- `watch(expr)` : Add expression to watch list
- `unwatch(expr)` : Remove expression from watch list
- `watchers` : List all watchers and their values (automatically listed on each breakpoint)
- `repl` : Open debugger's repl for evaluation in debugging script's context
- `exec expr` : Execute an expression in debugging script's context

Execution control

- `run` : Run script (automatically runs on debugger's start)
- `restart` : Restart script
- `kill` : Kill script

Various

- `scripts` : List all loaded scripts
- `version` : Display V8's version

Advanced usage

V8 inspector integration for Node.js

V8 Inspector integration allows attaching Chrome DevTools to Node.js instances for debugging and profiling. It uses the [Chrome DevTools Protocol](#).

V8 Inspector can be enabled by passing the `--inspect` flag when starting a Node.js application. It is also possible to supply a custom port with that flag, e.g. `--inspect=9222` will accept DevTools connections on port 9222.

To break on the first line of the application code, pass the `--inspect-brk` flag instead of `--inspect`.

```
$ node --inspect index.js
Debugger listening on ws://127.0.0.1:9229/dc9010dd-f8b8-4ac5-a510-c1a114ec7d29
For help, see: https://nodejs.org/en/docs/inspector
```

(In the example above, the UUID `dc9010dd-f8b8-4ac5-a510-c1a114ec7d29` at the end of the URL is generated on the fly, it varies in different debugging sessions.)

If the Chrome browser is older than 66.0.3345.0, use `inspector.html` instead of `js_app.html` in the above URL.

Chrome DevTools doesn't support debugging `worker threads` yet. `ndb` can be used to debug them.

Deprecated APIs

Node.js APIs might be deprecated for any of the following reasons:

- Use of the API is unsafe.
- An improved alternative API is available.
- Breaking changes to the API are expected in a future major release.

Node.js uses three kinds of Deprecations:

- Documentation-only
- Runtime
- End-of-Life

A Documentation-only deprecation is one that is expressed only within the Node.js API docs. These generate no side-effects while running Node.js. Some Documentation-only deprecations trigger a runtime warning when launched with `--pending-deprecation` flag (or its alternative, `NODE_PENDING_DEPRECATION=1` environment variable), similarly to Runtime deprecations below. Documentation-only deprecations that support that flag are explicitly labeled as such in the [list of Deprecated APIs](#).

A Runtime deprecation will, by default, generate a process warning that will be printed to `stderr` the first time the deprecated API is used. When the `--throw-deprecation` command-line flag is used, a Runtime deprecation will cause an error to be thrown.

An End-of-Life deprecation is used when functionality is or will soon be removed from Node.js.

Revoking deprecations

Occasionally, the deprecation of an API might be reversed. In such situations, this document will be updated with information relevant to the decision. However, the deprecation identifier will not be modified.

List of deprecated APIs

DEP0001: `http.OutgoingMessage.prototype.flush`

Type: End-of-Life

`OutgoingMessage.prototype.flush()` has been removed. Use `OutgoingMessage.prototype.flushHeaders()` instead.

DEP0002: `require('_linklist')`

Type: End-of-Life

The `_linklist` module is deprecated. Please use a userland alternative.

DEP0003: `_writableState.buffer`

Type: End-of-Life

The `_writableState.buffer` has been removed. Use `_writableState.getBuffer()` instead.

DEP0004: `CryptoStream.prototype.readyState`

Type: End-of-Life

The `CryptoStream.prototype.readyState` property was removed.

DEP0005: `Buffer()` constructor

Type: Runtime (supports `--pending-deprecation`)

The `Buffer()` function and `new Buffer()` constructor are deprecated due to API usability issues that can lead to accidental security issues.

As an alternative, use one of the following methods of constructing `Buffer` objects:

- `Buffer.alloc(size[, fill[, encoding]])` : Create a `Buffer` with *initialized* memory.
- `Buffer.allocUnsafe(size)` : Create a `Buffer` with *uninitialized* memory.
- `Buffer.allocUnsafeSlow(size)` : Create a `Buffer` with *uninitialized* memory.
- `Buffer.from(array)` : Create a `Buffer` with a copy of `array`
- `Buffer.from(arrayBuffer[, byteOffset[, length]])` - Create a `Buffer` that wraps the given `arrayBuffer` .
- `Buffer.from(buffer)` : Create a `Buffer` that copies `buffer` .
- `Buffer.from(string[, encoding])` : Create a `Buffer` that copies `string` .

Without `--pending-deprecation` , runtime warnings occur only for code not in `node_modules` . This means there will not be deprecation warnings for `Buffer()` usage in dependencies. With `--pending-deprecation` , a runtime warning results no matter where the `Buffer()` usage occurs.

DEP0006: `child_process options.customFds`

Type: End-of-Life

Within the `child_process` module's `spawn()` , `fork()` , and `exec()` methods, the `options.customFds` option is deprecated. The `options.stdio` option should be used instead.

DEP0007: Replace `cluster worker.suicide` with `worker.exitedAfterDisconnect`

Type: End-of-Life

In an earlier version of the Node.js `cluster` , a boolean property with the name `suicide` was added to the `worker` object. The intent of this property was to provide an indication of how and why the `worker` instance exited. In Node.js 6.0.0, the old property was deprecated and replaced with a new `worker.exitedAfterDisconnect` property. The old property name did not precisely describe the actual semantics and was unnecessarily emotion-laden.

DEP0008: `require('constants')`

Type: Documentation-only

The `constants` module is deprecated. When requiring access to constants relevant to specific Node.js builtin modules, developers should instead refer to the `constants` property exposed by the relevant module. For instance, `require('fs').constants` and `require('os').constants`.

DEP0009: `crypto.pbkdf2` without digest

Type: End-of-Life

Use of the `crypto.pbkdf2()` API without specifying a digest was deprecated in Node.js 6.0 because the method defaulted to using the non-recommended `'SHA1'` digest. Previously, a deprecation warning was printed. Starting in Node.js 8.0.0, calling `crypto.pbkdf2()` or `crypto.pbkdf2Sync()` with `digest` set to `undefined` will throw a `TypeError`.

Beginning in Node.js v11.0.0, calling these functions with `digest` set to `null` would print a deprecation warning to align with the behavior when `digest` is `undefined`.

Now, however, passing either `undefined` or `null` will throw a `TypeError`.

DEP0010: `crypto.createCredentials`

Type: End-of-Life

The `crypto.createCredentials()` API was removed. Please use `tls.createSecureContext()` instead.

DEP0011: `crypto.Credentials`

Type: End-of-Life

The `crypto.Credentials` class was removed. Please use `tls.SecureContext` instead.

DEP0012: `Domain.dispose`

Type: End-of-Life

`Domain.dispose()` has been removed. Recover from failed I/O actions explicitly via error event handlers set on the domain instead.

DEP0013: `fs` asynchronous function without callback

Type: End-of-Life

Calling an asynchronous function without a callback throws a `TypeError` in Node.js 10.0.0 onwards. See <https://github.com/nodejs/node/pull/12562>.

DEP0014: `fs.read` legacy String interface

Type: End-of-Life

The `fs.read()` legacy `String` interface is deprecated. Use the `Buffer` API as mentioned in the documentation instead.

DEP0015: `fs.readSync` legacy String interface

Type: End-of-Life

The `fs.readSync()` legacy `String` interface is deprecated. Use the `Buffer` API as mentioned in the documentation instead.

DEP0016: `GLOBAL` / `root`

Type: End-of-Life

The `GLOBAL` and `root` aliases for the `global` property were deprecated in Node.js 6.0.0 and have since been removed.

DEP0017: Intl.v8BreakIterator

Type: End-of-Life

`Intl.v8BreakIterator` was a non-standard extension and has been removed. See [Intl.Segmenter](#).

DEP0018: Unhandled promise rejections

Type: End-of-Life

Unhandled promise rejections are deprecated. By default, promise rejections that are not handled terminate the Node.js process with a non-zero exit code. To change the way Node.js treats unhandled rejections, use the `--unhandled-rejections` command-line option.

DEP0019: require('.') resolved outside directory

Type: End-of-Life

In certain cases, `require('.')` could resolve outside the package directory. This behavior has been removed.

DEP0020: Server.connections

Type: End-of-Life

The `Server.connections` property was deprecated in Node.js v0.9.7 and has been removed. Please use the `Server.getConnections()` method instead.

DEP0021: Server.listenFD

Type: End-of-Life

The `Server.listenFD()` method was deprecated and removed. Please use `Server.listen({fd: <number>})` instead.

DEP0022: os.tmpDir()

Type: End-of-Life

The `os.tmpDir()` API was deprecated in Node.js 7.0.0 and has since been removed. Please use `os.tmpdir()` instead.

DEP0023: os.getNetworkInterfaces()

Type: End-of-Life

The `os.getNetworkInterfaces()` method is deprecated. Please use the `os.networkInterfaces()` method instead.

DEP0024: REPLServer.prototype.convertToContext()

Type: End-of-Life

The `REPLServer.prototype.convertToContext()` API has been removed.

DEP0025: require('sys')

Type: Runtime

The `sys` module is deprecated. Please use the `util` module instead.

DEP0026: `util.print()`

Type: End-of-Life

`util.print()` has been removed. Please use `console.log()` instead.

DEP0027: `util.puts()`

Type: End-of-Life

`util.puts()` has been removed. Please use `console.log()` instead.

DEP0028: `util.debug()`

Type: End-of-Life

`util.debug()` has been removed. Please use `console.error()` instead.

DEP0029: `util.error()`

Type: End-of-Life

`util.error()` has been removed. Please use `console.error()` instead.

DEP0030: `SlowBuffer`

Type: Documentation-only

The `SlowBuffer` class is deprecated. Please use `Buffer.allocUnsafeSlow(size)` instead.

DEP0031: `ecdh.setPublicKey()`

Type: Documentation-only

The `ecdh.setPublicKey()` method is now deprecated as its inclusion in the API is not useful.

DEP0032: `domain` module

Type: Documentation-only

The `domain` module is deprecated and should not be used.

DEP0033: `EventEmitter.listenerCount()`

Type: Documentation-only

The `events.listenerCount(emitter, eventName)` API is deprecated. Please use `emitter.listenerCount(eventName)` instead.

DEP0034: `fs.exists(path, callback)`

Type: Documentation-only

The `fs.exists(path, callback)` API is deprecated. Please use `fs.stat()` or `fs.access()` instead.

DEP0035: `fs.lchmod(path, mode, callback)`

Type: Documentation-only

The `fs.lchmod(path, mode, callback)` API is deprecated.

DEP0036: `fs.lchmodSync(path, mode)`

Type: Documentation-only

The `fs.lchmodSync(path, mode)` API is deprecated.

DEP0037: `fs.lchown(path, uid, gid, callback)`

Type: Deprecation revoked

The `fs.lchown(path, uid, gid, callback)` API was deprecated. The deprecation was revoked because the requisite supporting APIs were added in libuv.

DEP0038: `fs.lchownSync(path, uid, gid)`

Type: Deprecation revoked

The `fs.lchownSync(path, uid, gid)` API was deprecated. The deprecation was revoked because the requisite supporting APIs were added in libuv.

DEP0039: `require.extensions`

Type: Documentation-only

The `require.extensions` property is deprecated.

DEP0040: `punycode` module

Type: Documentation-only (supports `--pending-deprecation`)

The `punycode` module is deprecated. Please use a userland alternative instead.

DEP0041: `NODE_REPL_HISTORY_FILE` environment variable

Type: End-of-Life

The `NODE_REPL_HISTORY_FILE` environment variable was removed. Please use `NODE_REPL_HISTORY` instead.

DEP0042: `tls.CryptoStream`

Type: End-of-Life

The `tls.CryptoStream` class was removed. Please use `tls.TLSSocket` instead.

DEP0043: `tls.SecurePair`

Type: Documentation-only

The `tls.SecurePair` class is deprecated. Please use `tls.TLSSocket` instead.

DEP0044: `util.isArray()`

Type: Documentation-only

The `util.isArray()` API is deprecated. Please use `Array.isArray()` instead.

DEP0045: `util.isBoolean()`

Type: Documentation-only

The `util.isBoolean()` API is deprecated.

DEP0046: `util.isBuffer()`

Type: Documentation-only

The `util.isBuffer()` API is deprecated. Please use `Buffer.isBuffer()` instead.

DEP0047: `util.isDate()`

Type: Documentation-only

The `util.isDate()` API is deprecated.

DEP0048: `util.isError()`

Type: Documentation-only

The `util.isError()` API is deprecated.

DEP0049: `util.isFunction()`

Type: Documentation-only

The `util.isFunction()` API is deprecated.

DEP0050: `util.isNull()`

Type: Documentation-only

The `util.isNull()` API is deprecated.

DEP0051: `util.isNullOrUndefined()`

Type: Documentation-only

The `util.isNullOrUndefined()` API is deprecated.

DEP0052: `util.isNumber()`

Type: Documentation-only

The `util.isNumber()` API is deprecated.

DEP0053: `util.isObject()`

Type: Documentation-only

The `util.isObject()` API is deprecated.

DEP0054: `util.isPrimitive()`

Type: Documentation-only

The `util.isPrimitive()` API is deprecated.

DEP0055: `util.isRegExp()`

Type: Documentation-only

The `util.isRegExp()` API is deprecated.

DEP0056: `util.isString()`

Type: Documentation-only

The `util.isString()` API is deprecated.

DEP0057: `util.isSymbol()`

Type: Documentation-only

The `util.isSymbol()` API is deprecated.

DEP0058: `util.isUndefined()`

Type: Documentation-only

The `util.isUndefined()` API is deprecated.

DEP0059: `util.log()`

Type: Documentation-only

The `util.log()` API is deprecated.

DEP0060: `util._extend()`

Type: Documentation-only

The `util._extend()` API is deprecated.

DEP0061: `fs.SyncWriteStream`

Type: End-of-Life

The `fs.SyncWriteStream` class was never intended to be a publicly accessible API and has been removed. No alternative API is available. Please use a userland alternative.

DEP0062: `node --debug`

Type: End-of-Life

`--debug` activates the legacy V8 debugger interface, which was removed as of V8 5.8. It is replaced by Inspector which is activated with `--inspect` instead.

DEP0063: `ServerResponse.prototype.writeHead()`

Type: Documentation-only

The `http` module `ServerResponse.prototype.writeHead()` API is deprecated. Please use `ServerResponse.prototype.writeHead()` instead.

The `ServerResponse.prototype.writeHead()` method was never documented as an officially supported API.

DEP0064: `tls.createSecurePair()`

Type: Runtime

The `tls.createSecurePair()` API was deprecated in documentation in Node.js 0.11.3. Users should use `tls.Socket` instead.

DEP0065: `repl.REPL_MODE_MAGIC` and `NODE_REPL_MODE=magic`

Type: End-of-Life

The `repl` module's `REPL_MODE_MAGIC` constant, used for `replMode` option, has been removed. Its behavior has been functionally identical to that of `REPL_MODE_SLOPPY` since Node.js 6.0.0, when V8 5.0 was imported. Please use `REPL_MODE_SLOPPY` instead.

The `NODE_REPL_MODE` environment variable is used to set the underlying `replMode` of an interactive `node` session. Its value, `magic`, is also removed. Please use `sloppy` instead.

DEP0066: `OutgoingMessage.prototype.headers`, `OutgoingMessage.prototype.headerNames`

Type: Runtime

The `http` module `OutgoingMessage.prototype.headers` and `OutgoingMessage.prototype.headerNames` properties are deprecated. Use one of the public methods (e.g. `OutgoingMessage.prototype.getHeader()`, `OutgoingMessage.prototype.getHeaders()`, `OutgoingMessage.prototype.getHeaderNames()`, `OutgoingMessage.prototype.getRawHeaderNames()`, `OutgoingMessage.prototype.hasHeader()`, `OutgoingMessage.prototype.removeHeader()`, `OutgoingMessage.prototype.setHeader()`) for working with outgoing headers.

The `OutgoingMessage.prototype.headers` and `OutgoingMessage.prototype.headerNames` properties were never documented as officially supported properties.

DEP0067: `OutgoingMessage.prototype._renderHeaders`

Type: Documentation-only

The `http` module `OutgoingMessage.prototype._renderHeaders()` API is deprecated.

The `OutgoingMessage.prototype._renderHeaders` property was never documented as an officially supported API.

DEP0068: `node debug`

Type: End-of-Life

`node debug` corresponds to the legacy CLI debugger which has been replaced with a V8-inspector based CLI debugger available through `node inspect`.

DEP0069: `vm.runInDebugContext(string)`

Type: End-of-Life

`DebugContext` has been removed in V8 and is not available in Node.js 10+.

`DebugContext` was an experimental API.

DEP0070: `async_hooks.currentId()`

Type: End-of-Life

`async_hooks.currentId()` was renamed to `async_hooks.executionAsyncId()` for clarity.

This change was made while `async_hooks` was an experimental API.

DEP0071: `async_hooks.triggerId()`

Type: End-of-Life

`async_hooks.triggerId()` was renamed to `async_hooks.triggerAsyncId()` for clarity.

This change was made while `async_hooks` was an experimental API.

DEP0072: `async_hooks.AsyncResource.triggerId()`

Type: End-of-Life

`async_hooks.AsyncResource.triggerId()` was renamed to `async_hooks.AsyncResource.triggerAsyncId()` for clarity.

This change was made while `async_hooks` was an experimental API.

DEP0073: Several internal properties of `net.Server`

Type: End-of-Life

Accessing several internal, undocumented properties of `net.Server` instances with inappropriate names is deprecated.

As the original API was undocumented and not generally useful for non-internal code, no replacement API is provided.

DEP0074: `REPLServer.bufferedCommand`

Type: End-of-Life

The `REPLServer.bufferedCommand` property was deprecated in favor of `REPLServer.clearBufferedCommand()`.

DEP0075: `REPLServer.parseREPLKeyword()`

Type: End-of-Life

`REPLServer.parseREPLKeyword()` was removed from userland visibility.

DEP0076: `tls.parseCertString()`

Type: Runtime

`tls.parseCertString()` is a trivial parsing helper that was made public by mistake. This function can usually be replaced with:

```
const querystring = require('querystring');
querystring.parse(str, '\n', '=');
```

This function is not completely equivalent to `querystring.parse()`. One difference is that `querystring.parse()` does url decoding:

```
> querystring.parse('%E5%A5%BD=1', '\n', '=');
{ '好': '1' }
```

```
> tls.parseCertString('%E5%A5%BD=1');  
{ '%E5%A5%BD': '1' }
```

DEP0077: Module._debug()

Type: Runtime

`Module._debug()` is deprecated.

The `Module._debug()` function was never documented as an officially supported API.

DEP0078: REPLServer.turnOffEditorMode()

Type: End-of-Life

`REPLServer.turnOffEditorMode()` was removed from userland visibility.

DEP0079: Custom inspection function on objects via .inspect()

Type: End-of-Life

Using a property named `inspect` on an object to specify a custom inspection function for `util.inspect()` is deprecated. Use `util.inspect.custom` instead. For backward compatibility with Node.js prior to version 6.4.0, both can be specified.

DEP0080: path._makeLong()

Type: Documentation-only

The internal `path._makeLong()` was not intended for public use. However, userland modules have found it useful. The internal API is deprecated and replaced with an identical, public `path.toNamespacedPath()` method.

DEP0081: fs.truncate() using a file descriptor

Type: Runtime

`fs.truncate()` `fs.truncateSync()` usage with a file descriptor is deprecated. Please use `fs.ftruncate()` or `fs.ftruncateSync()` to work with file descriptors.

DEP0082: REPLServer.prototype.memory()

Type: End-of-Life

`REPLServer.prototype.memory()` is only necessary for the internal mechanics of the `REPLServer` itself. Do not use this function.

DEP0083: Disabling ECDH by setting ecdhCurve to false

Type: End-of-Life.

The `ecdhCurve` option to `tls.createSecureContext()` and `tls.TLSSocket` could be set to `false` to disable ECDH entirely on the server only. This mode was deprecated in preparation for migrating to OpenSSL 1.1.0 and consistency with the client and is now unsupported. Use the `ciphers` parameter instead.

DEP0084: requiring bundled internal dependencies

Type: End-of-Life

Since Node.js versions 4.4.0 and 5.2.0, several modules only intended for internal usage were mistakenly exposed to user code through `require()`. These modules were:

- `v8/tools/codemap`
- `v8/tools/consarray`
- `v8/tools/csvparser`
- `v8/tools/logreader`
- `v8/tools/profile_view`
- `v8/tools/profile`
- `v8/tools/SourceMap`
- `v8/tools/splaytree`
- `v8/tools/tickprocessor-driver`
- `v8/tools/tickprocessor`
- `node-inspect/lib/_inspect` (from 7.6.0)
- `node-inspect/lib/internal/inspect_client` (from 7.6.0)
- `node-inspect/lib/internal/inspect_repl` (from 7.6.0)

The `v8/*` modules do not have any exports, and if not imported in a specific order would in fact throw errors. As such there are virtually no legitimate use cases for importing them through `require()`.

On the other hand, `node-inspect` can be installed locally through a package manager, as it is published on the npm registry under the same name. No source code modification is necessary if that is done.

DEP0085: AsyncHooks sensitive API

Type: End-of-Life

The AsyncHooks sensitive API was never documented and had various minor issues. Use the `AsyncResource` API instead. See <https://github.com/nodejs/node/issues/15572>.

DEP0086: Remove `runInAsyncIdScope`

Type: End-of-Life

`runInAsyncIdScope` doesn't emit the `'before'` or `'after'` event and can thus cause a lot of issues. See <https://github.com/nodejs/node/issues/14328>.

DEP0089: `require('assert')`

Type: Deprecation revoked

Importing `assert` directly was not recommended as the exposed functions use loose equality checks. The deprecation was revoked because use of the `assert` module is not discouraged, and the deprecation caused developer confusion.

DEP0090: Invalid GCM authentication tag lengths

Type: End-of-Life

Node.js used to support all GCM authentication tag lengths which are accepted by OpenSSL when calling `decipher.setAuthTag()`. Beginning with Node.js v11.0.0, only authentication tag lengths of 128, 120, 112, 104, 96, 64, and 32 bits are allowed. Authentication tags of other lengths are invalid per [NIST SP 800-38D](#).

DEP0091: `crypto.DEFAULT_ENCODING`

Type: Runtime

The `crypto.DEFAULT_ENCODING` property is deprecated.

DEP0092: Top-level `this` bound to `module.exports`

Type: Documentation-only

Assigning properties to the top-level `this` as an alternative to `module.exports` is deprecated. Developers should use `exports` or `module.exports` instead.

DEP0093: `crypto.fips` is deprecated and replaced

Type: Documentation-only

The `crypto.fips` property is deprecated. Please use `crypto.setFips()` and `crypto.getFips()` instead.

DEP0094: Using `assert.fail()` with more than one argument

Type: Runtime

Using `assert.fail()` with more than one argument is deprecated. Use `assert.fail()` with only one argument or use a different `assert` module method.

DEP0095: `timers.enroll()`

Type: Runtime

`timers.enroll()` is deprecated. Please use the publicly documented `setTimeout()` or `setInterval()` instead.

DEP0096: `timers.unenroll()`

Type: Runtime

`timers.unenroll()` is deprecated. Please use the publicly documented `clearTimeout()` or `clearInterval()` instead.

DEP0097: `MakeCallback` with `domain` property

Type: Runtime

Users of `MakeCallback` that add the `domain` property to carry context, should start using the `async_context` variant of `MakeCallback` or `CallbackScope`, or the high-level `AsyncResource` class.

DEP0098: AsyncHooks embedder `AsyncResource.emitBefore` and `AsyncResource.emitAfter` APIs

Type: End-of-Life

The embedded API provided by AsyncHooks exposes `.emitBefore()` and `.emitAfter()` methods which are very easy to use incorrectly which can lead to unrecoverable errors.

Use `asyncResource.runInAsyncScope()` API instead which provides a much safer, and more convenient, alternative. See <https://github.com/nodejs/node/pull/18513>.

DEP0099: Async context-unaware `node::MakeCallback` C++ APIs

Type: Compile-time

Certain versions of `node:MakeCallback` APIs available to native modules are deprecated. Please use the versions of the API that accept an `async_context` parameter.

DEP0100: `process.assert()`

Type: Runtime

`process.assert()` is deprecated. Please use the `assert` module instead.

This was never a documented feature.

DEP0101: `--with-lttng`

Type: End-of-Life

The `--with-lttng` compile-time option has been removed.

DEP0102: Using `noAssert` in `Buffer#(read|write)` operations

Type: End-of-Life

Using the `noAssert` argument has no functionality anymore. All input is going to be verified, no matter if it is set to true or not. Skipping the verification could lead to hard to find errors and crashes.

DEP0103: `process.binding('util').is[...]` typechecks

Type: Documentation-only (supports `--pending-deprecation`)

Using `process.binding()` in general should be avoided. The type checking methods in particular can be replaced by using `util.types`.

This deprecation has been superseded by the deprecation of the `process.binding()` API (DEP0111).

DEP0104: `process.env` string coercion

Type: Documentation-only (supports `--pending-deprecation`)

When assigning a non-string property to `process.env`, the assigned value is implicitly converted to a string. This behavior is deprecated if the assigned value is not a string, boolean, or number. In the future, such assignment might result in a thrown error. Please convert the property to a string before assigning it to `process.env`.

DEP0105: `decipher.finalto1`

Type: End-of-Life

`decipher.finalto1()` has never been documented and was an alias for `decipher.final()`. This API has been removed, and it is recommended to use `decipher.final()` instead.

DEP0106: `crypto.createCipher` and `crypto.createDecipher`

Type: Runtime

Using `crypto.createCipher()` and `crypto.createDecipher()` should be avoided as they use a weak key derivation function (MD5 with no salt) and static initialization vectors. It is recommended to derive a key using `crypto.pbkdf2()` or `crypto.scrypt()` and to use `crypto.createCipheriv()` and `crypto.createDecipheriv()` to obtain the `Cipher` and `Decipher` objects respectively.

DEP0107: `tls.convertNPNProtocols()`

Type: End-of-Life

This was an undocumented helper function not intended for use outside Node.js core and obsoleted by the removal of NPN (Next Protocol Negotiation) support.

DEP0108: `zlib.bytesRead`

Type: Runtime

Deprecated alias for `zlib.bytesWritten`. This original name was chosen because it also made sense to interpret the value as the number of bytes read by the engine, but is inconsistent with other streams in Node.js that expose values under these names.

DEP0109: `http`, `https`, and `tls` support for invalid URLs

Type: End-of-Life

Some previously supported (but strictly invalid) URLs were accepted through the `http.request()`, `http.get()`, `https.request()`, `https.get()`, and `tls.checkServerIdentity()` APIs because those were accepted by the legacy `url.parse()` API. The mentioned APIs now use the WHATWG URL parser that requires strictly valid URLs. Passing an invalid URL is deprecated and support will be removed in the future.

DEP0110: `vm.Script` cached data

Type: Documentation-only

The `produceCachedData` option is deprecated. Use `script.createCachedData()` instead.

DEP0111: `process.binding()`

Type: Documentation-only (supports `--pending-deprecation`)

`process.binding()` is for use by Node.js internal code only.

DEP0112: `dgram` private APIs

Type: Runtime

The `dgram` module previously contained several APIs that were never meant to be accessed outside of Node.js core:

`Socket.prototype._handle`, `Socket.prototype._receiving`, `Socket.prototype._bindState`, `Socket.prototype._queue`, `Socket.prototype._reuseAddr`, `Socket.prototype._healthCheck()`, `Socket.prototype._stopReceiving()`, and `dgram._createSocketHandle()`.

DEP0113: `Cipher.setAuthTag()`, `Decipher.getAuthTag()`

Type: End-of-Life

`Cipher.setAuthTag()` and `Decipher.getAuthTag()` are no longer available. They were never documented and would throw when called.

DEP0114: `crypto._toBuf()`

Type: End-of-Life

The `crypto._toBuf()` function was not designed to be used by modules outside of Node.js core and was removed.

DEP0115: `crypto.prng()`, `crypto.pseudoRandomBytes()`, `crypto.rng()`

Type: Documentation-only (supports `--pending-deprecation`)

In recent versions of Node.js, there is no difference between `crypto.randomBytes()` and `crypto.pseudoRandomBytes()`. The latter is deprecated along with the undocumented aliases `crypto.prng()` and `crypto.rng()` in favor of `crypto.randomBytes()` and might be removed in a future release.

DEP0116: Legacy URL API

Type: Deprecation revoked

The **Legacy URL API** is deprecated. This includes `url.format()`, `url.parse()`, `url.resolve()`, and the `legacyUrlObject`. Please use the **WHATWG URL API** instead.

DEP0117: Native crypto handles

Type: End-of-Life

Previous versions of Node.js exposed handles to internal native objects through the `_handle` property of the `Cipher`, `Decipher`, `DiffieHellman`, `DiffieHellmanGroup`, `ECDH`, `Hash`, `Hmac`, `Sign`, and `Verify` classes. The `_handle` property has been removed because improper use of the native object can lead to crashing the application.

DEP0118: `dns.lookup()` support for a falsy host name

Type: Runtime

Previous versions of Node.js supported `dns.lookup()` with a falsy host name like `dns.lookup(false)` due to backward compatibility. This behavior is undocumented and is thought to be unused in real world apps. It will become an error in future versions of Node.js.

DEP0119: `process.binding('uv').errname()` private API

Type: Documentation-only (supports `--pending-deprecation`)

`process.binding('uv').errname()` is deprecated. Please use `util.getSystemErrorMessage()` instead.

DEP0120: Windows Performance Counter support

Type: End-of-Life

Windows Performance Counter support has been removed from Node.js. The undocumented `COUNTER_NET_SERVER_CONNECTION()`, `COUNTER_NET_SERVER_CONNECTION_CLOSE()`, `COUNTER_HTTP_SERVER_REQUEST()`, `COUNTER_HTTP_SERVER_RESPONSE()`, `COUNTER_HTTP_CLIENT_REQUEST()`, and `COUNTER_HTTP_CLIENT_RESPONSE()` functions have been deprecated.

DEP0121: `net._setSimultaneousAccepts()`

Type: Runtime

The undocumented `net._setSimultaneousAccepts()` function was originally intended for debugging and performance tuning when using the `child_process` and `cluster` modules on Windows. The function is not generally useful and is being removed. See discussion here: <https://github.com/nodejs/node/issues/18391>

DEP0122: `tls.Server.prototype.setOptions()`

Type: Runtime

Please use `Server.prototype.setSecureContext()` instead.

DEP0123: setting the TLS ServerName to an IP address

Type: Runtime

Setting the TLS ServerName to an IP address is not permitted by [RFC 6066](#). This will be ignored in a future version.

DEP0124: using `REPLServer.rli`

Type: End-of-Life

This property is a reference to the instance itself.

DEP0125: `require('_stream_wrap')`

Type: Runtime

The `_stream_wrap` module is deprecated.

DEP0126: `timers.active()`

Type: Runtime

The previously undocumented `timers.active()` is deprecated. Please use the publicly documented `timeout.refresh()` instead. If re-referencing the timeout is necessary, `timeout.ref()` can be used with no performance impact since Node.js 10.

DEP0127: `timers._unrefActive()`

Type: Runtime

The previously undocumented and "private" `timers._unrefActive()` is deprecated. Please use the publicly documented `timeout.refresh()` instead. If unreferencing the timeout is necessary, `timeout.unref()` can be used with no performance impact since Node.js 10.

DEP0128: modules with an invalid `main` entry and an `index.js` file

Type: Runtime

Modules that have an invalid `main` entry (e.g., `./does-not-exist.js`) and also have an `index.js` file in the top level directory will resolve the `index.js` file. That is deprecated and is going to throw an error in future Node.js versions.

DEP0129: `ChildProcess._channel`

Type: Runtime

The `_channel` property of child process objects returned by `spawn()` and similar functions is not intended for public use. Use `ChildProcess.channel` instead.

DEP0130: `Module.createRequireFromPath()`

Type: End-of-Life

Use `module.createRequire()` instead.

DEP0131: Legacy HTTP parser

Type: End-of-Life

The legacy HTTP parser, used by default in versions of Node.js prior to 12.0.0, is deprecated and has been removed in v13.0.0. Prior to v13.0.0, the `--http-parser=legacy` command-line flag could be used to revert to using the legacy parser.

DEP0132: `worker.terminate()` with callback

Type: Runtime

Passing a callback to `worker.terminate()` is deprecated. Use the returned `Promise` instead, or a listener to the worker's `'exit'` event.

DEP0133: http connection

Type: Documentation-only

Prefer `response.socket` over `response.connection` and `request.socket` over `request.connection`.

DEP0134: process._tickCallback

Type: Documentation-only (supports `--pending-deprecation`)

The `process._tickCallback` property was never documented as an officially supported API.

DEP0135: WriteStream.open() and ReadStream.open() are internal

Type: Runtime

`WriteStream.open()` and `ReadStream.open()` are undocumented internal APIs that do not make sense to use in userland. File streams should always be opened through their corresponding factory methods `fs.createWriteStream()` and `fs.createReadStream()` or by passing a file descriptor in options.

DEP0136: http finished

Type: Documentation-only

`response.finished` indicates whether `response.end()` has been called, not whether `'finish'` has been emitted and the underlying data is flushed.

Use `response.writableFinished` or `response.writableEnded` accordingly instead to avoid the ambiguity.

To maintain existing behavior `response.finished` should be replaced with `response.writableEnded`.

DEP0137: Closing fs.FileHandle on garbage collection

Type: Runtime

Allowing a `fs.FileHandle` object to be closed on garbage collection is deprecated. In the future, doing so might result in a thrown error that will terminate the process.

Please ensure that all `fs.FileHandle` objects are explicitly closed using `FileHandle.prototype.close()` when the `fs.FileHandle` is no longer needed:

```
const fsPromises = require('fs').promises;
async function openAndClose() {
  let filehandle;
  try {
    filehandle = await fsPromises.open('thefile.txt', 'r');
  } finally {
    if (filehandle !== undefined)
      await filehandle.close();
  }
}
```

DEP0138: `process.mainModule`

Type: Documentation-only

`process.mainModule` is a CommonJS-only feature while `process` global object is shared with non-CommonJS environment. Its use within ECMAScript modules is unsupported.

It is deprecated in favor of `require.main`, because it serves the same purpose and is only available on CommonJS environment.

DEP0139: `process.umask()` with no arguments

Type: Documentation-only

Calling `process.umask()` with no argument causes the process-wide umask to be written twice. This introduces a race condition between threads, and is a potential security vulnerability. There is no safe, cross-platform alternative API.

DEP0140: Use `request.destroy()` instead of `request.abort()`

Type: Documentation-only

Use `request.destroy()` instead of `request.abort()`.

DEP0141: `repl.inputStream` and `repl.outputStream`

Type: Documentation-only (supports `--pending-deprecation`)

The `repl` module exported the input and output stream twice. Use `.input` instead of `.inputStream` and `.output` instead of `.outputStream`.

DEP0142: `repl._builtinLibs`

Type: Documentation-only

The `repl` module exports a `_builtinLibs` property that contains an array with native modules. It was incomplete so far and instead it's better to rely upon `require('module').builtinModules`.

DEP0143: `Transform._transformState`

Type: Runtime `Transform._transformState` will be removed in future versions where it is no longer required due to simplification of the implementation.

DEP0144: `module.parent`

Type: Documentation-only (supports `--pending-deprecation`)

A CommonJS module can access the first module that required it using `module.parent`. This feature is deprecated because it does not work consistently in the presence of ECMAScript modules and because it gives an inaccurate representation of the CommonJS module graph.

Some modules use it to check if they are the entry point of the current process. Instead, it is recommended to compare `require.main` and `module`:

```
if (require.main === module) {  
  // Code section that will run only if current file is the entry point.  
}
```

When looking for the CommonJS modules that have required the current one, `require.cache` and `module.children` can be used:

```
const moduleParents = Object.values(require.cache)
  .filter((m) => m.children.includes(module));
```

DEP0145: `socket.bufferSize`

Type: Documentation-only

`socket.bufferSize` is just an alias for `writable.writableLength`.

DEP0146: `new crypto.Certificate()`

Type: Documentation-only

The `crypto.Certificate()` constructor is deprecated. Use `static methods of crypto.Certificate()` instead.

DEP0147: `fs.rmdir(path, { recursive: true })`

Type: Runtime

In future versions of Node.js, `recursive` option will be ignored for `fs.rmdir`, `fs.rmdirSync`, and `fs.promises.rmdir`.

Use `fs.rm(path, { recursive: true, force: true })`, `fs.rmSync(path, { recursive: true, force: true })` or `fs.promises.rm(path, { recursive: true, force: true })` instead.

DEP0148: Folder mappings in "exports" (trailing "/")

Type: Runtime

Using a trailing "/" to define `subpath folder mappings` in the `subpath exports` or `subpath imports` fields is deprecated. Use `subpath patterns` instead.

DEP0149: `http.IncomingMessage#connection`

Type: Documentation-only.

Prefer `message.socket` over `message.connection`.

DEP0150: Changing the value of `process.config`

Type: Runtime

The `process.config` property is intended to provide access to configuration settings set when the Node.js binary was compiled. However, the property has been mutable by user code making it impossible to rely on. The ability to change the value has been deprecated and will be disabled in the future.

DEP0151: Main index lookup and extension searching

Type: Runtime

Previously, `index.js` and extension searching lookups would apply to `import 'pkg'` main entry point resolution, even when resolving ES modules.

With this deprecation, all ES module main entry point resolutions require an explicit `"exports" or "main" entry` with the exact file extension.

DEP0152: Extension PerformanceEntry properties

Type: Runtime

The 'gc', 'http2', and 'http' `<PerformanceEntry>` object types have additional properties assigned to them that provide additional information. These properties are now available within the standard `detail` property of the `PerformanceEntry` object. The existing accessors have been deprecated and should no longer be used.

Diagnostics Channel

Stability: 1 - Experimental

Source Code: [lib/diagnostics_channel.js](#)

The `diagnostics_channel` module provides an API to create named channels to report arbitrary message data for diagnostics purposes.

It can be accessed using:

```
import diagnostics_channel from 'diagnostics_channel';const diagnostics_channel = require('diagnostics_channel');
```

It is intended that a module writer wanting to report diagnostics messages will create one or many top-level channels to report messages through. Channels may also be acquired at runtime but it is not encouraged due to the additional overhead of doing so. Channels may be exported for convenience, but as long as the name is known it can be acquired anywhere.

If you intend for your module to produce diagnostics data for others to consume it is recommended that you include documentation of what named channels are used along with the shape of the message data. Channel names should generally include the module name to avoid collisions with data from other modules.

Public API

Overview

Following is a simple overview of the public API.

```
import diagnostics_channel from 'diagnostics_channel';

// Get a reusable channel object
const channel = diagnostics_channel.channel('my-channel');

// Subscribe to the channel
channel.subscribe((message, name) => {
  // Received data
});

// Check if the channel has an active subscriber
if (channel.hasSubscribers) {
  // Publish data to the channel
  channel.publish({
    some: 'data'
  });
}const diagnostics_channel = require('diagnostics_channel');

// Get a reusable channel object
```

```
const channel = diagnostics_channel.channel('my-channel');

// Subscribe to the channel
channel.subscribe((message, name) => {
  // Received data
});

// Check if the channel has an active subscriber
if (channel.hasSubscribers) {
  // Publish data to the channel
  channel.publish({
    some: 'data'
  });
}
```

diagnostics_channel.hasSubscribers(name)

- `name` `<string>` | `<symbol>` The channel name
- Returns: `<boolean>` If there are active subscribers

Check if there are active subscribers to the named channel. This is helpful if the message you want to send might be expensive to prepare.

This API is optional but helpful when trying to publish messages from very performance-sensitive code.

```
import diagnostics_channel from 'diagnostics_channel';

if (diagnostics_channel.hasSubscribers('my-channel')) {
  // There are subscribers, prepare and publish message
}const diagnostics_channel = require('diagnostics_channel');

if (diagnostics_channel.hasSubscribers('my-channel')) {
  // There are subscribers, prepare and publish message
}
```

diagnostics_channel.channel(name)

- `name` `<string>` | `<symbol>` The channel name
- Returns: `<Channel>` The named channel object

This is the primary entry-point for anyone wanting to interact with a named channel. It produces a channel object which is optimized to reduce overhead at publish time as much as possible.

```
import diagnostics_channel from 'diagnostics_channel';

const channel = diagnostics_channel.channel('my-channel');const diagnostics_channel = require('diagnostics_channel');

const channel = diagnostics_channel.channel('my-channel');
```

Class: Channel

The class `Channel` represents an individual named channel within the data pipeline. It is use to track subscribers and to publish messages when there are subscribers present. It exists as a separate object to avoid channel lookups at publish time, enabling very fast publish speeds and allowing for heavy use while incurring very minimal cost. Channels are created with `diagnostics_channel.channel(name)`, constructing a channel directly with `new Channel(name)` is not supported.

`channel.hasSubscribers`

- Returns: `<boolean>` If there are active subscribers

Check if there are active subscribers to this channel. This is helpful if the message you want to send might be expensive to prepare.

This API is optional but helpful when trying to publish messages from very performance-sensitive code.

```
import diagnostics_channel from 'diagnostics_channel';

const channel = diagnostics_channel.channel('my-channel');

if (channel.hasSubscribers) {
  // There are subscribers, prepare and publish message
}const diagnostics_channel = require('diagnostics_channel');

const channel = diagnostics_channel.channel('my-channel');

if (channel.hasSubscribers) {
  // There are subscribers, prepare and publish message
}
```

`channel.publish(message)`

- `message` `<any>` The message to send to the channel subscribers

Publish a message to any subscribers to the channel. This will trigger message handlers synchronously so they will execute within the same context.

```
import diagnostics_channel from 'diagnostics_channel';

const channel = diagnostics_channel.channel('my-channel');

channel.publish({
  some: 'message'
});const diagnostics_channel = require('diagnostics_channel');

const channel = diagnostics_channel.channel('my-channel');

channel.publish({
  some: 'message'
});
```

`channel.subscribe(onMessage)`

- `onMessage` `<Function>` The handler to receive channel messages
 - `message` `<any>` The message data

- `name` `<string>` | `<symbol>` The name of the channel

Register a message handler to subscribe to this channel. This message handler will be run synchronously whenever a message is published to the channel. Any errors thrown in the message handler will trigger an `'uncaughtException'` .

```
import diagnostics_channel from 'diagnostics_channel';

const channel = diagnostics_channel.channel('my-channel');

channel.subscribe((message, name) => {
  // Received data
});const diagnostics_channel = require('diagnostics_channel');

const channel = diagnostics_channel.channel('my-channel');

channel.subscribe((message, name) => {
  // Received data
});
```

`channel.unsubscribe(onMessage)`

- `onMessage` `<Function>` The previous subscribed handler to remove

Remove a message handler previously registered to this channel with `channel.subscribe(onMessage)` .

```
import diagnostics_channel from 'diagnostics_channel';

const channel = diagnostics_channel.channel('my-channel');

function onMessage(message, name) {
  // Received data
}

channel.subscribe(onMessage);

channel.unsubscribe(onMessage);const diagnostics_channel = require('diagnostics_channel');

const channel = diagnostics_channel.channel('my-channel');

function onMessage(message, name) {
  // Received data
}

channel.subscribe(onMessage);

channel.unsubscribe(onMessage);
```

DNS

Source Code: [lib/dns.js](#)

The `dns` module enables name resolution. For example, use it to look up IP addresses of host names.

Although named for the [Domain Name System \(DNS\)](#), it does not always use the DNS protocol for lookups. `dns.lookup()` uses the operating system facilities to perform name resolution. It may not need to perform any network communication. To perform name resolution the way other applications on the same system do, use `dns.lookup()`.

```
const dns = require('dns');

dns.lookup('example.org', (err, address, family) => {
  console.log('address: %j family: IPv%s', address, family);
});
// address: "93.184.216.34" family: IPv4
```

All other functions in the `dns` module connect to an actual DNS server to perform name resolution. They will always use the network to perform DNS queries. These functions do not use the same set of configuration files used by `dns.lookup()` (e.g. `/etc/hosts`). Use these functions to always perform DNS queries, bypassing other name-resolution facilities.

```
const dns = require('dns');

dns.resolve4('archive.org', (err, addresses) => {
  if (err) throw err;

  console.log(`addresses: ${JSON.stringify(addresses)}`);

  addresses.forEach((a) => {
    dns.reverse(a, (err, hostnames) => {
      if (err) {
        throw err;
      }
      console.log(`reverse for ${a}: ${JSON.stringify(hostnames)}`);
    });
  });
});
```

See the [Implementation considerations](#) section for more information.

Class: `dns.Resolver`

An independent resolver for DNS requests.

Creating a new resolver uses the default server settings. Setting the servers used for a resolver using `resolver.setServers()` does not affect other resolvers:

```
const { Resolver } = require('dns');
const resolver = new Resolver();
```

```

resolver.setServers(['4.4.4.4']);

// This request will use the server at 4.4.4.4, independent of global settings.
resolver.resolve4('example.org', (err, addresses) => {
  // ...
});

```

The following methods from the `dns` module are available:

- `resolver.getServers()`
- `resolver.resolve()`
- `resolver.resolve4()`
- `resolver.resolve6()`
- `resolver.resolveAny()`
- `resolver.resolveCaa()`
- `resolver.resolveCname()`
- `resolver.resolveMx()`
- `resolver.resolveNaptr()`
- `resolver.resolveNs()`
- `resolver.resolvePtr()`
- `resolver.resolveSoa()`
- `resolver.resolveSrv()`
- `resolver.resolveTxt()`
- `resolver.reverse()`
- `resolver.setServers()`

Resolver([options])

Create a new resolver.

- `options` `<Object>`
 - `timeout` `<integer>` Query timeout in milliseconds, or `-1` to use the default timeout.
 - `tries` `<integer>` The number of tries the resolver will try contacting each name server before giving up. **Default:** `4`

resolver.cancel()

Cancel all outstanding DNS queries made by this resolver. The corresponding callbacks will be called with an error with code `ECANCELLED`.

resolver.setLocalAddress([ipv4][, ipv6])

- `ipv4` `<string>` A string representation of an IPv4 address. **Default:** `'0.0.0.0'`
- `ipv6` `<string>` A string representation of an IPv6 address. **Default:** `:::0'`

The resolver instance will send its requests from the specified IP address. This allows programs to specify outbound interfaces when used on multi-homed systems.

If a v4 or v6 address is not specified, it is set to the default, and the operating system will choose a local address automatically.

The resolver will use the v4 local address when making requests to IPv4 DNS servers, and the v6 local address when making requests to IPv6 DNS servers. The `rrtype` of resolution requests has no impact on the local address used.

dns.getServers()

- Returns: `<string[]>`

Returns an array of IP address strings, formatted according to [RFC 5952](#), that are currently configured for DNS resolution. A string will include a port section if a custom port is used.

```
[
  '4.4.4.4',
  '2001:4860:4860::8888',
  '4.4.4.4:1053',
  '[2001:4860:4860::8888]:1053',
]
```

dns.lookup(hostname[, options], callback)

- `hostname` `<string>`
- `options` `<integer> | <Object>`
 - `family` `<integer>` The record family. Must be `4`, `6`, or `0`. The value `0` indicates that IPv4 and IPv6 addresses are both returned. **Default:** `0`.
 - `hints` `<number>` One or more [supported getaddrinfo flags](#). Multiple flags may be passed by bitwise ORing their values.
 - `all` `<boolean>` When `true`, the callback returns all resolved addresses in an array. Otherwise, returns a single address. **Default:** `false`.
 - `verbatim` `<boolean>` When `true`, the callback receives IPv4 and IPv6 addresses in the order the DNS resolver returned them. When `false`, IPv4 addresses are placed before IPv6 addresses. **Default:** currently `false` (addresses are reordered) but this is expected to change in the not too distant future. Default value is configurable using `dns.setDefaultResultOrder()` or `--dns-result-order`. New code should use `{ verbatim: true }`.
- `callback` `<Function>`
 - `err` `<Error>`
 - `address` `<string>` A string representation of an IPv4 or IPv6 address.
 - `family` `<integer>` `4` or `6`, denoting the family of `address`, or `0` if the address is not an IPv4 or IPv6 address. `0` is a likely indicator of a bug in the name resolution service used by the operating system.

Resolves a host name (e.g. `'nodejs.org'`) into the first found A (IPv4) or AAAA (IPv6) record. All `option` properties are optional. If `options` is an integer, then it must be `4` or `6` – if `options` is not provided, then IPv4 and IPv6 addresses are both returned if found.

With the `all` option set to `true`, the arguments for `callback` change to `(err, addresses)`, with `addresses` being an array of objects with the properties `address` and `family`.

On error, `err` is an `Error` object, where `err.code` is the error code. Keep in mind that `err.code` will be set to `'ENOTFOUND'` not only when the host name does not exist but also when the lookup fails in other ways such as no available file descriptors.

`dns.lookup()` does not necessarily have anything to do with the DNS protocol. The implementation uses an operating system facility that can associate names with addresses, and vice versa. This implementation can have subtle but important consequences on the behavior of any Node.js program. Please take some time to consult the [Implementation considerations section](#) before using `dns.lookup()`.

Example usage:

```
const dns = require('dns');
const options = {
```

```

family: 6,
hints: dns.ADDRCONFIG | dns.V4MAPPED,
};
dns.lookup('example.com', options, (err, address, family) =>
  console.log('address: %j family: IPv%s', address, family));
// address: "2606:2800:220:1:248:1893:25c8:1946" family: IPv6

// When options.all is true, the result will be an Array.
options.all = true;
dns.lookup('example.com', options, (err, addresses) =>
  console.log('addresses: %j', addresses));
// addresses: [{"address":"2606:2800:220:1:248:1893:25c8:1946","family":6}]

```

If this method is invoked as its `util.promisify()` ed version, and `all` is not set to `true`, it returns a `Promise` for an `Object` with `address` and `family` properties.

Supported getaddrinfo flags

The following flags can be passed as hints to `dns.lookup()`.

- `dns.ADDRCONFIG` : Limits returned address types to the types of non-loopback addresses configured on the system. For example, IPv4 addresses are only returned if the current system has at least one IPv4 address configured.
- `dns.V4MAPPED` : If the IPv6 family was specified, but no IPv6 addresses were found, then return IPv4 mapped IPv6 addresses. It is not supported on some operating systems (e.g FreeBSD 10.1).
- `dns.ALL` : If `dns.V4MAPPED` is specified, return resolved IPv6 addresses as well as IPv4 mapped IPv6 addresses.

dns.lookupService(address, port, callback)

- `address` `<string>`
- `port` `<number>`
- `callback` `<Function>`
 - `err` `<Error>`
 - `hostname` `<string>` e.g. `example.com`
 - `service` `<string>` e.g. `http`

Resolves the given `address` and `port` into a host name and service using the operating system's underlying `getnameinfo` implementation.

If `address` is not a valid IP address, a `TypeError` will be thrown. The `port` will be coerced to a number. If it is not a legal port, a `TypeError` will be thrown.

On an error, `err` is an `Error` object, where `err.code` is the error code.

```

const dns = require('dns');
dns.lookupService('127.0.0.1', 22, (err, hostname, service) => {
  console.log(hostname, service);
  // Prints: localhost ssh
});

```

If this method is invoked as its `util.promisify()` ed version, it returns a `Promise` for an `Object` with `hostname` and `service` properties.

dns.resolve(hostname[, rrtype], callback)

- `hostname` `<string>` Host name to resolve.
- `rrtype` `<string>` Resource record type. **Default:** `'A'`.
- `callback` `<Function>`
 - `err` `<Error>`
 - `records` `<string[]> | <Object[]> | <Object>`

Uses the DNS protocol to resolve a host name (e.g. `'nodejs.org'`) into an array of the resource records. The `callback` function has arguments (`err`, `records`). When successful, `records` will be an array of resource records. The type and structure of individual results varies based on `rrtype`:

rrtype	records contains	Result type	Shorthand method
<code>'A'</code>	IPv4 addresses (default)	<code><string></code>	<code>dns.resolve4()</code>
<code>'AAAA'</code>	IPv6 addresses	<code><string></code>	<code>dns.resolve6()</code>
<code>'ANY'</code>	any records	<code><Object></code>	<code>dns.resolveAny()</code>
<code>'CAA'</code>	CA authorization records	<code><Object></code>	<code>dns.resolveCaa()</code>
<code>'CNAME'</code>	canonical name records	<code><string></code>	<code>dns.resolveCname()</code>
<code>'MX'</code>	mail exchange records	<code><Object></code>	<code>dns.resolveMx()</code>
<code>'NAPTR'</code>	name authority pointer records	<code><Object></code>	<code>dns.resolveNaptr()</code>
<code>'NS'</code>	name server records	<code><string></code>	<code>dns.resolveNs()</code>
<code>'PTR'</code>	pointer records	<code><string></code>	<code>dns.resolvePtr()</code>
<code>'SOA'</code>	start of authority records	<code><Object></code>	<code>dns.resolveSoa()</code>
<code>'SRV'</code>	service records	<code><Object></code>	<code>dns.resolveSrv()</code>
<code>'TXT'</code>	text records	<code><string[]></code>	<code>dns.resolveTxt()</code>

On error, `err` is an `Error` object, where `err.code` is one of the `DNS error codes`.

dns.resolve4(hostname[, options], callback)

- `hostname` `<string>` Host name to resolve.
- `options` `<Object>`
 - `ttl` `<boolean>` Retrieve the Time-To-Live value (TTL) of each record. When `true`, the callback receives an array of `{ address: '1.2.3.4', ttl: 60 }` objects rather than an array of strings, with the TTL expressed in seconds.
- `callback` `<Function>`
 - `err` `<Error>`
 - `addresses` `<string[]> | <Object[]>`

Uses the DNS protocol to resolve a IPv4 addresses (A records) for the `hostname`. The `addresses` argument passed to the `callback` function will contain an array of IPv4 addresses (e.g. `['74.125.79.104', '74.125.79.105', '74.125.79.106']`).

dns.resolve6(hostname[, options], callback)

- `hostname` `<string>` Host name to resolve.
- `options` `<Object>`
 - `ttl` `<boolean>` Retrieve the Time-To-Live value (TTL) of each record. When `true`, the callback receives an array of `{ address: '0:1:2:3:4:5:6:7', ttl: 60 }` objects rather than an array of strings, with the TTL expressed in seconds.
- `callback` `<Function>`
 - `err` `<Error>`
 - `addresses` `<string[]> | <Object[]>`

Uses the DNS protocol to resolve a IPv6 addresses (AAAA records) for the `hostname`. The `addresses` argument passed to the `callback` function will contain an array of IPv6 addresses.

dns.resolveAny(hostname, callback)

- `hostname` `<string>`
- `callback` `<Function>`
 - `err` `<Error>`
 - `ret` `<Object[]>`

Uses the DNS protocol to resolve all records (also known as ANY or * query). The `ret` argument passed to the `callback` function will be an array containing various types of records. Each object has a property `type` that indicates the type of the current record. And depending on the `type`, additional properties will be present on the object:

Type	Properties
'A'	<code>address / ttl</code>
'AAAA'	<code>address / ttl</code>
'CNAME'	<code>value</code>
'MX'	Refer to <code>dns.resolveMx()</code>
'NAPTR'	Refer to <code>dns.resolveNaptr()</code>
'NS'	<code>value</code>
'PTR'	<code>value</code>
'SOA'	Refer to <code>dns.resolveSoa()</code>
'SRV'	Refer to <code>dns.resolveSrv()</code>
'TXT'	This type of record contains an array property called <code>entries</code> which refers to <code>dns.resolveTxt()</code> , e.g. <code>{ entries: ['...'], type: 'TXT' }</code>

Here is an example of the `ret` object passed to the callback:

```
[ { type: 'A', address: '127.0.0.1', ttl: 299 },
  { type: 'CNAME', value: 'example.com' },
  { type: 'MX', exchange: 'alt4.aspmx.l.example.com', priority: 50 },
  { type: 'NS', value: 'ns1.example.com' },
```

```
{ type: 'TXT', entries: [ 'v=spf1 include:_spf.example.com ~all' ] },
{ type: 'SOA',
  nsname: 'ns1.example.com',
  hostmaster: 'admin.example.com',
  serial: 156696742,
  refresh: 900,
  retry: 900,
  expire: 1800,
  minttl: 60 } ]
```

DNS server operators may choose not to respond to `ANY` queries. It may be better to call individual methods like `dns.resolve4()`, `dns.resolveMx()`, and so on. For more details, see [RFC 8482](#).

dns.resolveCname(hostname, callback)

- `hostname` `<string>`
- `callback` `<Function>`
 - `err` `<Error>`
 - `addresses` `<string[]>`

Uses the DNS protocol to resolve `CNAME` records for the `hostname`. The `addresses` argument passed to the `callback` function will contain an array of canonical name records available for the `hostname` (e.g. `['bar.example.com']`).

dns.resolveCaa(hostname, callback)

- `hostname` `<string>`
- `callback` `<Function>`
 - `err` `<Error>`
 - `records` `<Object[]>`

Uses the DNS protocol to resolve `CAA` records for the `hostname`. The `addresses` argument passed to the `callback` function will contain an array of certification authority authorization records available for the `hostname` (e.g. `[{critical: 0, iodef: 'mailto:pkg@example.com'}, {critical: 128, issue: 'pkg.example.com'}]`).

dns.resolveMx(hostname, callback)

- `hostname` `<string>`
- `callback` `<Function>`
 - `err` `<Error>`
 - `addresses` `<Object[]>`

Uses the DNS protocol to resolve mail exchange records (`MX` records) for the `hostname`. The `addresses` argument passed to the `callback` function will contain an array of objects containing both a `priority` and `exchange` property (e.g. `[{priority: 10, exchange: 'mx.example.com'}, ...]`).

dns.resolveNaptr(hostname, callback)

- `hostname` `<string>`
- `callback` `<Function>`

- `err` `<Error>`
- `addresses` `<Object[]>`

Uses the DNS protocol to resolve regular expression based records (`NAPTR` records) for the `hostname`. The `addresses` argument passed to the `callback` function will contain an array of objects with the following properties:

- `flags`
- `service`
- `regexp`
- `replacement`
- `order`
- `preference`

```
{
  flags: 's',
  service: 'SIP+D2U',
  regexp: '',
  replacement: '_sip._udp.example.com',
  order: 30,
  preference: 100
}
```

dns.resolveNs(hostname, callback)

- `hostname` `<string>`
- `callback` `<Function>`
 - `err` `<Error>`
 - `addresses` `<string[]>`

Uses the DNS protocol to resolve name server records (`NS` records) for the `hostname`. The `addresses` argument passed to the `callback` function will contain an array of name server records available for `hostname` (e.g. `['ns1.example.com', 'ns2.example.com']`).

dns.resolvePtr(hostname, callback)

- `hostname` `<string>`
- `callback` `<Function>`
 - `err` `<Error>`
 - `addresses` `<string[]>`

Uses the DNS protocol to resolve pointer records (`PTR` records) for the `hostname`. The `addresses` argument passed to the `callback` function will be an array of strings containing the reply records.

dns.resolveSoa(hostname, callback)

- `hostname` `<string>`
- `callback` `<Function>`
 - `err` `<Error>`
 - `address` `<Object>`

Uses the DNS protocol to resolve a start of authority record (SOA record) for the `hostname` . The `address` argument passed to the `callback` function will be an object with the following properties:

- `nsname`
- `hostmaster`
- `serial`
- `refresh`
- `retry`
- `expire`
- `minttl`

```
{
  nsname: 'ns.example.com',
  hostmaster: 'root.example.com',
  serial: 2013101809,
  refresh: 10000,
  retry: 2400,
  expire: 604800,
  minttl: 3600
}
```

`dns.resolveSrv(hostname, callback)`

- `hostname` `<string>`
- `callback` `<Function>`
 - `err` `<Error>`
 - `addresses` `<Object[]>`

Uses the DNS protocol to resolve service records (SRV records) for the `hostname` . The `addresses` argument passed to the `callback` function will be an array of objects with the following properties:

- `priority`
- `weight`
- `port`
- `name`

```
{
  priority: 10,
  weight: 5,
  port: 21223,
  name: 'service.example.com'
}
```

`dns.resolveTxt(hostname, callback)`

- `hostname` `<string>`
- `callback` `<Function>`

- `err` `<Error>`
- `records` `<string[][]>`

Uses the DNS protocol to resolve text queries (TXT records) for the `hostname`. The `records` argument passed to the `callback` function is a two-dimensional array of the text records available for `hostname` (e.g. [['v=spf1 ip4:0.0.0.0 ', '~all']]). Each sub-array contains TXT chunks of one record. Depending on the use case, these could be either joined together or treated separately.

dns.reverse(ip, callback)

- `ip` `<string>`
- `callback` `<Function>`
 - `err` `<Error>`
 - `hostnames` `<string[]>`

Performs a reverse DNS query that resolves an IPv4 or IPv6 address to an array of host names.

On error, `err` is an `Error` object, where `err.code` is one of the `DNS error codes`.

dns.setDefaultResultOrder(order)

- `order` `<string>` must be 'ipv4first' or 'verbatim'.

Set the default value of `verbatim` in `dns.lookup()` and `dnsPromises.lookup()`. The value could be:

- `ipv4first`: sets default `verbatim` `false`.
- `verbatim`: sets default `verbatim` `true`.

The default is `ipv4first` and `dns.setDefaultResultOrder()` have higher priority than `--dns-result-order`. When using `worker threads`, `dns.setDefaultResultOrder()` from the main thread won't affect the default dns orders in workers.

dns.setServers(servers)

- `servers` `<string[]>` array of `RFC 5952` formatted addresses

Sets the IP address and port of servers to be used when performing DNS resolution. The `servers` argument is an array of `RFC 5952` formatted addresses. If the port is the IANA default DNS port (53) it can be omitted.

```
dns.setServers([
  '4.4.4.4',
  '[2001:4860:4860::8888]',
  '4.4.4.4:1053',
  '[2001:4860:4860::8888]:1053',
]);
```

An error will be thrown if an invalid address is provided.

The `dns.setServers()` method must not be called while a DNS query is in progress.

The `dns.setServers()` method affects only `dns.resolve()`, `dns.resolve*()` and `dns.reverse()` (and specifically *not* `dns.lookup()`).

This method works much like `resolve.conf`. That is, if attempting to resolve with the first server provided results in a `NOTFOUND` error, the `resolve()` method will *not* attempt to resolve with subsequent servers provided. Fallback DNS servers will only be used if the earlier ones

time out or result in some other error.

DNS promises API

The `dns.promises` API provides an alternative set of asynchronous DNS methods that return `Promise` objects rather than using callbacks. The API is accessible via `require('dns').promises` or `require('dns/promises')`.

Class: `dnsPromises.Resolver`

An independent resolver for DNS requests.

Creating a new resolver uses the default server settings. Setting the servers used for a resolver using `resolver.setServers()` does not affect other resolvers:

```
const { Resolver } = require('dns').promises;
const resolver = new Resolver();
resolver.setServers(['4.4.4.4']);

// This request will use the server at 4.4.4.4, independent of global settings.
resolver.resolve4('example.org').then((addresses) => {
  // ...
});

// Alternatively, the same code can be written using async-await style.
(async function() {
  const addresses = await resolver.resolve4('example.org');
})();
```

The following methods from the `dnsPromises` API are available:

- `resolver.getServers()`
- `resolver.resolve()`
- `resolver.resolve4()`
- `resolver.resolve6()`
- `resolver.resolveAny()`
- `resolver.resolveCaa()`
- `resolver.resolveCname()`
- `resolver.resolveMx()`
- `resolver.resolveNaptr()`
- `resolver.resolveNs()`
- `resolver.resolvePtr()`
- `resolver.resolveSoa()`
- `resolver.resolveSrv()`
- `resolver.resolveTxt()`
- `resolver.reverse()`
- `resolver.setServers()`

resolver.cancel()

Cancel all outstanding DNS queries made by this resolver. The corresponding promises will be rejected with an error with code `ECANCELLED`.

dnsPromises.getServers()

- Returns: `<string[]>`

Returns an array of IP address strings, formatted according to [RFC 5952](#), that are currently configured for DNS resolution. A string will include a port section if a custom port is used.

```
[
  '4.4.4.4',
  '2001:4860:4860::8888',
  '4.4.4.4:1053',
  '[2001:4860:4860::8888]:1053',
]
```

dnsPromises.lookup(hostname[, options])

- `hostname` `<string>`
- `options` `<integer> | <Object>`
 - `family` `<integer>` The record family. Must be `4`, `6`, or `0`. The value `0` indicates that IPv4 and IPv6 addresses are both returned. **Default:** `0`.
 - `hints` `<number>` One or more [supported `getaddrinfo` flags](#). Multiple flags may be passed by bitwise **OR**ing their values.
 - `all` `<boolean>` When `true`, the `Promise` is resolved with all addresses in an array. Otherwise, returns a single address. **Default:** `false`.
 - `verbatim` `<boolean>` When `true`, the `Promise` is resolved with IPv4 and IPv6 addresses in the order the DNS resolver returned them. When `false`, IPv4 addresses are placed before IPv6 addresses. **Default:** currently `false` (addresses are reordered) but this is expected to change in the not too distant future. Default value is configurable using `dns.setDefaultResultOrder()` or `--dns-result-order`. New code should use `{ verbatim: true }`.

Resolves a host name (e.g. `'nodejs.org'`) into the first found A (IPv4) or AAAA (IPv6) record. All `option` properties are optional. If `options` is an integer, then it must be `4` or `6` – if `options` is not provided, then IPv4 and IPv6 addresses are both returned if found.

With the `all` option set to `true`, the `Promise` is resolved with `addresses` being an array of objects with the properties `address` and `family`.

On error, the `Promise` is rejected with an `Error` object, where `err.code` is the error code. Keep in mind that `err.code` will be set to `'ENOTFOUND'` not only when the host name does not exist but also when the lookup fails in other ways such as no available file descriptors.

`dnsPromises.lookup()` does not necessarily have anything to do with the DNS protocol. The implementation uses an operating system facility that can associate names with addresses, and vice versa. This implementation can have subtle but important consequences on the behavior of any Node.js program. Please take some time to consult the [Implementation considerations section](#) before using `dnsPromises.lookup()`.

Example usage:

```
const dns = require('dns');
const dnsPromises = dns.promises;
const options = {
  family: 6,
  hints: dns.ADDRCONFIG | dns.V4MAPPED,
```

```

});

dnsPromises.lookup('example.com', options).then((result) => {
  console.log('address: %j family: IPv%s', result.address, result.family);
  // address: "2606:2800:220:1:248:1893:25c8:1946" family: IPv6
});

// When options.all is true, the result will be an Array.
options.all = true;
dnsPromises.lookup('example.com', options).then((result) => {
  console.log('addresses: %j', result);
  // addresses: [{ "address": "2606:2800:220:1:248:1893:25c8:1946", "family": 6 }]
});

```

dnsPromises.lookupService(address, port)

- `address` `<string>`
- `port` `<number>`

Resolves the given `address` and `port` into a host name and service using the operating system's underlying `getnameinfo` implementation.

If `address` is not a valid IP address, a `TypeError` will be thrown. The `port` will be coerced to a number. If it is not a legal port, a `TypeError` will be thrown.

On error, the `Promise` is rejected with an `Error` object, where `err.code` is the error code.

```

const dnsPromises = require('dns').promises;
dnsPromises.lookupService('127.0.0.1', 22).then((result) => {
  console.log(result.hostname, result.service);
  // Prints: localhost ssh
});

```

dnsPromises.resolve(hostname[, rrtype])

- `hostname` `<string>` Host name to resolve.
- `rrtype` `<string>` Resource record type. **Default:** `'A'`.

Uses the DNS protocol to resolve a host name (e.g. `'nodejs.org'`) into an array of the resource records. When successful, the `Promise` is resolved with an array of resource records. The type and structure of individual results vary based on `rrtype`:

<code>rrtype</code>	<code>records</code> contains	Result type	Shorthand method
<code>'A'</code>	IPv4 addresses (default)	<code><string></code>	<code>dnsPromises.resolve4()</code>
<code>'AAAA'</code>	IPv6 addresses	<code><string></code>	<code>dnsPromises.resolve6()</code>
<code>'ANY'</code>	any records	<code><Object></code>	<code>dnsPromises.resolveAny()</code>
<code>'CAA'</code>	CA authorization records	<code><Object></code>	<code>dnsPromises.resolveCaa()</code>
<code>'CNAME'</code>	canonical name records	<code><string></code>	<code>dnsPromises.resolveCname()</code>
<code>'MX'</code>	mail exchange records	<code><Object></code>	<code>dnsPromises.resolveMx()</code>

rrtype	records contains	Result type	Shorthand method
'NAPTR'	name authority pointer records	<Object>	<code>dnsPromises.resolveNaptr()</code>
'NS'	name server records	<string>	<code>dnsPromises.resolveNs()</code>
'PTR'	pointer records	<string>	<code>dnsPromises.resolvePtr()</code>
'SOA'	start of authority records	<Object>	<code>dnsPromises.resolveSoa()</code>
'SRV'	service records	<Object>	<code>dnsPromises.resolveSrv()</code>
'TXT'	text records	<string[]>	<code>dnsPromises.resolveTxt()</code>

On error, the `Promise` is rejected with an `Error` object, where `err.code` is one of the `DNS error codes`.

dnsPromises.resolve4(hostname[, options])

- `hostname` <string> Host name to resolve.
- `options` <Object>
 - `ttl` <boolean> Retrieve the Time-To-Live value (TTL) of each record. When `true`, the `Promise` is resolved with an array of `{ address: '1.2.3.4', ttl: 60 }` objects rather than an array of strings, with the TTL expressed in seconds.

Uses the DNS protocol to resolve IPv4 addresses (A records) for the `hostname`. On success, the `Promise` is resolved with an array of IPv4 addresses (e.g. `['74.125.79.104', '74.125.79.105', '74.125.79.106']`).

dnsPromises.resolve6(hostname[, options])

- `hostname` <string> Host name to resolve.
- `options` <Object>
 - `ttl` <boolean> Retrieve the Time-To-Live value (TTL) of each record. When `true`, the `Promise` is resolved with an array of `{ address: '0:1:2:3:4:5:6:7', ttl: 60 }` objects rather than an array of strings, with the TTL expressed in seconds.

Uses the DNS protocol to resolve IPv6 addresses (AAAA records) for the `hostname`. On success, the `Promise` is resolved with an array of IPv6 addresses.

dnsPromises.resolveAny(hostname)

- `hostname` <string>

Uses the DNS protocol to resolve all records (also known as ANY or * query). On success, the `Promise` is resolved with an array containing various types of records. Each object has a property `type` that indicates the type of the current record. And depending on the `type`, additional properties will be present on the object:

Type	Properties
'A'	address / ttl
'AAAA'	address / ttl
'CNAME'	value
'MX'	Refer to <code>dnsPromises.resolveMx()</code>
'NAPTR'	Refer to <code>dnsPromises.resolveNaptr()</code>

Type	Properties
'NS'	value
'PTR'	value
'SOA'	Refer to <code>dnsPromises.resolveSoa()</code>
'SRV'	Refer to <code>dnsPromises.resolveSrv()</code>
'TXT'	This type of record contains an array property called <code>entries</code> which refers to <code>dnsPromises.resolveTxt()</code> , e.g. <code>{ entries: ['...'], type: 'TXT' }</code>

Here is an example of the result object:

```
[ { type: 'A', address: '127.0.0.1', ttl: 299 },
  { type: 'CNAME', value: 'example.com' },
  { type: 'MX', exchange: 'alt4.aspmx.l.example.com', priority: 50 },
  { type: 'NS', value: 'ns1.example.com' },
  { type: 'TXT', entries: [ 'v=spf1 include:_spf.example.com ~all' ] },
  { type: 'SOA',
    nsname: 'ns1.example.com',
    hostmaster: 'admin.example.com',
    serial: 156696742,
    refresh: 900,
    retry: 900,
    expire: 1800,
    minttl: 60 } ]
```

dnsPromises.resolveCaa(hostname)

- `hostname` `<string>`

Uses the DNS protocol to resolve CAA records for the `hostname`. On success, the `Promise` is resolved with an array of objects containing available certification authority authorization records available for the `hostname` (e.g. `[{critical: 0, iodef: 'mailto:pki@example.com'}, {critical: 128, issue: 'pki.example.com'}]`).

dnsPromises.resolveCname(hostname)

- `hostname` `<string>`

Uses the DNS protocol to resolve CNAME records for the `hostname`. On success, the `Promise` is resolved with an array of canonical name records available for the `hostname` (e.g. `['bar.example.com']`).

dnsPromises.resolveMx(hostname)

- `hostname` `<string>`

Uses the DNS protocol to resolve mail exchange records (MX records) for the `hostname`. On success, the `Promise` is resolved with an array of objects containing both a `priority` and `exchange` property (e.g. `[{priority: 10, exchange: 'mx.example.com'}, ...]`).

dnsPromises.resolveNaptr(hostname)

- `hostname` `<string>`

Uses the DNS protocol to resolve regular expression based records (NAPTR records) for the `hostname` . On success, the `Promise` is resolved with an array of objects with the following properties:

- `flags`
- `service`
- `regexp`
- `replacement`
- `order`
- `preference`

```
{
  flags: 's',
  service: 'SIP+D2U',
  regexp: '',
  replacement: '_sip._udp.example.com',
  order: 30,
  preference: 100
}
```

`dnsPromises.resolveNs(hostname)`

- `hostname` `<string>`

Uses the DNS protocol to resolve name server records (NS records) for the `hostname` . On success, the `Promise` is resolved with an array of name server records available for `hostname` (e.g. `['ns1.example.com', 'ns2.example.com']`).

`dnsPromises.resolvePtr(hostname)`

- `hostname` `<string>`

Uses the DNS protocol to resolve pointer records (PTR records) for the `hostname` . On success, the `Promise` is resolved with an array of strings containing the reply records.

`dnsPromises.resolveSoa(hostname)`

- `hostname` `<string>`

Uses the DNS protocol to resolve a start of authority record (SOA record) for the `hostname` . On success, the `Promise` is resolved with an object with the following properties:

- `nsname`
- `hostmaster`
- `serial`
- `refresh`
- `retry`
- `expire`
- `minttl`

```
{
  nsname: 'ns.example.com',
  hostmaster: 'root.example.com',
}
```



```
serial: 2013101809,
refresh: 10000,
retry: 2400,
expire: 604800,
minttl: 3600
}
```

dnsPromises.resolveSrv(hostname)

- `hostname` `<string>`

Uses the DNS protocol to resolve service records (`SRV` records) for the `hostname` . On success, the `Promise` is resolved with an array of objects with the following properties:

- `priority`
- `weight`
- `port`
- `name`

```
{
  priority: 10,
  weight: 5,
  port: 21223,
  name: 'service.example.com'
}
```

dnsPromises.resolveTxt(hostname)

- `hostname` `<string>`

Uses the DNS protocol to resolve text queries (`TXT` records) for the `hostname` . On success, the `Promise` is resolved with a two-dimensional array of the text records available for `hostname` (e.g. [['v=spf1 ip4:0.0.0.0 ', '~all']]). Each sub-array contains TXT chunks of one record. Depending on the use case, these could be either joined together or treated separately.

dnsPromises.reverse(ip)

- `ip` `<string>`

Performs a reverse DNS query that resolves an IPv4 or IPv6 address to an array of host names.

On error, the `Promise` is rejected with an `Error` object, where `err.code` is one of the `DNS error codes` .

dnsPromises.setDefaultResultOrder(order)

- `order` `<string>` must be `'ipv4first'` or `'verbatim'` .

Set the default value of `verbatim` in `dns.lookup()` and `dnsPromises.lookup()` . The value could be:

- `ipv4first`: sets default `verbatim` `false` .
- `verbatim`: sets default `verbatim` `true` .

The default is `ipv4first` and `dnsPromises.setDefaultResultOrder()` have higher priority than `--dns-result-order` . When using `worker threads` , `dnsPromises.setDefaultResultOrder()` from the main thread won't affect the default dns orders in workers.

dnsPromises.setServers(servers)

- `servers` `<string[]>` array of [RFC 5952](#) formatted addresses

Sets the IP address and port of servers to be used when performing DNS resolution. The `servers` argument is an array of [RFC 5952](#) formatted addresses. If the port is the IANA default DNS port (53) it can be omitted.

```
dnsPromises.setServers([
  '4.4.4.4',
  '[2001:4860:4860::8888]',
  '4.4.4.4:1053',
  '[2001:4860:4860::8888]:1053',
]);
```

An error will be thrown if an invalid address is provided.

The `dnsPromises.setServers()` method must not be called while a DNS query is in progress.

This method works much like [resolve.conf](#). That is, if attempting to resolve with the first server provided results in a `NOTFOUND` error, the `resolve()` method will *not* attempt to resolve with subsequent servers provided. Fallback DNS servers will only be used if the earlier ones time out or result in some other error.

Error codes

Each DNS query can return one of the following error codes:

- `dns.NODATA` : DNS server returned answer with no data.
- `dns.FORMERR` : DNS server claims query was misformatted.
- `dns.SERVFAIL` : DNS server returned general failure.
- `dns.NOTFOUND` : Domain name not found.
- `dns.NOTIMP` : DNS server does not implement requested operation.
- `dns.REFUSED` : DNS server refused query.
- `dns.BADQUERY` : Misformatted DNS query.
- `dns.BADNAME` : Misformatted host name.
- `dns.BADFAMILY` : Unsupported address family.
- `dns.BADRESP` : Misformatted DNS reply.
- `dns.CONNREFUSED` : Could not contact DNS servers.
- `dns.TIMEOUT` : Timeout while contacting DNS servers.
- `dns.EOF` : End of file.
- `dns.FILE` : Error reading file.
- `dns.NOMEM` : Out of memory.
- `dns.DESTRUCTION` : Channel is being destroyed.
- `dns.BADSTR` : Misformatted string.
- `dns.BADFLAGS` : Illegal flags specified.
- `dns.NONAME` : Given host name is not numeric.
- `dns.BADHINTS` : Illegal hints flags specified.
- `dns.NOTINITIALIZED` : c-ares library initialization not yet performed.
- `dns.LOADIPHLPAPI` : Error loading `iphlpapi.dll`.

- `dns.ADDRGETNETWORKPARAMS` : Could not find `GetNetworkParams` function.
- `dns.CANCELLED` : DNS query cancelled.

Implementation considerations

Although `dns.lookup()` and the various `dns.resolve*()/dns.reverse()` functions have the same goal of associating a network name with a network address (or vice versa), their behavior is quite different. These differences can have subtle but significant consequences on the behavior of Node.js programs.

`dns.lookup()`

Under the hood, `dns.lookup()` uses the same operating system facilities as most other programs. For instance, `dns.lookup()` will almost always resolve a given name the same way as the `ping` command. On most POSIX-like operating systems, the behavior of the `dns.lookup()` function can be modified by changing settings in `nsswitch.conf(5)` and/or `resolv.conf(5)`, but changing these files will change the behavior of all other programs running on the same operating system.

Though the call to `dns.lookup()` will be asynchronous from JavaScript's perspective, it is implemented as a synchronous call to `getaddrinfo(3)` that runs on libuv's threadpool. This can have surprising negative performance implications for some applications, see the `UV_THREADPOOL_SIZE` documentation for more information.

Various networking APIs will call `dns.lookup()` internally to resolve host names. If that is an issue, consider resolving the host name to an address using `dns.resolve()` and using the address instead of a host name. Also, some networking APIs (such as `socket.connect()` and `dgram.createSocket()`) allow the default resolver, `dns.lookup()`, to be replaced.

`dns.resolve()`, `dns.resolve*()` and `dns.reverse()`

These functions are implemented quite differently than `dns.lookup()`. They do not use `getaddrinfo(3)` and they *always* perform a DNS query on the network. This network communication is always done asynchronously, and does not use libuv's threadpool.

As a result, these functions cannot have the same negative impact on other processing that happens on libuv's threadpool that `dns.lookup()` can have.

They do not use the same set of configuration files than what `dns.lookup()` uses. For instance, *they do not use the configuration from `/etc/hosts`.*

Domain

Stability: 0 - Deprecated

Source Code: [lib/domain.js](#)

This module is pending deprecation. Once a replacement API has been finalized, this module will be fully deprecated. Most developers should **not** have cause to use this module. Users who absolutely must have the functionality that domains provide may rely on it for the time being but should expect to have to migrate to a different solution in the future.

Domains provide a way to handle multiple different IO operations as a single group. If any of the event emitters or callbacks registered to a domain emit an `'error'` event, or throw an error, then the domain object will be notified, rather than losing the context of the error in the `process.on('uncaughtException')` handler, or causing the program to exit immediately with an error code.

Warning: Don't ignore errors!

Domain error handlers are not a substitute for closing down a process when an error occurs.

By the very nature of how `throw` works in JavaScript, there is almost never any way to safely "pick up where it left off", without leaking references, or creating some other sort of undefined brittle state.

The safest way to respond to a thrown error is to shut down the process. Of course, in a normal web server, there may be many open connections, and it is not reasonable to abruptly shut those down because an error was triggered by someone else.

The better approach is to send an error response to the request that triggered the error, while letting the others finish in their normal time, and stop listening for new requests in that worker.

In this way, `domain` usage goes hand-in-hand with the `cluster` module, since the primary process can fork a new worker when a worker encounters an error. For Node.js programs that scale to multiple machines, the terminating proxy or service registry can take note of the failure, and react accordingly.

For example, this is not a good idea:

```
// XXX WARNING! BAD IDEA!

const d = require('domain').create();
d.on('error', (er) => {
  // The error won't crash the process, but what it does is worse!
  // Though we've prevented abrupt process restarting, we are leaking
  // a lot of resources if this ever happens.
  // This is no better than process.on('uncaughtException')!
  console.log(`error, but oh well ${er.message}`);
});
d.run(() => {
  require('http').createServer((req, res) => {
    handleRequest(req, res);
  }).listen(PORT);
});
```

By using the context of a domain, and the resilience of separating our program into multiple worker processes, we can react more appropriately, and handle errors with much greater safety.

```
// Much better!

const cluster = require('cluster');
const PORT = +process.env.PORT || 1337;

if (cluster.isPrimary) {
  // A more realistic scenario would have more than 2 workers,
  // and perhaps not put the primary and worker in the same file.
  //
  // It is also possible to get a bit fancier about logging, and
  // implement whatever custom logic is needed to prevent DoS
  // attacks and other bad behavior.
  //
  // See the options in the cluster documentation.
  //
  // The important thing is that the primary does very little,
```

```

// increasing our resilience to unexpected errors.

cluster.fork();
cluster.fork();

cluster.on('disconnect', (worker) => {
  console.error('disconnect!');
  cluster.fork();
});

} else {
  // the worker
  //
  // This is where we put our bugs!

  const domain = require('domain');

  // See the cluster documentation for more details about using
  // worker processes to serve requests. How it works, caveats, etc.

  const server = require('http').createServer((req, res) => {
    const d = domain.create();
    d.on('error', (er) => {
      console.error(`error ${er.stack}`);

      // We're in dangerous territory!
      // By definition, something unexpected occurred,
      // which we probably didn't want.
      // Anything can happen now! Be very careful!

      try {
        // Make sure we close down within 30 seconds
        const killtimer = setTimeout(() => {
          process.exit(1);
        }, 30000);
        // But don't keep the process open just for that!
        killtimer.unref();

        // Stop taking new requests.
        server.close();

        // Let the primary know we're dead. This will trigger a
        // 'disconnect' in the cluster primary, and then it will fork
        // a new worker.
        cluster.worker.disconnect();

        // Try to send an error to the request that triggered the problem
        res.statusCode = 500;
        res.setHeader('content-type', 'text/plain');
        res.end('Oops, there was a problem!\n');
      } catch (er2) {

```

```

    // Oh well, not much we can do at this point.
    console.error(`Error sending 500! ${er2.stack}`);
  }
});

// Because req and res were created before this domain existed,
// we need to explicitly add them.
// See the explanation of implicit vs explicit binding below.
d.add(req);
d.add(res);

// Now run the handler function in the domain.
d.run(() => {
  handleRequest(req, res);
});
server.listen(PORT);
}

// This part is not important. Just an example routing thing.
// Put fancy application logic here.
function handleRequest(req, res) {
  switch (req.url) {
    case '/error':
      // We do some async stuff, and then...
      setTimeout(() => {
        // Whoops!
        flerb.bark();
      }, timeout);
      break;
    default:
      res.end('ok');
  }
}

```

Additions to Error objects

Any time an `Error` object is routed through a domain, a few extra fields are added to it.

- `error.domain` The domain that first handled the error.
- `error.domainEmitter` The event emitter that emitted an `'error'` event with the error object.
- `error.domainBound` The callback function which was bound to the domain, and passed an error as its first argument.
- `error.domainThrown` A boolean indicating whether the error was thrown, emitted, or passed to a bound callback function.

Implicit binding

If domains are in use, then all `new EventEmitter` objects (including Stream objects, requests, responses, etc.) will be implicitly bound to the active domain at the time of their creation.

Additionally, callbacks passed to lowlevel event loop requests (such as to `fs.open()`, or other callback-taking methods) will automatically be bound to the active domain. If they throw, then the domain will catch the error.

In order to prevent excessive memory usage, `Domain` objects themselves are not implicitly added as children of the active domain. If they were, then it would be too easy to prevent request and response objects from being properly garbage collected.

To nest `Domain` objects as children of a parent `Domain` they must be explicitly added.

Implicit binding routes thrown errors and `'error'` events to the `Domain`'s `'error'` event, but does not register the `EventEmitter` on the `Domain`. Implicit binding only takes care of thrown errors and `'error'` events.

Explicit binding

Sometimes, the domain in use is not the one that ought to be used for a specific event emitter. Or, the event emitter could have been created in the context of one domain, but ought to instead be bound to some other domain.

For example, there could be one domain in use for an HTTP server, but perhaps we would like to have a separate domain to use for each request.

That is possible via explicit binding.

```
// Create a top-level domain for the server
const domain = require('domain');
const http = require('http');
const serverDomain = domain.create();

serverDomain.run(() => {
  // Server is created in the scope of serverDomain
  http.createServer((req, res) => {
    // Req and res are also created in the scope of serverDomain
    // however, we'd prefer to have a separate domain for each request.
    // create it first thing, and add req and res to it.
    const reqd = domain.create();
    reqd.add(req);
    reqd.add(res);
    reqd.on('error', (er) => {
      console.error('Error', er, req.url);
      try {
        res.writeHead(500);
        res.end('Error occurred, sorry.');
```

domain.create()

- Returns: `<Domain>`

Class: Domain

- Extends: `<EventEmitter>`

The `Domain` class encapsulates the functionality of routing errors and uncaught exceptions to the active `Domain` object.

To handle the errors that it catches, listen to its `'error'` event.

domain.members

- `<Array>`

An array of timers and event emitters that have been explicitly added to the domain.

domain.add(emitter)

- `emitter` `<EventEmitter>` | `<Timer>` emitter or timer to be added to the domain

Explicitly adds an emitter to the domain. If any event handlers called by the emitter throw an error, or if the emitter emits an `'error'` event, it will be routed to the domain's `'error'` event, just like with implicit binding.

This also works with timers that are returned from `setInterval()` and `setTimeout()`. If their callback function throws, it will be caught by the domain `'error'` handler.

If the Timer or `EventEmitter` was already bound to a domain, it is removed from that one, and bound to this one instead.

domain.bind(callback)

- `callback` `<Function>` The callback function
- Returns: `<Function>` The bound function

The returned function will be a wrapper around the supplied callback function. When the returned function is called, any errors that are thrown will be routed to the domain's `'error'` event.

```
const d = domain.create();

function readSomeFile(filename, cb) {
  fs.readFile(filename, 'utf8', d.bind((er, data) => {
    // If this throws, it will also be passed to the domain.
    return cb(er, data ? JSON.parse(data) : null);
  }));
}

d.on('error', (er) => {
  // An error occurred somewhere. If we throw it now, it will crash the program
  // with the normal line number and stack message.
});
```

domain.enter()

The `enter()` method is plumbing used by the `run()`, `bind()`, and `intercept()` methods to set the active domain. It sets `domain.active` and `process.domain` to the domain, and implicitly pushes the domain onto the domain stack managed by the domain module (see `domain.exit()` for details on the domain stack). The call to `enter()` delimits the beginning of a chain of asynchronous calls and I/O operations bound to a domain.

Calling `enter()` changes only the active domain, and does not alter the domain itself. `enter()` and `exit()` can be called an arbitrary number of times on a single domain.

domain.exit()

The `exit()` method exits the current domain, popping it off the domain stack. Any time execution is going to switch to the context of a different chain of asynchronous calls, it's important to ensure that the current domain is exited. The call to `exit()` delimits either the end of or an interruption to the chain of asynchronous calls and I/O operations bound to a domain.

If there are multiple, nested domains bound to the current execution context, `exit()` will exit any domains nested within this domain.

Calling `exit()` changes only the active domain, and does not alter the domain itself. `enter()` and `exit()` can be called an arbitrary number of times on a single domain.

domain.intercept(callback)

- `callback` `<Function>` The callback function
- Returns: `<Function>` The intercepted function

This method is almost identical to `domain.bind(callback)`. However, in addition to catching thrown errors, it will also intercept `Error` objects sent as the first argument to the function.

In this way, the common `if (err) return callback(err);` pattern can be replaced with a single error handler in a single place.

```
const d = domain.create();

function readSomeFile(filename, cb) {
  fs.readFile(filename, 'utf8', d.intercept((data) => {
    // Note, the first argument is never passed to the
    // callback since it is assumed to be the 'Error' argument
    // and thus intercepted by the domain.

    // If this throws, it will also be passed to the domain
    // so the error-handling logic can be moved to the 'error'
    // event on the domain instead of being repeated throughout
    // the program.
    return cb(null, JSON.parse(data));
  }));
}

d.on('error', (er) => {
  // An error occurred somewhere. If we throw it now, it will crash the program
  // with the normal line number and stack message.
});
```

domain.remove(emitter)

- `emitter` `<EventEmitter>` | `<Timer>` emitter or timer to be removed from the domain

The opposite of `domain.add(emitter)`. Removes domain handling from the specified emitter.

domain.run(fn[, ...args])

- `fn` `<Function>`

- `...args` `<any>`

Run the supplied function in the context of the domain, implicitly binding all event emitters, timers, and lowlevel requests that are created in that context. Optionally, arguments can be passed to the function.

This is the most basic way to use a domain.

```
const domain = require('domain');
const fs = require('fs');
const d = domain.create();
d.on('error', (er) => {
  console.error('Caught error!', er);
});
d.run(() => {
  process.nextTick(() => {
    setTimeout(() => { // Simulating some various async stuff
      fs.open('non-existent file', 'r', (er, fd) => {
        if (er) throw er;
        // proceed...
      });
    }, 100);
  });
});
```

In this example, the `d.on('error')` handler will be triggered, rather than crashing the program.

Domains and promises

As of Node.js 8.0.0, the handlers of promises are run inside the domain in which the call to `.then()` or `.catch()` itself was made:

```
const d1 = domain.create();
const d2 = domain.create();

let p;
d1.run(() => {
  p = Promise.resolve(42);
});

d2.run(() => {
  p.then((v) => {
    // running in d2
  });
});
```

A callback may be bound to a specific domain using `domain.bind(callback)`:

```
const d1 = domain.create();
const d2 = domain.create();

let p;
```

```

d1.run(() => {
  p = Promise.resolve(42);
});

d2.run(() => {
  p.then(p.domain.bind((v) => {
    // running in d1
  }));
});

```

Domains will not interfere with the error handling mechanisms for promises. In other words, no `'error'` event will be emitted for unhandled `Promise` rejections.

Errors

Applications running in Node.js will generally experience four categories of errors:

- Standard JavaScript errors such as `<EvalError>`, `<SyntaxError>`, `<RangeError>`, `<ReferenceError>`, `<TypeError>`, and `<URIError>`.
- System errors triggered by underlying operating system constraints such as attempting to open a file that does not exist or attempting to send data over a closed socket.
- User-specified errors triggered by application code.
- `AssertionError`s are a special class of error that can be triggered when Node.js detects an exceptional logic violation that should never occur. These are raised typically by the `assert` module.

All JavaScript and system errors raised by Node.js inherit from, or are instances of, the standard JavaScript `<Error>` class and are guaranteed to provide *at least* the properties available on that class.

Error propagation and interception

Node.js supports several mechanisms for propagating and handling errors that occur while an application is running. How these errors are reported and handled depends entirely on the type of `Error` and the style of the API that is called.

All JavaScript errors are handled as exceptions that *immediately* generate and throw an error using the standard JavaScript `throw` mechanism. These are handled using the `try...catch` construct provided by the JavaScript language.

```

// Throws with a ReferenceError because z is not defined.
try {
  const m = 1;
  const n = m + z;
} catch (err) {
  // Handle the error here.
}

```

Any use of the JavaScript `throw` mechanism will raise an exception that *must* be handled using `try...catch` or the Node.js process will exit immediately.

With few exceptions, *Synchronous* APIs (any blocking method that does not accept a `callback` function, such as `fs.readFileSync`), will use `throw` to report errors.

Errors that occur within *Asynchronous APIs* may be reported in multiple ways:

- Most asynchronous methods that accept a `callback` function will accept an `Error` object passed as the first argument to that function. If that first argument is not `null` and is an instance of `Error`, then an error occurred that should be handled.

```
const fs = require('fs');
fs.readFile('a file that does not exist', (err, data) => {
  if (err) {
    console.error('There was an error reading the file!', err);
    return;
  }
  // Otherwise handle the data
});
```

- When an asynchronous method is called on an object that is an `EventEmitter`, errors can be routed to that object's `'error'` event.

```
const net = require('net');
const connection = net.connect('localhost');

// Adding an 'error' event handler to a stream:
connection.on('error', (err) => {
  // If the connection is reset by the server, or if it can't
  // connect at all, or on any sort of error encountered by
  // the connection, the error will be sent here.
  console.error(err);
});

connection.pipe(process.stdout);
```

- A handful of typically asynchronous methods in the Node.js API may still use the `throw` mechanism to raise exceptions that must be handled using `try...catch`. There is no comprehensive list of such methods; please refer to the documentation of each method to determine the appropriate error handling mechanism required.

The use of the `'error'` event mechanism is most common for `stream-based` and `event emitter-based` APIs, which themselves represent a series of asynchronous operations over time (as opposed to a single operation that may pass or fail).

For *all* `EventEmitter` objects, if an `'error'` event handler is not provided, the error will be thrown, causing the Node.js process to report an uncaught exception and crash unless either: The `domain` module is used appropriately or a handler has been registered for the `'uncaughtException'` event.

```
const EventEmitter = require('events');
const ee = new EventEmitter();

setImmediate(() => {
  // This will crash the process because no 'error' event
  // handler has been added.
  ee.emit('error', new Error('This will crash'));
});
```

Errors generated in this way *cannot* be intercepted using `try...catch` as they are thrown *after* the calling code has already exited.

Developers must refer to the documentation for each method to determine exactly how errors raised by those methods are propagated.

Error-first callbacks

Most asynchronous methods exposed by the Node.js core API follow an idiomatic pattern referred to as an *error-first callback*. With this pattern, a callback function is passed to the method as an argument. When the operation either completes or an error is raised, the callback function is called with the `Error` object (if any) passed as the first argument. If no error was raised, the first argument will be passed as `null`.

```
const fs = require('fs');

function errorFirstCallback(err, data) {
  if (err) {
    console.error('There was an error', err);
    return;
  }
  console.log(data);
}

fs.readFile('/some/file/that/does-not-exist', errorFirstCallback);
fs.readFile('/some/file/that/does-exist', errorFirstCallback);
```

The JavaScript `try...catch` mechanism **cannot** be used to intercept errors generated by asynchronous APIs. A common mistake for beginners is to try to use `throw` inside an error-first callback:

```
// THIS WILL NOT WORK:
const fs = require('fs');

try {
  fs.readFile('/some/file/that/does-not-exist', (err, data) => {
    // Mistaken assumption: throwing here...
    if (err) {
      throw err;
    }
  });
} catch (err) {
  // This will not catch the throw!
  console.error(err);
}
```

This will not work because the callback function passed to `fs.readFile()` is called asynchronously. By the time the callback has been called, the surrounding code, including the `try...catch` block, will have already exited. Throwing an error inside the callback **can crash the Node.js process** in most cases. If `domains` are enabled, or a handler has been registered with `process.on('uncaughtException')`, such errors can be intercepted.

Class: `Error`

A generic JavaScript `<Error>` object that does not denote any specific circumstance of why the error occurred. `Error` objects capture a "stack trace" detailing the point in the code at which the `Error` was instantiated, and may provide a text description of the error.

All errors generated by Node.js, including all system and JavaScript errors, will either be instances of, or inherit from, the `Error` class.

`new Error(message)`

- `message` `<string>`

Creates a new `Error` object and sets the `error.message` property to the provided text message. If an object is passed as `message`, the text message is generated by calling `message.toString()`. The `error.stack` property will represent the point in the code at which `new Error()` was called. Stack traces are dependent on [V8's stack trace API](#). Stack traces extend only to either (a) the beginning of *synchronous code execution*, or (b) the number of frames given by the property `Error.stackTraceLimit`, whichever is smaller.

Error.captureStackTrace(targetObject[, constructorOpt])

- `targetObject` `<Object>`
- `constructorOpt` `<Function>`

Creates a `.stack` property on `targetObject`, which when accessed returns a string representing the location in the code at which `Error.captureStackTrace()` was called.

```
const myObject = {};
Error.captureStackTrace(myObject);
myObject.stack; // Similar to `new Error().stack`
```

The first line of the trace will be prefixed with `${myObject.name}: ${myObject.message}`.

The optional `constructorOpt` argument accepts a function. If given, all frames above `constructorOpt`, including `constructorOpt`, will be omitted from the generated stack trace.

The `constructorOpt` argument is useful for hiding implementation details of error generation from the user. For instance:

```
function MyError() {
  Error.captureStackTrace(this, MyError);
}

// Without passing MyError to captureStackTrace, the MyError
// frame would show up in the .stack property. By passing
// the constructor, we omit that frame, and retain all frames below it.
new MyError().stack;
```

Error.stackTraceLimit

- `<number>`

The `Error.stackTraceLimit` property specifies the number of stack frames collected by a stack trace (whether generated by `new Error().stack` or `Error.captureStackTrace(obj)`).

The default value is `10` but may be set to any valid JavaScript number. Changes will affect any stack trace captured *after* the value has been changed.

If set to a non-number value, or set to a negative number, stack traces will not capture any frames.

error.code

- `<string>`

The `error.code` property is a string label that identifies the kind of error. `error.code` is the most stable way to identify an error. It will only change between major versions of Node.js. In contrast, `error.message` strings may change between any versions of Node.js. See [Node.js error codes](#) for details about specific codes.

error.message

- `<string>`

The `error.message` property is the string description of the error as set by calling `new Error(message)`. The `message` passed to the constructor will also appear in the first line of the stack trace of the `Error`, however changing this property after the `Error` object is created *may not* change the first line of the stack trace (for example, when `error.stack` is read before this property is changed).

```
const err = new Error('The message');
console.error(err.message);
// Prints: The message
```

error.stack

- `<string>`

The `error.stack` property is a string describing the point in the code at which the `Error` was instantiated.

```
Error: Things keep happening!
  at /home/gbusey/file.js:525:2
  at Frobnicator.refrobulate (/home/gbusey/business-logic.js:424:21)
  at Actor.<anonymous> (/home/gbusey/actors.js:400:8)
  at increaseSynergy (/home/gbusey/actors.js:701:6)
```

The first line is formatted as `<error class name>: <error message>`, and is followed by a series of stack frames (each line beginning with "at "). Each frame describes a call site within the code that lead to the error being generated. V8 attempts to display a name for each function (by variable name, function name, or object method name), but occasionally it will not be able to find a suitable name. If V8 cannot determine a name for the function, only location information will be displayed for that frame. Otherwise, the determined function name will be displayed with location information appended in parentheses.

Frames are only generated for JavaScript functions. If, for example, execution synchronously passes through a C++ addon function called `cheetahify` which itself calls a JavaScript function, the frame representing the `cheetahify` call will not be present in the stack traces:

```
const cheetahify = require('./native-binding.node');

function makeFaster() {
  // `cheetahify()` *synchronously* calls speedy.
  cheetahify(function speedy() {
    throw new Error('oh no!');
  });
}

makeFaster();
// will throw:
//   /home/gbusey/file.js:6
//     throw new Error('oh no!');
//           ^
//   Error: oh no!
//     at speedy (/home/gbusey/file.js:6:11)
//     at makeFaster (/home/gbusey/file.js:5:3)
//     at Object.<anonymous> (/home/gbusey/file.js:10:1)
//     at Module._compile (module.js:456:26)
```

```
//      at Object.Module._extensions..js (module.js:474:10)
//      at Module.load (module.js:356:32)
//      at Function.Module._load (module.js:312:12)
//      at Function.Module.runMain (module.js:497:10)
//      at startup (node.js:119:16)
//      at node.js:906:3
```

The location information will be one of:

- `native`, if the frame represents a call internal to V8 (as in `[].forEach`).
- `plain-filename.js:line:column`, if the frame represents a call internal to Node.js.
- `/absolute/path/to/file.js:line:column`, if the frame represents a call in a user program, or its dependencies.

The string representing the stack trace is lazily generated when the `error.stack` property is **accessed**.

The number of frames captured by the stack trace is bounded by the smaller of `Error.stackTraceLimit` or the number of available frames on the current event loop tick.

Class: `AssertionError`

- Extends: `<errors.Error>`

Indicates the failure of an assertion. For details, see [Class: `assert.AssertionError`](#).

Class: `RangeError`

- Extends: `<errors.Error>`

Indicates that a provided argument was not within the set or range of acceptable values for a function; whether that is a numeric range, or outside the set of options for a given function parameter.

```
require('net').connect(-1);
// Throws "RangeError: "port" option should be >= 0 and < 65536: -1"
```

Node.js will generate and throw `RangeError` instances *immediately* as a form of argument validation.

Class: `ReferenceError`

- Extends: `<errors.Error>`

Indicates that an attempt is being made to access a variable that is not defined. Such errors commonly indicate typos in code, or an otherwise broken program.

While client code may generate and propagate these errors, in practice, only V8 will do so.

```
doesNotExist;
// Throws ReferenceError, doesNotExist is not a variable in this program.
```

Unless an application is dynamically generating and running code, `ReferenceError` instances indicate a bug in the code or its dependencies.

Class: `SyntaxError`

- Extends: `<errors.Error>`

Indicates that a program is not valid JavaScript. These errors may only be generated and propagated as a result of code evaluation. Code evaluation may happen as a result of `eval`, `Function`, `require`, or `vm`. These errors are almost always indicative of a broken program.

```
try {
  require('vm').runInThisContext('binary ! isNotOk');
} catch (err) {
  // 'err' will be a SyntaxError.
}
```

`SyntaxError` instances are unrecoverable in the context that created them – they may only be caught by other contexts.

Class: `SystemError`

- Extends: `<errors.Error>`

Node.js generates system errors when exceptions occur within its runtime environment. These usually occur when an application violates an operating system constraint. For example, a system error will occur if an application attempts to read a file that does not exist.

- `address` `<string>` If present, the address to which a network connection failed
- `code` `<string>` The string error code
- `dest` `<string>` If present, the file path destination when reporting a file system error
- `errno` `<number>` The system-provided error number
- `info` `<Object>` If present, extra details about the error condition
- `message` `<string>` A system-provided human-readable description of the error
- `path` `<string>` If present, the file path when reporting a file system error
- `port` `<number>` If present, the network connection port that is not available
- `syscall` `<string>` The name of the system call that triggered the error

`error.address`

- `<string>`

If present, `error.address` is a string describing the address to which a network connection failed.

`error.code`

- `<string>`

The `error.code` property is a string representing the error code.

`error.dest`

- `<string>`

If present, `error.dest` is the file path destination when reporting a file system error.

`error.errno`

- `<number>`

The `error.errno` property is a negative number which corresponds to the error code defined in `libuv Error handling`.

On Windows the error number provided by the system will be normalized by libuv.

To get the string representation of the error code, use `util.getSystemErrorMessage(error.errno)`.

error.info

- `<Object>`

If present, `error.info` is an object with details about the error condition.

error.message

- `<string>`

`error.message` is a system-provided human-readable description of the error.

error.path

- `<string>`

If present, `error.path` is a string containing a relevant invalid pathname.

error.port

- `<number>`

If present, `error.port` is the network connection port that is not available.

error.syscall

- `<string>`

The `error.syscall` property is a string describing the `syscall` that failed.

Common system errors

This is a list of system errors commonly-encountered when writing a Node.js program. For a comprehensive list, see the [errno\(3\) man page](#).

- **EACCES** (Permission denied): An attempt was made to access a file in a way forbidden by its file access permissions.
- **EADDRINUSE** (Address already in use): An attempt to bind a server (`net` , `http` , or `https`) to a local address failed due to another server on the local system already occupying that address.
- **ECONNREFUSED** (Connection refused): No connection could be made because the target machine actively refused it. This usually results from trying to connect to a service that is inactive on the foreign host.
- **ECONNRESET** (Connection reset by peer): A connection was forcibly closed by a peer. This normally results from a loss of the connection on the remote socket due to a timeout or reboot. Commonly encountered via the `http` and `net` modules.
- **EEXIST** (File exists): An existing file was the target of an operation that required that the target not exist.
- **EISDIR** (Is a directory): An operation expected a file, but the given pathname was a directory.
- **EMFILE** (Too many open files in system): Maximum number of `file descriptors` allowable on the system has been reached, and requests for another descriptor cannot be fulfilled until at least one has been closed. This is encountered when opening many files at once in parallel, especially on systems (in particular, macOS) where there is a low file descriptor limit for processes. To remedy a low limit, run `ulimit -n 2048` in the same shell that will run the Node.js process.
- **ENOENT** (No such file or directory): Commonly raised by `fs` operations to indicate that a component of the specified pathname does not exist. No entity (file or directory) could be found by the given path.

- **ENOTDIR** (Not a directory): A component of the given pathname existed, but was not a directory as expected. Commonly raised by `fs.readdir`.
- **ENOTEMPTY** (Directory not empty): A directory with entries was the target of an operation that requires an empty directory, usually `fs.unlink`.
- **ENOTFOUND** (DNS lookup failed): Indicates a DNS failure of either `EAI_NODATA` or `EAI_NONAME`. This is not a standard POSIX error.
- **EPERM** (Operation not permitted): An attempt was made to perform an operation that requires elevated privileges.
- **EPIPE** (Broken pipe): A write on a pipe, socket, or FIFO for which there is no process to read the data. Commonly encountered at the `net` and `http` layers, indicative that the remote side of the stream being written to has been closed.
- **ETIMEDOUT** (Operation timed out): A connect or send request failed because the connected party did not properly respond after a period of time. Usually encountered by `http` or `net`. Often a sign that a `socket.end()` was not properly called.

Class: **TypeError**

- Extends `<errors.Error>`

Indicates that a provided argument is not an allowable type. For example, passing a function to a parameter which expects a string would be a `TypeError`.

```
require('url').parse(() => { });
// Throws TypeError, since it expected a string.
```

Node.js will generate and throw `TypeError` instances *immediately* as a form of argument validation.

Exceptions vs. errors

A JavaScript exception is a value that is thrown as a result of an invalid operation or as the target of a `throw` statement. While it is not required that these values are instances of `Error` or classes which inherit from `Error`, all exceptions thrown by Node.js or the JavaScript runtime *will* be instances of `Error`.

Some exceptions are *unrecoverable* at the JavaScript layer. Such exceptions will *always* cause the Node.js process to crash. Examples include `assert()` checks or `abort()` calls in the C++ layer.

OpenSSL errors

Errors originating in `crypto` or `tls` are of class `Error`, and in addition to the standard `.code` and `.message` properties, may have some additional OpenSSL-specific properties.

error.opensslErrorStack

An array of errors that can give context to where in the OpenSSL library an error originates from.

error.function

The OpenSSL function the error originates in.

error.library

The OpenSSL library the error originates in.

error.reason

A human-readable string describing the reason for the error.

Node.js error codes

ABORT_ERR

Used when an operation has been aborted (typically using an `AbortController`).

APIs *not* using `AbortSignal`s typically do not raise an error with this code.

This code does not use the regular `ERR_*` convention Node.js errors use in order to be compatible with the web platform's `AbortError`.

ERR_AMBIGUOUS_ARGUMENT

A function argument is being used in a way that suggests that the function signature may be misunderstood. This is thrown by the `assert` module when the `message` parameter in `assert.throws(block, message)` matches the error message thrown by `block` because that usage suggests that the user believes `message` is the expected message rather than the message the `AssertionError` will display if `block` does not throw.

ERR_ARG_NOT_ITERABLE

An iterable argument (i.e. a value that works with `for...of` loops) was required, but not provided to a Node.js API.

ERR_ASSERTION

A special type of error that can be triggered whenever Node.js detects an exceptional logic violation that should never occur. These are raised typically by the `assert` module.

ERR_ASYNC_CALLBACK

An attempt was made to register something that is not a function as an `AsyncHooks` callback.

ERR_ASYNC_TYPE

The type of an asynchronous resource was invalid. Users are also able to define their own types if using the public embedder API.

ERR_BROTLI_COMPRESSION_FAILED

Data passed to a Brotli stream was not successfully compressed.

ERR_BROTLI_INVALID_PARAM

An invalid parameter key was passed during construction of a Brotli stream.

ERR_BUFFER_CONTEXT_NOT_AVAILABLE

An attempt was made to create a Node.js `Buffer` instance from addon or embedder code, while in a JS engine Context that is not associated with a Node.js instance. The data passed to the `Buffer` method will have been released by the time the method returns.

When encountering this error, a possible alternative to creating a `Buffer` instance is to create a normal `Uint8Array`, which only differs in the prototype of the resulting object. `Uint8Array`s are generally accepted in all Node.js core APIs where `Buffer`s are; they are available in all Contexts.

ERR_BUFFER_OUT_OF_BOUNDS

An operation outside the bounds of a `Buffer` was attempted.

ERR_BUFFER_TOO_LARGE

An attempt has been made to create a `Buffer` larger than the maximum allowed size.

ERR_CANNOT_WATCH_SIGINT

Node.js was unable to watch for the `SIGINT` signal.

ERR_CHILD_CLOSED_BEFORE_REPLY

A child process was closed before the parent received a reply.

ERR_CHILD_PROCESS_IPC_REQUIRED

Used when a child process is being forked without specifying an IPC channel.

ERR_CHILD_PROCESS_STDIO_MAXBUFFER

Used when the main process is trying to read data from the child process's `STDERR/STDOUT`, and the data's length is longer than the `maxBuffer` option.

ERR_CLOSED_MESSAGE_PORT

There was an attempt to use a `MessagePort` instance in a closed state, usually after `.close()` has been called.

ERR_CONSOLE_WRITABLE_STREAM

`Console` was instantiated without `stdout` stream, or `Console` has a non-writable `stdout` or `stderr` stream.

ERR_CONSTRUCT_CALL_INVALID

A class constructor was called that is not callable.

ERR_CONSTRUCT_CALL_REQUIRED

A constructor for a class was called without `new`.

ERR_CONTEXT_NOT_INITIALIZED

The vm context passed into the API is not yet initialized. This could happen when an error occurs (and is caught) during the creation of the context, for example, when the allocation fails or the maximum call stack size is reached when the context is created.

ERR_CRYPTOCUSTOM_ENGINE_NOT_SUPPORTED

A client certificate engine was requested that is not supported by the version of OpenSSL being used.

ERR_CRYPTO_ECDH_INVALID_FORMAT

An invalid value for the `format` argument was passed to the `crypto.ECDH()` class `getPublicKey()` method.

ERR_CRYPTO_ECDH_INVALID_PUBLIC_KEY

An invalid value for the `key` argument has been passed to the `crypto.ECDH()` class `computeSecret()` method. It means that the public key lies outside of the elliptic curve.

ERR_CRYPTO_ENGINE_UNKNOWN

An invalid crypto engine identifier was passed to `require('crypto').setEngine()`.

ERR_CRYPTO_FIPS_FORCED

The `--force-fips` command-line argument was used but there was an attempt to enable or disable FIPS mode in the `crypto` module.

ERR_CRYPTO_FIPS_UNAVAILABLE

An attempt was made to enable or disable FIPS mode, but FIPS mode was not available.

ERR_CRYPTO_HASH_FINALIZED

`hash.digest()` was called multiple times. The `hash.digest()` method must be called no more than one time per instance of a `Hash` object.

ERR_CRYPTO_HASH_UPDATE_FAILED

`hash.update()` failed for any reason. This should rarely, if ever, happen.

ERR_CRYPTO_INCOMPATIBLE_KEY

The given crypto keys are incompatible with the attempted operation.

ERR_CRYPTO_INCOMPATIBLE_KEY_OPTIONS

The selected public or private key encoding is incompatible with other options.

ERR_CRYPTO_INITIALIZATION_FAILED

Initialization of the crypto subsystem failed.

ERR_CRYPTO_INVALID_AUTH_TAG

An invalid authentication tag was provided.

ERR_CRYPTO_INVALID_COUNTER

An invalid counter was provided for a counter-mode cipher.

ERR_CRYPTO_INVALID_CURVE

An invalid elliptic-curve was provided.

ERR_CRYPTO_INVALID_DIGEST

An invalid `crypto digest algorithm` was specified.

ERR_CRYPTO_INVALID_IV

An invalid initialization vector was provided.

ERR_CRYPTO_INVALID_JWK

An invalid JSON Web Key was provided.

ERR_CRYPTO_INVALID_KEY_OBJECT_TYPE

The given crypto key object's type is invalid for the attempted operation.

ERR_CRYPTO_INVALID_KEYLEN

An invalid key length was provided.

ERR_CRYPTO_INVALID_KEYPAIR

An invalid key pair was provided.

ERR_CRYPTO_INVALID_KEYTYPE

An invalid key type was provided.

ERR_CRYPTO_INVALID_MESSAGELEN

An invalid message length was provided.

ERR_CRYPTO_INVALID_SCRIPT_PARAMS

Invalid script algorithm parameters were provided.

ERR_CRYPTO_INVALID_STATE

A crypto method was used on an object that was in an invalid state. For instance, calling `cipher.getAuthTag()` before calling `cipher.final()`.

ERR_CRYPTO_INVALID_TAG_LENGTH

An invalid authentication tag length was provided.

ERR_CRYPTO_JOB_INIT_FAILED

Initialization of an asynchronous crypto operation failed.

ERR_CRYPTO_JWK_UNSUPPORTED_CURVE

Key's Elliptic Curve is not registered for use in the [JSON Web Key Elliptic Curve Registry](#).

ERR_CRYPTO_JWK_UNSUPPORTED_KEY_TYPE

Key's Asymmetric Key Type is not registered for use in the [JSON Web Key Types Registry](#).

ERR_CRYPTO_OPERATION_FAILED

A crypto operation failed for an otherwise unspecified reason.

ERR_CRYPTO_PBKDF2_ERROR

The PBKDF2 algorithm failed for unspecified reasons. OpenSSL does not provide more details and therefore neither does Node.js.

ERR_CRYPTO_SCRIPT_INVALID_PARAMETER

One or more `crypto.script()` or `crypto.scriptSync()` parameters are outside their legal range.

ERR_CRYPTO_SCRIPT_NOT_SUPPORTED

Node.js was compiled without `script` support. Not possible with the official release binaries but can happen with custom builds, including distro builds.

ERR_CRYPTO_SIGN_KEY_REQUIRED

A signing `key` was not provided to the `sign.sign()` method.

ERR_CRYPTO_TIMING_SAFE_EQUAL_LENGTH

`crypto.timingSafeEqual()` was called with `Buffer`, `TypedArray`, or `DataView` arguments of different lengths.

ERR_CRYPTO_UNKNOWN_CIPHER

An unknown cipher was specified.

ERR_CRYPTO_UNKNOWN_DH_GROUP

An unknown Diffie-Hellman group name was given. See `crypto.getDiffieHellman()` for a list of valid group names.

ERR_CRYPTO_UNSUPPORTED_OPERATION

An attempt to invoke an unsupported crypto operation was made.

ERR_DEBUGGER_ERROR

An error occurred with the `debugger`.

ERR_DEBUGGER_STARTUP_ERROR

The `debugger` timed out waiting for the required host/port to be free.

ERR_DLOPEN_FAILED

A call to `process.dlopen()` failed.

ERR_DIR_CLOSED

The `fs.Dir` was previously closed.

ERR_DIR_CONCURRENT_OPERATION

A synchronous read or close call was attempted on an `fs.Dir` which has ongoing asynchronous operations.

ERR_DNS_SET_SERVERS_FAILED

`c-ares` failed to set the DNS server.

ERR_DOMAIN_CALLBACK_NOT_AVAILABLE

The `domain` module was not usable since it could not establish the required error handling hooks, because `process.setUncaughtExceptionCaptureCallback()` had been called at an earlier point in time.

ERR_DOMAIN_CANNOT_SET_UNCAUGHT_EXCEPTION_CAPTURE

`process.setUncaughtExceptionCaptureCallback()` could not be called because the `domain` module has been loaded at an earlier point in time.

The stack trace is extended to include the point in time at which the `domain` module had been loaded.

ERR_ENCODING_INVALID_ENCODED_DATA

Data provided to `TextDecoder()` API was invalid according to the encoding provided.

ERR_ENCODING_NOT_SUPPORTED

Encoding provided to `TextDecoder()` API was not one of the [WHATWG Supported Encodings](#).

ERR_EVAL_ESM_CANNOT_PRINT

`--print` cannot be used with ESM input.

ERR_EVENT_RECURSION

Thrown when an attempt is made to recursively dispatch an event on `EventTarget`.

ERR_EXECUTION_ENVIRONMENT_NOT_AVAILABLE

The JS execution context is not associated with a Node.js environment. This may occur when Node.js is used as an embedded library and some hooks for the JS engine are not set up properly.

ERR_FALSY_VALUE_REJECTION

A `Promise` that was callbackified via `util.callbackify()` was rejected with a falsy value.

ERR_FEATURE_UNAVAILABLE_ON_PLATFORM

Used when a feature that is not available to the current platform which is running Node.js is used.

ERR_FS_CP_DIR_TO_NON_DIR

An attempt was made to copy a directory to a non-directory (file, symlink, etc.) using `fs.cp()`.

ERR_FS_CP_EEXIST

An attempt was made to copy over a file that already existed with `fs.cp()`, with the `force` and `errorOnExist` set to `true`.

ERR_FS_CP_EINVAL

When using `fs.cp()`, `src` or `dest` pointed to an invalid path.

ERR_FS_CP_FIFO_PIPE

An attempt was made to copy a named pipe with `fs.cp()`.

ERR_FS_CP_NON_DIR_TO_DIR

An attempt was made to copy a non-directory (file, symlink, etc.) to a directory using `fs.cp()`.

ERR_FS_CP_SOCKET

An attempt was made to copy to a socket with `fs.cp()`.

ERR_FS_CP_SYMLINK_TO_SUBDIRECTORY

When using `fs.cp()`, a symlink in `dest` pointed to a subdirectory of `src`.

ERR_FS_CP_UNKNOWN

An attempt was made to copy to an unknown file type with `fs.cp()`.

ERR_FS_EISDIR

Path is a directory.

ERR_FS_FILE_TOO_LARGE

An attempt has been made to read a file whose size is larger than the maximum allowed size for a `Buffer`.

ERR_FS_INVALID_SYMLINK_TYPE

An invalid symlink type was passed to the `fs.symlink()` or `fs.symlinkSync()` methods.

ERR_HTTP_HEADERS_SENT

An attempt was made to add more headers after the headers had already been sent.

ERR_HTTP_INVALID_HEADER_VALUE

An invalid HTTP header value was specified.

ERR_HTTP_INVALID_STATUS_CODE

Status code was outside the regular status code range (100-999).

ERR_HTTP_REQUEST_TIMEOUT

The client has not sent the entire request within the allowed time.

ERR_HTTP_SOCKET_ENCODING

Changing the socket encoding is not allowed per [RFC 7230 Section 3](#).

ERR_HTTP_TRAILER_INVALID

The `Trailer` header was set even though the transfer encoding does not support that.

ERR_HTTP2_ALTSVC_INVALID_ORIGIN

HTTP/2 ALTSVC frames require a valid origin.

ERR_HTTP2_ALTSVC_LENGTH

HTTP/2 ALTSVC frames are limited to a maximum of 16,382 payload bytes.

ERR_HTTP2_CONNECT_AUTHORITY

For HTTP/2 requests using the `CONNECT` method, the `:authority` pseudo-header is required.

ERR_HTTP2_CONNECT_PATH

For HTTP/2 requests using the `CONNECT` method, the `:path` pseudo-header is forbidden.

ERR_HTTP2_CONNECT_SCHEME

For HTTP/2 requests using the `CONNECT` method, the `:scheme` pseudo-header is forbidden.

ERR_HTTP2_ERROR

A non-specific HTTP/2 error has occurred.

ERR_HTTP2_GOAWAY_SESSION

New HTTP/2 Streams may not be opened after the `Http2Session` has received a `GOAWAY` frame from the connected peer.

ERR_HTTP2_HEADER_SINGLE_VALUE

Multiple values were provided for an HTTP/2 header field that was required to have only a single value.

ERR_HTTP2_HEADERS_AFTER_RESPOND

An additional headers was specified after an HTTP/2 response was initiated.

ERR_HTTP2_HEADERS_SENT

An attempt was made to send multiple response headers.

ERR_HTTP2_INFO_STATUS_NOT_ALLOWED

Informational HTTP status codes (`1xx`) may not be set as the response status code on HTTP/2 responses.

ERR_HTTP2_INVALID_CONNECTION_HEADERS

HTTP/1 connection specific headers are forbidden to be used in HTTP/2 requests and responses.

ERR_HTTP2_INVALID_HEADER_VALUE

An invalid HTTP/2 header value was specified.

ERR_HTTP2_INVALID_INFO_STATUS

An invalid HTTP informational status code has been specified. Informational status codes must be an integer between `100` and `199` (inclusive).

ERR_HTTP2_INVALID_ORIGIN

HTTP/2 `ORIGIN` frames require a valid origin.

ERR_HTTP2_INVALID_PACKED_SETTINGS_LENGTH

Input `Buffer` and `Uint8Array` instances passed to the `http2.getUnpackedSettings()` API must have a length that is a multiple of six.

ERR_HTTP2_INVALID_PSEUDOHEADER

Only valid HTTP/2 pseudoheaders (`:status` , `:path` , `:authority` , `:scheme` , and `:method`) may be used.

ERR_HTTP2_INVALID_SESSION

An action was performed on an `Http2Session` object that had already been destroyed.

ERR_HTTP2_INVALID_SETTING_VALUE

An invalid value has been specified for an HTTP/2 setting.

ERR_HTTP2_INVALID_STREAM

An operation was performed on a stream that had already been destroyed.

ERR_HTTP2_MAX_PENDING_SETTINGS_ACK

Whenever an HTTP/2 `SETTINGS` frame is sent to a connected peer, the peer is required to send an acknowledgment that it has received and applied the new `SETTINGS`. By default, a maximum number of unacknowledged `SETTINGS` frames may be sent at any given time. This error code is used when that limit has been reached.

ERR_HTTP2_NESTED_PUSH

An attempt was made to initiate a new push stream from within a push stream. Nested push streams are not permitted.

ERR_HTTP2_NO_MEM

Out of memory when using the `http2session.setLocalWindowSize(windowSize)` API.

ERR_HTTP2_NO_SOCKET_MANIPULATION

An attempt was made to directly manipulate (read, write, pause, resume, etc.) a socket attached to an `Http2Session`.

ERR_HTTP2_ORIGIN_LENGTH

HTTP/2 `ORIGIN` frames are limited to a length of 16382 bytes.

ERR_HTTP2_OUT_OF_STREAMS

The number of streams created on a single HTTP/2 session reached the maximum limit.

ERR_HTTP2_PAYLOAD_FORBIDDEN

A message payload was specified for an HTTP response code for which a payload is forbidden.

ERR_HTTP2_PING_CANCEL

An HTTP/2 ping was canceled.

ERR_HTTP2_PING_LENGTH

HTTP/2 ping payloads must be exactly 8 bytes in length.

ERR_HTTP2_PSEUDOHEADER_NOT_ALLOWED

An HTTP/2 pseudo-header has been used inappropriately. Pseudo-headers are header key names that begin with the `:` prefix.

ERR_HTTP2_PUSH_DISABLED

An attempt was made to create a push stream, which had been disabled by the client.

ERR_HTTP2_SEND_FILE

An attempt was made to use the `Http2Stream.prototype.responseWithFile()` API to send a directory.

ERR_HTTP2_SEND_FILE_NOSEEK

An attempt was made to use the `Http2Stream.prototype.responseWithFile()` API to send something other than a regular file, but `offset` or `length` options were provided.

ERR_HTTP2_SESSION_ERROR

The `Http2Session` closed with a non-zero error code.

ERR_HTTP2_SETTINGS_CANCEL

The `Http2Session` settings canceled.

ERR_HTTP2_SOCKET_BOUND

An attempt was made to connect a `Http2Session` object to a `net.Socket` or `tls.TLSSocket` that had already been bound to another `Http2Session` object.

ERR_HTTP2_SOCKET_UNBOUND

An attempt was made to use the `socket` property of an `Http2Session` that has already been closed.

ERR_HTTP2_STATUS_101

Use of the `101` Informational status code is forbidden in HTTP/2.

ERR_HTTP2_STATUS_INVALID

An invalid HTTP status code has been specified. Status codes must be an integer between `100` and `599` (inclusive).

ERR_HTTP2_STREAM_CANCEL

An `Http2Stream` was destroyed before any data was transmitted to the connected peer.

ERR_HTTP2_STREAM_ERROR

A non-zero error code was been specified in an `RST_STREAM` frame.

ERR_HTTP2_STREAM_SELF_DEPENDENCY

When setting the priority for an HTTP/2 stream, the stream may be marked as a dependency for a parent stream. This error code is used when an attempt is made to mark a stream and dependent of itself.

ERR_HTTP2_TOO_MANY_INVALID_FRAMES

The limit of acceptable invalid HTTP/2 protocol frames sent by the peer, as specified through the `maxSessionInvalidFrames` option, has been exceeded.

ERR_HTTP2_TRAILERS_ALREADY_SENT

Trailing headers have already been sent on the `Http2Stream`.

ERR_HTTP2_TRAILERS_NOT_READY

The `http2stream.sendTrailers()` method cannot be called until after the `'wantTrailers'` event is emitted on an `Http2Stream` object. The `'wantTrailers'` event will only be emitted if the `waitForTrailers` option is set for the `Http2Stream`.

ERR_HTTP2_UNSUPPORTED_PROTOCOL

`http2.connect()` was passed a URL that uses any protocol other than `http:` or `https:`.

ERR_ILLEGAL_CONSTRUCTOR

An attempt was made to construct an object using a non-public constructor.

ERR_INCOMPATIBLE_OPTION_PAIR

An option pair is incompatible with each other and cannot be used at the same time.

ERR_INPUT_TYPE_NOT_ALLOWED

Stability: 1 - Experimental

The `--input-type` flag was used to attempt to execute a file. This flag can only be used with input via `--eval`, `--print` or `STDIN`.

ERR_INSPECTOR_ALREADY_ACTIVATED

While using the `inspector` module, an attempt was made to activate the inspector when it already started to listen on a port. Use `inspector.close()` before activating it on a different address.

ERR_INSPECTOR_ALREADY_CONNECTED

While using the `inspector` module, an attempt was made to connect when the inspector was already connected.

ERR_INSPECTOR_CLOSED

While using the `inspector` module, an attempt was made to use the inspector after the session had already closed.

ERR_INSPECTOR_COMMAND

An error occurred while issuing a command via the `inspector` module.

ERR_INSPECTOR_NOT_ACTIVE

The `inspector` is not active when `inspector.waitForDebugger()` is called.

ERR_INSPECTOR_NOT_AVAILABLE

The `inspector` module is not available for use.

ERR_INSPECTOR_NOT_CONNECTED

While using the `inspector` module, an attempt was made to use the inspector before it was connected.

ERR_INSPECTOR_NOT_WORKER

An API was called on the main thread that can only be used from the worker thread.

ERR_INTERNAL_ASSERTION

There was a bug in Node.js or incorrect usage of Node.js internals. To fix the error, open an issue at <https://github.com/nodejs/node/issues>.

ERR_INVALID_ADDRESS_FAMILY

The provided address family is not understood by the Node.js API.

ERR_INVALID_ARG_TYPE

An argument of the wrong type was passed to a Node.js API.

ERR_INVALID_ARG_VALUE

An invalid or unsupported value was passed for a given argument.

ERR_INVALID_ASYNC_ID

An invalid `asyncId` or `triggerAsyncId` was passed using `AsyncHooks`. An id less than -1 should never happen.

ERR_INVALID_BUFFER_SIZE

A swap was performed on a `Buffer` but its size was not compatible with the operation.

ERR_INVALID_CALLBACK

A callback function was required but was not been provided to a Node.js API.

ERR_INVALID_CHAR

Invalid characters were detected in headers.

ERR_INVALID_CURSOR_POS

A cursor on a given stream cannot be moved to a specified row without a specified column.

ERR_INVALID_FD

A file descriptor ('fd') was not valid (e.g. it was a negative value).

ERR_INVALID_FD_TYPE

A file descriptor ('fd') type was not valid.

ERR_INVALID_FILE_URL_HOST

A Node.js API that consumes `file:` URLs (such as certain functions in the `fs` module) encountered a file URL with an incompatible host. This situation can only occur on Unix-like systems where only `localhost` or an empty host is supported.

ERR_INVALID_FILE_URL_PATH

A Node.js API that consumes `file:` URLs (such as certain functions in the `fs` module) encountered a file URL with an incompatible path. The exact semantics for determining whether a path can be used is platform-dependent.

ERR_INVALID_HANDLE_TYPE

An attempt was made to send an unsupported "handle" over an IPC communication channel to a child process. See `subprocess.send()` and `process.send()` for more information.

ERR_INVALID_HTTP_TOKEN

An invalid HTTP token was supplied.

ERR_INVALID_IP_ADDRESS

An IP address is not valid.

ERR_INVALID_MODULE

An attempt was made to load a module that does not exist or was otherwise not valid.

ERR_INVALID_MODULE_SPECIFIER

The imported module string is an invalid URL, package name, or package subpath specifier.

ERR_INVALID_PACKAGE_CONFIG

An invalid `package.json` file failed parsing.

ERR_INVALID_PACKAGE_TARGET

The `package.json` `"exports"` field contains an invalid target mapping value for the attempted module resolution.

ERR_INVALID_PERFORMANCE_MARK

While using the Performance Timing API (`perf_hooks`), a performance mark is invalid.

ERR_INVALID_PROTOCOL

An invalid `options.protocol` was passed to `http.request()`.

ERR_INVALID_REPL_EVAL_CONFIG

Both `breakEvalOnSigint` and `eval` options were set in the `REPL` config, which is not supported.

ERR_INVALID_REPL_INPUT

The input may not be used in the `REPL`. The conditions under which this error is used are described in the `REPL` documentation.

ERR_INVALID_RETURN_PROPERTY

Thrown in case a function option does not provide a valid value for one of its returned object properties on execution.

ERR_INVALID_RETURN_PROPERTY_VALUE

Thrown in case a function option does not provide an expected value type for one of its returned object properties on execution.

ERR_INVALID_RETURN_VALUE

Thrown in case a function option does not return an expected value type on execution, such as when a function is expected to return a promise.

ERR_INVALID_STATE

Indicates that an operation cannot be completed due to an invalid state. For instance, an object may have already been destroyed, or may be performing another operation.

ERR_INVALID_SYNC_FORK_INPUT

A `Buffer`, `TypedArray`, `DataView` or `string` was provided as `stdio` input to an asynchronous fork. See the documentation for the `child_process` module for more information.

ERR_INVALID_THIS

A Node.js API function was called with an incompatible `this` value.

```
const urlSearchParams = new URLSearchParams('foo=bar&baz=new');

const buf = Buffer.alloc(1);
urlSearchParams.has.call(buf, 'foo');
// Throws a TypeError with code 'ERR_INVALID_THIS'
```

ERR_INVALID_TRANSFER_OBJECT

An invalid transfer object was passed to `postMessage()`.

ERR_INVALID_TUPLE

An element in the `iterable` provided to the `WHATWG URLSearchParams constructor` did not represent a `[name, value]` tuple – that is, if an element is not iterable, or does not consist of exactly two elements.

ERR_INVALID_URI

An invalid URI was passed.

ERR_INVALID_URL

An invalid URL was passed to the `WHATWG URL constructor` to be parsed. The thrown error object typically has an additional property `'input'` that contains the URL that failed to parse.

ERR_INVALID_URL_SCHEME

An attempt was made to use a URL of an incompatible scheme (protocol) for a specific purpose. It is only used in the `WHATWG URL API` support in the `fs` module (which only accepts URLs with `'file'` scheme), but may be used in other Node.js APIs as well in the future.

ERR_IPC_CHANNEL_CLOSED

An attempt was made to use an IPC communication channel that was already closed.

ERR_IPC_DISCONNECTED

An attempt was made to disconnect an IPC communication channel that was already disconnected. See the documentation for the `child_process` module for more information.

ERR_IPC_ONE_PIPE

An attempt was made to create a child Node.js process using more than one IPC communication channel. See the documentation for the `child_process` module for more information.

ERR_IPC_SYNC_FORK

An attempt was made to open an IPC communication channel with a synchronously forked Node.js process. See the documentation for the `child_process` module for more information.

ERR_MANIFEST_ASSERT_INTEGRITY

An attempt was made to load a resource, but the resource did not match the integrity defined by the policy manifest. See the documentation for `policy` manifests for more information.

ERR_MANIFEST_DEPENDENCY_MISSING

An attempt was made to load a resource, but the resource was not listed as a dependency from the location that attempted to load it. See the documentation for `policy` manifests for more information.

ERR_MANIFEST_INTEGRITY_MISMATCH

An attempt was made to load a policy manifest, but the manifest had multiple entries for a resource which did not match each other. Update the manifest entries to match in order to resolve this error. See the documentation for `policy` manifests for more information.

ERR_MANIFEST_INVALID_RESOURCE_FIELD

A policy manifest resource had an invalid value for one of its fields. Update the manifest entry to match in order to resolve this error. See the documentation for `policy` manifests for more information.

ERR_MANIFEST_PARSE_POLICY

An attempt was made to load a policy manifest, but the manifest was unable to be parsed. See the documentation for `policy` manifests for more information.

ERR_MANIFEST_TDZ

An attempt was made to read from a policy manifest, but the manifest initialization has not yet taken place. This is likely a bug in Node.js.

ERR_MANIFEST_UNKNOWN_ONERROR

A policy manifest was loaded, but had an unknown value for its "onerror" behavior. See the documentation for `policy` manifests for more information.

ERR_MEMORY_ALLOCATION_FAILED

An attempt was made to allocate memory (usually in the C++ layer) but it failed.

ERR_MESSAGE_TARGET_CONTEXT_UNAVAILABLE

A message posted to a `MessagePort` could not be deserialized in the target `vm Context`. Not all Node.js objects can be successfully instantiated in any context at this time, and attempting to transfer them using `postMessage()` can fail on the receiving side in that case.

ERR_METHOD_NOT_IMPLEMENTED

A method is required but not implemented.

ERR_MISSING_ARGS

A required argument of a Node.js API was not passed. This is only used for strict compliance with the API specification (which in some cases may accept `func(undefined)` but not `func()`). In most native Node.js APIs, `func(undefined)` and `func()` are treated identically, and the `ERR_INVALID_ARG_TYPE` error code may be used instead.

ERR_MISSING_OPTION

For APIs that accept options objects, some options might be mandatory. This code is thrown if a required option is missing.

ERR_MISSING_PASSPHRASE

An attempt was made to read an encrypted key without specifying a passphrase.

ERR_MISSING_PLATFORM_FOR_WORKER

The V8 platform used by this instance of Node.js does not support creating Workers. This is caused by lack of embedder support for Workers. In particular, this error will not occur with standard builds of Node.js.

ERR_MISSING_TRANSFERABLE_IN_TRANSFER_LIST

An object that needs to be explicitly listed in the `transferList` argument is in the object passed to a `postMessage()` call, but is not provided in the `transferList` for that call. Usually, this is a `MessagePort`.

In Node.js versions prior to v15.0.0, the error code being used here was `ERR_MISSING_MESSAGE_PORT_IN_TRANSFER_LIST`. However, the set of transferable object types has been expanded to cover more types than `MessagePort`.

ERR_MODULE_NOT_FOUND

Stability: 1 - Experimental

An `ES Module` could not be resolved.

ERR_MULTIPLE_CALLBACK

A callback was called more than once.

A callback is almost always meant to only be called once as the query can either be fulfilled or rejected but not both at the same time. The latter would be possible by calling a callback more than once.

ERR_NAPI_CONS_FUNCTION

While using `Node-API`, a constructor passed was not a function.

ERR_NAPI_INVALID_DATAVIEW_ARGS

While calling `napi_create_dataview()`, a given `offset` was outside the bounds of the dataview or `offset + length` was larger than a length of given `buffer`.

ERR_NAPI_INVALID_TYPEDARRAY_ALIGNMENT

While calling `napi_create_typedarray()`, the provided `offset` was not a multiple of the element size.

ERR_NAPI_INVALID_TYPEDARRAY_LENGTH

While calling `napi_create_typedarray()`, `(length * size_of_element) + byte_offset` was larger than the length of given `buffer`.

ERR_NAPI_TSFN_CALL_JS

An error occurred while invoking the JavaScript portion of the thread-safe function.

ERR_NAPI_TSFN_GET_UNDEFINED

An error occurred while attempting to retrieve the JavaScript `undefined` value.

ERR_NAPI_TSFN_START_IDLE_LOOP

On the main thread, values are removed from the queue associated with the thread-safe function in an idle loop. This error indicates that an error has occurred when attempting to start the loop.

ERR_NAPI_TSFN_STOP_IDLE_LOOP

Once no more items are left in the queue, the idle loop must be suspended. This error indicates that the idle loop has failed to stop.

ERR_NO_CRYPTO

An attempt was made to use crypto features while Node.js was not compiled with OpenSSL crypto support.

ERR_NO_ICU

An attempt was made to use features that require `ICU`, but Node.js was not compiled with ICU support.

ERR_NON_CONTEXT_AWARE_DISABLED

A non-context-aware native addon was loaded in a process that disallows them.

ERR_OUT_OF_RANGE

A given value is out of the accepted range.

ERR_PACKAGE_IMPORT_NOT_DEFINED

The `package.json` `"imports"` field does not define the given internal package specifier mapping.

ERR_PACKAGE_PATH_NOT_EXPORTED

The `package.json` `"exports"` field does not export the requested subpath. Because exports are encapsulated, private internal modules that are not exported cannot be imported through the package resolution, unless using an absolute URL.

ERR_PERFORMANCE_INVALID_TIMESTAMP

An invalid timestamp value was provided for a performance mark or measure.

ERR_PERFORMANCE_MEASURE_INVALID_OPTIONS

Invalid options were provided for a performance measure.

ERR_PROTO_ACCESS

Accessing `Object.prototype.__proto__` has been forbidden using `--disable-proto=throw`. `Object.getPrototypeOf` and `Object.setPrototypeOf` should be used to get and set the prototype of an object.

ERR_REQUIRE_ESM

Stability: 1 - Experimental

An attempt was made to `require()` an `ES Module`.

ERR_SCRIPT_EXECUTION_INTERRUPTED

Script execution was interrupted by `SIGINT` (For example, `ctrl + c` was pressed.)

ERR_SCRIPT_EXECUTION_TIMEOUT

Script execution timed out, possibly due to bugs in the script being executed.

ERR_SERVER_ALREADY_LISTEN

The `server.listen()` method was called while a `net.Server` was already listening. This applies to all instances of `net.Server`, including HTTP, HTTPS, and HTTP/2 `Server` instances.

ERR_SERVER_NOT_RUNNING

The `server.close()` method was called when a `net.Server` was not running. This applies to all instances of `net.Server`, including HTTP, HTTPS, and HTTP/2 `Server` instances.

ERR_SOCKET_ALREADY_BOUND

An attempt was made to bind a socket that has already been bound.

ERR_SOCKET_BAD_BUFFER_SIZE

An invalid (negative) size was passed for either the `recvBufferSize` or `sendBufferSize` options in `dgram.createSocket()`.

ERR_SOCKET_BAD_PORT

An API function expecting a port ≥ 0 and < 65536 received an invalid value.

ERR_SOCKET_BAD_TYPE

An API function expecting a socket type (`udp4` or `udp6`) received an invalid value.

ERR_SOCKET_BUFFER_SIZE

While using `dgram.createSocket()`, the size of the receive or send `Buffer` could not be determined.

ERR_SOCKET_CLOSED

An attempt was made to operate on an already closed socket.

ERR_SOCKET_DGRAM_IS_CONNECTED

A `dgram.connect()` call was made on an already connected socket.

ERR_SOCKET_DGRAM_NOT_CONNECTED

A `dgram.disconnect()` or `dgram.remoteAddress()` call was made on a disconnected socket.

ERR_SOCKET_DGRAM_NOT_RUNNING

A call was made and the UDP subsystem was not running.

ERR_SRI_PARSE

A string was provided for a Subresource Integrity check, but was unable to be parsed. Check the format of integrity attributes by looking at the [Subresource Integrity specification](#).

ERR_STREAM_ALREADY_FINISHED

A stream method was called that cannot complete because the stream was finished.

ERR_STREAM_CANNOT_PIPE

An attempt was made to call `stream.pipe()` on a `Writable` stream.

ERR_STREAM_DESTROYED

A stream method was called that cannot complete because the stream was destroyed using `stream.destroy()`.

ERR_STREAM_NULL_VALUES

An attempt was made to call `stream.write()` with a `null` chunk.

ERR_STREAM_PREMATURE_CLOSE

An error returned by `stream.finished()` and `stream.pipeline()`, when a stream or a pipeline ends non gracefully with no explicit error.

ERR_STREAM_PUSH_AFTER_EOF

An attempt was made to call `stream.push()` after a `null` (EOF) had been pushed to the stream.

ERR_STREAM_UNSHIFT_AFTER_END_EVENT

An attempt was made to call `stream.unshift()` after the `'end'` event was emitted.

ERR_STREAM_WRAP

Prevents an abort if a string decoder was set on the Socket or if the decoder is in `objectMode`.

```
const Socket = require('net').Socket;
const instance = new Socket();

instance.setEncoding('utf8');
```

ERR_STREAM_WRITE_AFTER_END

An attempt was made to call `stream.write()` after `stream.end()` has been called.

ERR_STRING_TOO_LONG

An attempt has been made to create a string longer than the maximum allowed length.

ERR_SYNTHETIC

An artificial error object used to capture the call stack for diagnostic reports.

ERR_SYSTEM_ERROR

An unspecified or non-specific system error has occurred within the Node.js process. The error object will have an `err.info` object property with additional details.

ERR_TLS_CERT_ALTNAME_INVALID

While using TLS, the host name/IP of the peer did not match any of the `subjectAltNames` in its certificate.

ERR_TLS_DH_PARAM_SIZE

While using TLS, the parameter offered for the Diffie-Hellman (`DH`) key-agreement protocol is too small. By default, the key length must be greater than or equal to 1024 bits to avoid vulnerabilities, even though it is strongly recommended to use 2048 bits or larger for stronger security.

ERR_TLS_HANDSHAKE_TIMEOUT

A TLS/SSL handshake timed out. In this case, the server must also abort the connection.

ERR_TLS_INVALID_CONTEXT

The context must be a `SecureContext` .

ERR_TLS_INVALID_PROTOCOL_METHOD

The specified `secureProtocol` method is invalid. It is either unknown, or disabled because it is insecure.

ERR_TLS_INVALID_PROTOCOL_VERSION

Valid TLS protocol versions are `'TLSv1'` , `'TLSv1.1'` , or `'TLSv1.2'` .

ERR_TLS_INVALID_STATE

The TLS socket must be connected and securely established. Ensure the 'secure' event is emitted before continuing.

ERR_TLS_PROTOCOL_VERSION_CONFLICT

Attempting to set a TLS protocol `minVersion` or `maxVersion` conflicts with an attempt to set the `secureProtocol` explicitly. Use one mechanism or the other.

ERR_TLS_PSK_SET_IDENTITY_HINT_FAILED

Failed to set PSK identity hint. Hint may be too long.

ERR_TLS_RENEGOTIATION_DISABLED

An attempt was made to renegotiate TLS on a socket instance with TLS disabled.

ERR_TLS_REQUIRED_SERVER_NAME

While using TLS, the `server.addContext()` method was called without providing a host name in the first parameter.

ERR_TLS_SESSION_ATTACK

An excessive amount of TLS renegotiations is detected, which is a potential vector for denial-of-service attacks.

ERR_TLS_SNI_FROM_SERVER

An attempt was made to issue Server Name Indication from a TLS server-side socket, which is only valid from a client.

ERR_TRACE_EVENTS_CATEGORY_REQUIRED

The `trace_events.createTracing()` method requires at least one trace event category.

ERR_TRACE_EVENTS_UNAVAILABLE

The `trace_events` module could not be loaded because Node.js was compiled with the `--without-v8-platform` flag.

ERR_TRANSFORM_ALREADY_TRANSFORMING

A `Transform` stream finished while it was still transforming.

ERR_TRANSFORM_WITH_LENGTH_0

A `Transform` stream finished with data still in the write buffer.

ERR_TTY_INIT_FAILED

The initialization of a TTY failed due to a system error.

ERR_UNAVAILABLE_DURING_EXIT

Function was called within a `process.on('exit')` handler that shouldn't be called within `process.on('exit')` handler.

ERR_UNCAUGHT_EXCEPTION_CAPTURE_ALREADY_SET

`process.setUncaughtExceptionCaptureCallback()` was called twice, without first resetting the callback to `null`.

This error is designed to prevent accidentally overwriting a callback registered from another module.

ERR_UNESCAPED_CHARACTERS

A string that contained unescaped characters was received.

ERR_UNHANDLED_ERROR

An unhandled error occurred (for instance, when an `'error'` event is emitted by an `EventEmitter` but an `'error'` handler is not registered).

ERR_UNKNOWN_BUILTIN_MODULE

Used to identify a specific kind of internal Node.js error that should not typically be triggered by user code. Instances of this error point to an internal bug within the Node.js binary itself.

ERR_UNKNOWN_CREDENTIAL

A Unix group or user identifier that does not exist was passed.

ERR_UNKNOWN_ENCODING

An invalid or unknown encoding option was passed to an API.

ERR_UNKNOWN_FILE_EXTENSION

Stability: 1 - Experimental

An attempt was made to load a module with an unknown or unsupported file extension.

ERR_UNKNOWN_MODULE_FORMAT

An attempt was made to load a module with an unknown or unsupported format.

ERR_UNKNOWN_SIGNAL

An invalid or unknown process signal was passed to an API expecting a valid signal (such as `subprocess.kill()`).

ERR_UNSUPPORTED_DIR_IMPORT

`import` a directory URL is unsupported. Instead, [self-reference a package using its name](#) and [define a custom subpath](#) in the "exports" field of the `package.json` file.

```
import './'; // unsupported
import './index.js'; // supported
import 'package-name'; // supported
```

ERR_UNSUPPORTED_ESM_URL_SCHEME

`import` with URL schemes other than `file` and `data` is unsupported.

ERR_VALID_PERFORMANCE_ENTRY_TYPE

While using the Performance Timing API (`perf_hooks`), no valid performance entry types are found.

ERR_VM_DYNAMIC_IMPORT_CALLBACK_MISSING

A dynamic import callback was not specified.

ERR_VM_MODULE_ALREADY_LINKED

The module attempted to be linked is not eligible for linking, because of one of the following reasons:

- It has already been linked (`linkingStatus` is `'linked'`)
- It is being linked (`linkingStatus` is `'linking'`)
- Linking has failed for this module (`linkingStatus` is `'errored'`)

ERR_VM_MODULE_CACHED_DATA_REJECTED

The `cachedData` option passed to a module constructor is invalid.

ERR_VM_MODULE_CANNOT_CREATE_CACHED_DATA

Cached data cannot be created for modules which have already been evaluated.

ERR_VM_MODULE_DIFFERENT_CONTEXT

The module being returned from the linker function is from a different context than the parent module. Linked modules must share the same context.

ERR_VM_MODULE_LINKING_ERRORED

The linker function returned a module for which linking has failed.

ERR_VM_MODULE_LINK_FAILURE

The module was unable to be linked due to a failure.

ERR_VM_MODULE_NOT_MODULE

The fulfilled value of a linking promise is not a `vm.Module` object.

ERR_VM_MODULE_STATUS

The current module's status does not allow for this operation. The specific meaning of the error depends on the specific function.

ERR_WASI_ALREADY_STARTED

The WASI instance has already started.

ERR_WASI_NOT_STARTED

The WASI instance has not been started.

ERR_WORKER_INIT_FAILED

The `Worker` initialization failed.

ERR_WORKER_INVALID_EXEC_ARGV

The `execArgv` option passed to the `Worker` constructor contains invalid flags.

ERR_WORKER_NOT_RUNNING

An operation failed because the `Worker` instance is not currently running.

ERR_WORKER_OUT_OF_MEMORY

The `Worker` instance terminated because it reached its memory limit.

ERR_WORKER_PATH

The path for the main script of a worker is neither an absolute path nor a relative path starting with `./` or `../`.

ERR_WORKER_UNSERIALIZABLE_ERROR

All attempts at serializing an uncaught exception from a worker thread failed.

ERR_WORKER_UNSUPPORTED_EXTENSION

The pathname used for the main script of a worker has an unknown file extension.

ERR_WORKER_UNSUPPORTED_OPERATION

The requested functionality is not supported in worker threads.

ERR_ZLIB_INITIALIZATION_FAILED

Creation of a `zlib` object failed due to incorrect configuration.

HPE_HEADER_OVERFLOW

Too much HTTP header data was received. In order to protect against malicious or malconfigured clients, if more than 8 KB of HTTP header data is received then HTTP parsing will abort without a request or response object being created, and an `Error` with this code will be emitted.

HPE_UNEXPECTED_CONTENT_LENGTH

Server is sending both a `Content-Length` header and `Transfer-Encoding: chunked`.

`Transfer-Encoding: chunked` allows the server to maintain an HTTP persistent connection for dynamically generated content. In this case, the `Content-Length` HTTP header cannot be used.

Use `Content-Length` or `Transfer-Encoding: chunked`.

MODULE_NOT_FOUND

A module file could not be resolved while attempting a `require()` or `import` operation.

Legacy Node.js error codes

Stability: 0 - Deprecated. These error codes are either inconsistent, or have been removed.

ERR_CANNOT_TRANSFER_OBJECT

The value passed to `postMessage()` contained an object that is not supported for transferring.

ERR_CRYPTO_HASH_DIGEST_NO_UTF16

The UTF-16 encoding was used with `hash.digest()`. While the `hash.digest()` method does allow an `encoding` argument to be passed in, causing the method to return a string rather than a `Buffer`, the UTF-16 encoding (e.g. `ucs` or `utf16le`) is not supported.

ERR_HTTP2_FRAME_ERROR

Used when a failure occurs sending an individual frame on the HTTP/2 session.

ERR_HTTP2_HEADERS_OBJECT

Used when an HTTP/2 Headers Object is expected.

ERR_HTTP2_HEADER_REQUIRED

Used when a required header is missing in an HTTP/2 message.

ERR_HTTP2_INFO_HEADERS_AFTER_RESPOND

HTTP/2 informational headers must only be sent *prior* to calling the `Http2Stream.prototype.respond()` method.

ERR_HTTP2_STREAM_CLOSED

Used when an action has been performed on an HTTP/2 Stream that has already been closed.

ERR_HTTP_INVALID_CHAR

Used when an invalid character is found in an HTTP response status message (reason phrase).

ERR_INDEX_OUT_OF_RANGE

A given index was out of the accepted range (e.g. negative offsets).

ERR_INVALID_OPT_VALUE

An invalid or unexpected value was passed in an options object.

ERR_INVALID_OPT_VALUE_ENCODING

An invalid or unknown file encoding was passed.

ERR_MISSING_MESSAGE_PORT_IN_TRANSFER_LIST

This error code was replaced by `ERR_MISSING_TRANSFERABLE_IN_TRANSFER_LIST` in Node.js v15.0.0, because it is no longer accurate as other types of transferable objects also exist now.

ERR_NAPI_CONS_PROTOTYPE_OBJECT

Used by the `Node-API` when `Constructor.prototype` is not an object.

ERR_NO_LONGER_SUPPORTED

A Node.js API was called in an unsupported manner, such as `Buffer.write(string, encoding, offset[, length])`.

ERR_OPERATION_FAILED

An operation failed. This is typically used to signal the general failure of an asynchronous operation.

ERR_OUTOFMEMORY

Used generically to identify that an operation caused an out of memory condition.

ERR_PARSE_HISTORY_DATA

The `repl` module was unable to parse data from the REPL history file.

ERR_SOCKET_CANNOT_SEND

Data could not be sent on a socket.

ERR_STDERR_CLOSE

An attempt was made to close the `process.stderr` stream. By design, Node.js does not allow `stdout` or `stderr` streams to be closed by user code.

ERR_STDOUT_CLOSE

An attempt was made to close the `process.stdout` stream. By design, Node.js does not allow `stdout` or `stderr` streams to be closed by user code.

ERR_STREAM_READ_NOT_IMPLEMENTED

Used when an attempt is made to use a readable stream that has not implemented `readable._read()`.

ERR_TLS_RENEGOTIATION_FAILED

Used when a TLS renegotiation request has failed in a non-specific way.

ERR_TRANSFERRING_EXTERNALIZED_SHAREDARRAYBUFFER

A `SharedArrayBuffer` whose memory is not managed by the JavaScript engine or by Node.js was encountered during serialization. Such a `SharedArrayBuffer` cannot be serialized.

This can only happen when native addons create `SharedArrayBuffer`s in "externalized" mode, or put existing `SharedArrayBuffer` into externalized mode.

ERR_UNKNOWN_STDIN_TYPE

An attempt was made to launch a Node.js process with an unknown `stdin` file type. This error is usually an indication of a bug within Node.js itself, although it is possible for user code to trigger it.

ERR_UNKNOWN_STREAM_TYPE

An attempt was made to launch a Node.js process with an unknown `stdout` or `stderr` file type. This error is usually an indication of a bug within Node.js itself, although it is possible for user code to trigger it.

ERR_V8BREAKITERATOR

The V8 `BreakIterator` API was used but the full ICU data set is not installed.

ERR_VALUE_OUT_OF_RANGE

Used when a given value is out of the accepted range.

ERR_VM_MODULE_NOT_LINKED

The module must be successfully linked before instantiation.

ERR_ZLIB_BINDING_CLOSED

Used when an attempt is made to use a `zlib` object after it has already been closed.

ERR_CPU_USAGE

The native call from `process.cpuUsage` could not be processed.

Events

Stability: 2 - Stable

Source Code: [lib/events.js](#)

Much of the Node.js core API is built around an idiomatic asynchronous event-driven architecture in which certain kinds of objects (called "emitters") emit named events that cause `Function` objects ("listeners") to be called.

For instance: a `net.Server` object emits an event each time a peer connects to it; a `fs.ReadStream` emits an event when the file is opened; a `stream` emits an event whenever data is available to be read.

All objects that emit events are instances of the `EventEmitter` class. These objects expose an `eventEmitter.on()` function that allows one or more functions to be attached to named events emitted by the object. Typically, event names are camel-cased strings but any valid JavaScript property key can be used.

When the `EventEmitter` object emits an event, all of the functions attached to that specific event are called *synchronously*. Any values returned by the called listeners are *ignored* and discarded.

The following example shows a simple `EventEmitter` instance with a single listener. The `eventEmitter.on()` method is used to register listeners, while the `eventEmitter.emit()` method is used to trigger the event.

```
const EventEmitter = require('events');

class MyEmitter extends EventEmitter {}

const myEmitter = new MyEmitter();
myEmitter.on('event', () => {
  console.log('an event occurred!');
});
myEmitter.emit('event');
```

Passing arguments and `this` to listeners

The `eventEmitter.emit()` method allows an arbitrary set of arguments to be passed to the listener functions. Keep in mind that when an ordinary listener function is called, the standard `this` keyword is intentionally set to reference the `EventEmitter` instance to which the listener is attached.

```
const myEmitter = new MyEmitter();
myEmitter.on('event', function(a, b) {
  console.log(a, b, this, this === myEmitter);
  // Prints:
  //   a b MyEmitter {
  //     domain: null,
  //     _events: { event: [Function] },
  //     _eventsCount: 1,
  //     _maxListeners: undefined } true
});
myEmitter.emit('event', 'a', 'b');
```

It is possible to use ES6 Arrow Functions as listeners, however, when doing so, the `this` keyword will no longer reference the `EventEmitter` instance:

```
const myEmitter = new MyEmitter();
myEmitter.on('event', (a, b) => {
  console.log(a, b, this);
  // Prints: a b {}
});
myEmitter.emit('event', 'a', 'b');
```

Asynchronous vs. synchronous

The `EventEmitter` calls all listeners synchronously in the order in which they were registered. This ensures the proper sequencing of events and helps avoid race conditions and logic errors. When appropriate, listener functions can switch to an asynchronous mode of operation using

the `setImmediate()` or `process.nextTick()` methods:

```
const myEmitter = new MyEmitter();
myEmitter.on('event', (a, b) => {
  setImmediate(() => {
    console.log('this happens asynchronously');
  });
});
myEmitter.emit('event', 'a', 'b');
```

Handling events only once

When a listener is registered using the `eventEmitter.on()` method, that listener is invoked *every time* the named event is emitted.

```
const myEmitter = new MyEmitter();
let m = 0;
myEmitter.on('event', () => {
  console.log(++m);
});
myEmitter.emit('event');
// Prints: 1
myEmitter.emit('event');
// Prints: 2
```

Using the `eventEmitter.once()` method, it is possible to register a listener that is called at most once for a particular event. Once the event is emitted, the listener is unregistered and *then* called.

```
const myEmitter = new MyEmitter();
let m = 0;
myEmitter.once('event', () => {
  console.log(++m);
});
myEmitter.emit('event');
// Prints: 1
myEmitter.emit('event');
// Ignored
```

Error events

When an error occurs within an `EventEmitter` instance, the typical action is for an `'error'` event to be emitted. These are treated as special cases within Node.js.

If an `EventEmitter` does *not* have at least one listener registered for the `'error'` event, and an `'error'` event is emitted, the error is thrown, a stack trace is printed, and the Node.js process exits.

```
const myEmitter = new MyEmitter();
myEmitter.emit('error', new Error('whoops!'));
// Throws and crashes Node.js
```

To guard against crashing the Node.js process the `domain` module can be used. (Note, however, that the `domain` module is deprecated.)

As a best practice, listeners should always be added for the `'error'` events.

```
const myEmitter = new MyEmitter();
myEmitter.on('error', (err) => {
  console.error('whoops! there was an error');
});
myEmitter.emit('error', new Error('whoops!'));
// Prints: whoops! there was an error
```

It is possible to monitor `'error'` events without consuming the emitted error by installing a listener using the symbol `events.errorMonitor`.

```
const { EventEmitter, errorMonitor } = require('events');

const myEmitter = new EventEmitter();
myEmitter.on(errorMonitor, (err) => {
  MyMonitoringTool.log(err);
});
myEmitter.emit('error', new Error('whoops!'));
// Still throws and crashes Node.js
```

Capture rejections of promises

Stability: 1 - `captureRejections` is experimental.

Using `async` functions with event handlers is problematic, because it can lead to an unhandled rejection in case of a thrown exception:

```
const ee = new EventEmitter();
ee.on('something', async (value) => {
  throw new Error('kaboom');
});
```

The `captureRejections` option in the `EventEmitter` constructor or the global setting change this behavior, installing a `.then(undefined, handler)` handler on the `Promise`. This handler routes the exception asynchronously to the `Symbol.for('nodejs.rejection')` method if there is one, or to `'error'` event handler if there is none.

```
const ee1 = new EventEmitter({ captureRejections: true });
ee1.on('something', async (value) => {
  throw new Error('kaboom');
});

ee1.on('error', console.log);

const ee2 = new EventEmitter({ captureRejections: true });
```



```
ee2.on('something', async (value) => {
  throw new Error('kaboom');
});

ee2[Symbol.for('nodejs.rejection')] = console.log;
```

Setting `events.captureRejections = true` will change the default for all new instances of `EventEmitter`.

```
const events = require('events');
events.captureRejections = true;
const ee1 = new events.EventEmitter();
ee1.on('something', async (value) => {
  throw new Error('kaboom');
});

ee1.on('error', console.log);
```

The `'error'` events that are generated by the `captureRejections` behavior do not have a catch handler to avoid infinite error loops: the recommendation is to **not use `async` functions as `'error'` event handlers**.

Class: `EventEmitter`

The `EventEmitter` class is defined and exposed by the `events` module:

```
const EventEmitter = require('events');
```

All `EventEmitter`s emit the event `'newListener'` when new listeners are added and `'removeListener'` when existing listeners are removed.

It supports the following option:

- `captureRejections` `<boolean>` It enables *automatic capturing of promise rejection*. **Default:** `false`.

Event: `'newListener'`

- `eventName` `<string> | <symbol>` The name of the event being listened for
- `listener` `<Function>` The event handler function

The `EventEmitter` instance will emit its own `'newListener'` event *before* a listener is added to its internal array of listeners.

Listeners registered for the `'newListener'` event are passed the event name and a reference to the listener being added.

The fact that the event is triggered before adding the listener has a subtle but important side effect: any *additional* listeners registered to the same `name` *within* the `'newListener'` callback are inserted *before* the listener that is in the process of being added.

```
class MyEmitter extends EventEmitter {}

const myEmitter = new MyEmitter();
// Only do this once so we don't loop forever
myEmitter.once('newListener', (event, listener) => {
  if (event === 'event') {
```

```
// Insert a new listener in front
myEmitter.on('event', () => {
  console.log('B');
});
}
});
myEmitter.on('event', () => {
  console.log('A');
});
myEmitter.emit('event');
// Prints:
//   B
//   A
```

Event: 'removeListener'

- `eventName` `<string> | <symbol>` The event name
- `listener` `<Function>` The event handler function

The 'removeListener' event is emitted *after* the `listener` is removed.

`emitter.addListener(eventName, listener)`

- `eventName` `<string> | <symbol>`
- `listener` `<Function>`

Alias for `emitter.on(eventName, listener)`.

`emitter.emit(eventName[, ...args])`

- `eventName` `<string> | <symbol>`
- `...args` `<any>`
- Returns: `<boolean>`

Synchronously calls each of the listeners registered for the event named `eventName`, in the order they were registered, passing the supplied arguments to each.

Returns `true` if the event had listeners, `false` otherwise.

```
const EventEmitter = require('events');
const myEmitter = new EventEmitter();

// First listener
myEmitter.on('event', function firstListener() {
  console.log('Helloooo! first listener');
});
// Second listener
myEmitter.on('event', function secondListener(arg1, arg2) {
  console.log(`event with parameters ${arg1}, ${arg2} in second listener`);
});
// Third listener
myEmitter.on('event', function thirdListener(...args) {
  const parameters = args.join(', ');
```

```

    console.log(`event with parameters ${parameters} in third listener`);
  });

  console.log(myEmitter.listeners('event'));

  myEmitter.emit('event', 1, 2, 3, 4, 5);

  // Prints:
  // [
  //   [Function: firstListener],
  //   [Function: secondListener],
  //   [Function: thirdListener]
  // ]
  // Helloooo! first listener
  // event with parameters 1, 2 in second listener
  // event with parameters 1, 2, 3, 4, 5 in third listener

```

emitter.eventNames()

- Returns: `<Array>`

Returns an array listing the events for which the emitter has registered listeners. The values in the array are strings or `Symbol`s.

```

const EventEmitter = require('events');
const myEE = new EventEmitter();
myEE.on('foo', () => {});
myEE.on('bar', () => {});

const sym = Symbol('symbol');
myEE.on(sym, () => {});

console.log(myEE.eventNames());
// Prints: [ 'foo', 'bar', Symbol(symbol) ]

```

emitter.getMaxListeners()

- Returns: `<integer>`

Returns the current max listener value for the `EventEmitter` which is either set by `emitter.setMaxListeners(n)` or defaults to `events.defaultMaxListeners`.

emitter.listenerCount(eventName)

- `eventName` `<string>` | `<symbol>` The name of the event being listened for
- Returns: `<integer>`

Returns the number of listeners listening to the event named `eventName`.

emitter.listeners(eventName)

- `eventName` `<string>` | `<symbol>`
- Returns: `<Function[]>`

Returns a copy of the array of listeners for the event named `eventName`.

```
server.on('connection', (stream) => {
  console.log('someone connected!');
});
console.log(util.inspect(server.listeners('connection')));
// Prints: [ [Function] ]
```

`emitter.off(eventName, listener)`

- `eventName` `<string> | <symbol>`
- `listener` `<Function>`
- Returns: `<EventEmitter>`

Alias for `emitter.removeListener()`.

`emitter.on(eventName, listener)`

- `eventName` `<string> | <symbol>` The name of the event.
- `listener` `<Function>` The callback function
- Returns: `<EventEmitter>`

Adds the `listener` function to the end of the listeners array for the event named `eventName`. No checks are made to see if the `listener` has already been added. Multiple calls passing the same combination of `eventName` and `listener` will result in the `listener` being added, and called, multiple times.

```
server.on('connection', (stream) => {
  console.log('someone connected!');
});
```

Returns a reference to the `EventEmitter`, so that calls can be chained.

By default, event listeners are invoked in the order they are added. The `emitter.prependListener()` method can be used as an alternative to add the event listener to the beginning of the listeners array.

```
const myEE = new EventEmitter();
myEE.on('foo', () => console.log('a'));
myEE.prependListener('foo', () => console.log('b'));
myEE.emit('foo');
// Prints:
//   b
//   a
```

`emitter.once(eventName, listener)`

- `eventName` `<string> | <symbol>` The name of the event.
- `listener` `<Function>` The callback function
- Returns: `<EventEmitter>`

Adds a **one-time** `listener` function for the event named `eventName`. The next time `eventName` is triggered, this listener is removed and then invoked.

```
server.once('connection', (stream) => {
  console.log('Ah, we have our first user!');
});
```

Returns a reference to the `EventEmitter`, so that calls can be chained.

By default, event listeners are invoked in the order they are added. The `emitter.prependOnceListener()` method can be used as an alternative to add the event listener to the beginning of the listeners array.

```
const myEE = new EventEmitter();
myEE.once('foo', () => console.log('a'));
myEE.prependOnceListener('foo', () => console.log('b'));
myEE.emit('foo');
// Prints:
//   b
//   a
```

`emitter.prependListener(eventName, listener)`

- `eventName` `<string> | <symbol>` The name of the event.
- `listener` `<Function>` The callback function
- Returns: `<EventEmitter>`

Adds the `listener` function to the *beginning* of the listeners array for the event named `eventName`. No checks are made to see if the `listener` has already been added. Multiple calls passing the same combination of `eventName` and `listener` will result in the `listener` being added, and called, multiple times.

```
server.prependListener('connection', (stream) => {
  console.log('someone connected!');
});
```

Returns a reference to the `EventEmitter`, so that calls can be chained.

`emitter.prependOnceListener(eventName, listener)`

- `eventName` `<string> | <symbol>` The name of the event.
- `listener` `<Function>` The callback function
- Returns: `<EventEmitter>`

Adds a **one-time** `listener` function for the event named `eventName` to the *beginning* of the listeners array. The next time `eventName` is triggered, this listener is removed, and then invoked.

```
server.prependOnceListener('connection', (stream) => {
  console.log('Ah, we have our first user!');
});
```

Returns a reference to the `EventEmitter`, so that calls can be chained.

`emitter.removeAllListeners([eventName])`

- `eventName` `<string> | <symbol>`
- Returns: `<EventEmitter>`

Removes all listeners, or those of the specified `eventName`.

It is bad practice to remove listeners added elsewhere in the code, particularly when the `EventEmitter` instance was created by some other component or module (e.g. sockets or file streams).

Returns a reference to the `EventEmitter`, so that calls can be chained.

`emitter.removeListener(eventName, listener)`

- `eventName` `<string> | <symbol>`
- `listener` `<Function>`
- Returns: `<EventEmitter>`

Removes the specified `listener` from the listener array for the event named `eventName`.

```
const callback = (stream) => {
  console.log('someone connected!');
};
server.on('connection', callback);
// ...
server.removeListener('connection', callback);
```

`removeListener()` will remove, at most, one instance of a listener from the listener array. If any single listener has been added multiple times to the listener array for the specified `eventName`, then `removeListener()` must be called multiple times to remove each instance.

Once an event is emitted, all listeners attached to it at the time of emitting are called in order. This implies that any `removeListener()` or `removeAllListeners()` calls *after* emitting and *before* the last listener finishes execution will not remove them from `emit()` in progress. Subsequent events behave as expected.

```
const myEmitter = new MyEmitter();

const callbackA = () => {
  console.log('A');
  myEmitter.removeListener('event', callbackB);
};

const callbackB = () => {
  console.log('B');
};

myEmitter.on('event', callbackA);

myEmitter.on('event', callbackB);

// callbackA removes listener callbackB but it will still be called.
// Internal listener array at time of emit [callbackA, callbackB]
myEmitter.emit('event');
// Prints:
//   A
```

```
// B

// callbackB is now removed.
// Internal listener array [callbackA]
myEmitter.emit('event');
// Prints:
// A
```

Because listeners are managed using an internal array, calling this will change the position indices of any listener registered *after* the listener being removed. This will not impact the order in which listeners are called, but it means that any copies of the listener array as returned by the `emitter.listeners()` method will need to be recreated.

When a single function has been added as a handler multiple times for a single event (as in the example below), `removeListener()` will remove the most recently added instance. In the example the `once('ping')` listener is removed:

```
const ee = new EventEmitter();

function pong() {
  console.log('pong');
}

ee.on('ping', pong);
ee.once('ping', pong);
ee.removeListener('ping', pong);

ee.emit('ping');
ee.emit('ping');
```

Returns a reference to the `EventEmitter`, so that calls can be chained.

`emitter.setMaxListeners(n)`

- `n` `<integer>`
- Returns: `<EventEmitter>`

By default `EventEmitter` s will print a warning if more than `10` listeners are added for a particular event. This is a useful default that helps finding memory leaks. The `emitter.setMaxListeners()` method allows the limit to be modified for this specific `EventEmitter` instance. The value can be set to `Infinity` (or `0`) to indicate an unlimited number of listeners.

Returns a reference to the `EventEmitter`, so that calls can be chained.

`emitter.rawListeners(eventName)`

- `eventName` `<string> | <symbol>`
- Returns: `<Function[]>`

Returns a copy of the array of listeners for the event named `eventName`, including any wrappers (such as those created by `.once()`).

```
const emitter = new EventEmitter();
emitter.once('log', () => console.log('log once'));

// Returns a new Array with a function `onceWrapper` which has a property
```

```
// `listener` which contains the original listener bound above
const listeners = emitter.rawListeners('log');
const logFnWrapper = listeners[0];

// Logs "log once" to the console and does not unbind the `once` event
logFnWrapper.listener();

// Logs "log once" to the console and removes the listener
logFnWrapper();

emitter.on('log', () => console.log('log persistently'));
// Will return a new Array with a single function bound by `.on()` above
const newListeners = emitter.rawListeners('log');

// Logs "log persistently" twice
newListeners[0]();
emitter.emit('log');
```

emitter[Symbol.for('nodejs.rejection')](err, eventName[, ...args])

Stability: 1 - captureRejections is experimental.

- `err` `Error`
- `eventName` `<string>` | `<symbol>`
- `...args` `<any>`

The `Symbol.for('nodejs.rejection')` method is called in case a promise rejection happens when emitting an event and `captureRejections` is enabled on the emitter. It is possible to use `events.captureRejectionSymbol` in place of `Symbol.for('nodejs.rejection')`.

```
const { EventEmitter, captureRejectionSymbol } = require('events');

class MyClass extends EventEmitter {
  constructor() {
    super({ captureRejections: true });
  }

  [captureRejectionSymbol](err, event, ...args) {
    console.log('rejection happened for', event, 'with', err, ...args);
    this.destroy(err);
  }

  destroy(err) {
    // Tear the resource down here.
  }
}
```

events.defaultMaxListeners

By default, a maximum of `10` listeners can be registered for any single event. This limit can be changed for individual `EventEmitter` instances using the `emitter.setMaxListeners(n)` method. To change the default for *all* `EventEmitter` instances, the `events.defaultMaxListeners` property can be used. If this value is not a positive number, a `RangeError` is thrown.

Take caution when setting the `events.defaultMaxListeners` because the change affects *all* `EventEmitter` instances, including those created before the change is made. However, calling `emitter.setMaxListeners(n)` still has precedence over `events.defaultMaxListeners`.

This is not a hard limit. The `EventEmitter` instance will allow more listeners to be added but will output a trace warning to `stderr` indicating that a "possible EventEmitter memory leak" has been detected. For any single `EventEmitter`, the `emitter.getMaxListeners()` and `emitter.setMaxListeners()` methods can be used to temporarily avoid this warning:

```
emitter.setMaxListeners(emitter.getMaxListeners() + 1);
emitter.once('event', () => {
  // do stuff
  emitter.setMaxListeners(Math.max(emitter.getMaxListeners() - 1, 0));
});
```

The `--trace-warnings` command-line flag can be used to display the stack trace for such warnings.

The emitted warning can be inspected with `process.on('warning')` and will have the additional `emitter`, `type` and `count` properties, referring to the event emitter instance, the event's name and the number of attached listeners, respectively. Its `name` property is set to `'MaxListenersExceededWarning'`.

events.errorMonitor

This symbol shall be used to install a listener for only monitoring `'error'` events. Listeners installed using this symbol are called before the regular `'error'` listeners are called.

Installing a listener using this symbol does not change the behavior once an `'error'` event is emitted, therefore the process will still crash if no regular `'error'` listener is installed.

events.getEventListeners(emitterOrTarget, eventName)

- `emitterOrTarget` `<EventEmitter> | <EventTarget>`
- `eventName` `<string> | <symbol>`
- Returns: `<Function[]>`

Returns a copy of the array of listeners for the event named `eventName`.

For `EventEmitter` s this behaves exactly the same as calling `.listeners` on the emitter.

For `EventTarget` s this is the only way to get the event listeners for the event target. This is useful for debugging and diagnostic purposes.

```
const { getEventListeners, EventEmitter } = require('events');

{
  const ee = new EventEmitter();
  const listener = () => console.log('Events are fun');
  ee.on('foo', listener);
  getEventListeners(ee, 'foo'); // [listener]
```

```

}
{
  const et = new EventTarget();
  const listener = () => console.log('Events are fun');
  et.addEventListener('foo', listener);
  getEventListeners(et, 'foo'); // [listener]
}

```

events.once(emitter, name[, options])

- `emitter` `<EventEmitter>`
- `name` `<string>`
- `options` `<Object>`
 - `signal` `<AbortSignal>` Can be used to cancel waiting for the event.
- Returns: `<Promise>`

Creates a `Promise` that is fulfilled when the `EventEmitter` emits the given event or that is rejected if the `EventEmitter` emits `'error'` while waiting. The `Promise` will resolve with an array of all the arguments emitted to the given event.

This method is intentionally generic and works with the web platform `EventTarget` interface, which has no special `'error'` event semantics and does not listen to the `'error'` event.

```

const { once, EventEmitter } = require('events');

async function run() {
  const ee = new EventEmitter();

  process.nextTick(() => {
    ee.emit('myevent', 42);
  });

  const [value] = await once(ee, 'myevent');
  console.log(value);

  const err = new Error('kaboom');
  process.nextTick(() => {
    ee.emit('error', err);
  });

  try {
    await once(ee, 'myevent');
  } catch (err) {
    console.log('error happened', err);
  }
}

run();

```

The special handling of the `'error'` event is only used when `events.once()` is used to wait for another event. If `events.once()` is used to wait for the `'error'` event itself, then it is treated as any other kind of event without special handling:

```
const { EventEmitter, once } = require('events');

const ee = new EventEmitter();

once(ee, 'error')
  .then([err] => console.log('ok', err.message))
  .catch(err => console.log('error', err.message));

ee.emit('error', new Error('boom'));

// Prints: ok boom
```

An `<AbortSignal>` can be used to cancel waiting for the event:

```
const { EventEmitter, once } = require('events');

const ee = new EventEmitter();
const ac = new AbortController();

async function foo(emitter, event, signal) {
  try {
    await once(emitter, event, { signal });
    console.log('event emitted!');
  } catch (error) {
    if (error.name === 'AbortError') {
      console.error('Waiting for the event was canceled!');
    } else {
      console.error('There was an error', error.message);
    }
  }
}

foo(ee, 'foo', ac.signal);
ac.abort(); // Abort waiting for the event
ee.emit('foo'); // Prints: Waiting for the event was canceled!
```

Awaiting multiple events emitted on `process.nextTick()`

There is an edge case worth noting when using the `events.once()` function to await multiple events emitted on in the same batch of `process.nextTick()` operations, or whenever multiple events are emitted synchronously. Specifically, because the `process.nextTick()` queue is drained before the `Promise` microtask queue, and because `EventEmitter` emits all events synchronously, it is possible for `events.once()` to miss an event.

```
const { EventEmitter, once } = require('events');

const myEE = new EventEmitter();

async function foo() {
  await once(myEE, 'bar');
```

```

console.log('bar');

// This Promise will never resolve because the 'foo' event will
// have already been emitted before the Promise is created.
await once(myEE, 'foo');
console.log('foo');
}

process.nextTick(() => {
  myEE.emit('bar');
  myEE.emit('foo');
});

foo().then(() => console.log('done'));

```

To catch both events, create each of the Promises *before* awaiting either of them, then it becomes possible to use `Promise.all()`, `Promise.race()`, or `Promise.allSettled()`:

```

const { EventEmitter, once } = require('events');

const myEE = new EventEmitter();

async function foo() {
  await Promise.all([once(myEE, 'bar'), once(myEE, 'foo')]);
  console.log('foo', 'bar');
}

process.nextTick(() => {
  myEE.emit('bar');
  myEE.emit('foo');
});

foo().then(() => console.log('done'));

```

events.captureRejections

Stability: 1 - captureRejections is experimental.

Value: `<boolean>`

Change the default `captureRejections` option on all new `EventEmitter` objects.

events.captureRejectionSymbol

Stability: 1 - captureRejections is experimental.

Value: `Symbol.for('nodejs.rejection')`

See how to write a custom [rejection handler](#).

`events.listenerCount(emitter, eventName)`

Stability: 0 - Deprecated: Use `emitter.listenerCount()` instead.

- `emitter` `<EventEmitter>` The emitter to query
- `eventName` `<string>` | `<symbol>` The event name

A class method that returns the number of listeners for the given `eventName` registered on the given `emitter`.

```
const { EventEmitter, listenerCount } = require('events');
const myEmitter = new EventEmitter();
myEmitter.on('event', () => {});
myEmitter.on('event', () => {});
console.log(listenerCount(myEmitter, 'event'));
// Prints: 2
```

`events.on(emitter, eventName[, options])`

- `emitter` `<EventEmitter>`
- `eventName` `<string>` | `<symbol>` The name of the event being listened for
- `options` `<Object>`
 - `signal` `<AbortSignal>` Can be used to cancel awaiting events.
- Returns: `<AsyncIterator>` that iterates `eventName` events emitted by the `emitter`

```
const { on, EventEmitter } = require('events');

(async () => {
  const ee = new EventEmitter();

  // Emit later on
  process.nextTick(() => {
    ee.emit('foo', 'bar');
    ee.emit('foo', 42);
  });

  for await (const event of on(ee, 'foo')) {
    // The execution of this inner block is synchronous and it
    // processes one event at a time (even with await). Do not use
    // if concurrent execution is required.
    console.log(event); // prints ['bar'] [42]
  }
  // Unreachable here
})();
```

Returns an `AsyncIterator` that iterates `eventName` events. It will throw if the `EventEmitter` emits `'error'`. It removes all listeners when exiting the loop. The `value` returned by each iteration is an array composed of the emitted event arguments.

An `<AbortSignal>` can be used to cancel waiting on events:

```
const { on, EventEmitter } = require('events');
const ac = new AbortController();

(async () => {
  const ee = new EventEmitter();

  // Emit later on
  process.nextTick(() => {
    ee.emit('foo', 'bar');
    ee.emit('foo', 42);
  });

  for await (const event of on(ee, 'foo', { signal: ac.signal })) {
    // The execution of this inner block is synchronous and it
    // processes one event at a time (even with await). Do not use
    // if concurrent execution is required.
    console.log(event); // prints ['bar'] [42]
  }
  // Unreachable here
})();

process.nextTick(() => ac.abort());
```

events.setMaxListeners(n[, ...eventTargets])

- `n` `<number>` A non-negative number. The maximum number of listeners per `EventTarget` event.
- `...eventTargets` `<EventTarget[]> | <EventEmitter[]>` Zero or more `<EventTarget>` or `<EventEmitter>` instances. If none are specified, `n` is set as the default max for all newly created `<EventTarget>` and `<EventEmitter>` objects.

```
const {
  setMaxListeners,
  EventEmitter
} = require('events');

const target = new EventTarget();
const emitter = new EventEmitter();

setMaxListeners(5, target, emitter);
```

EventTarget and Event API

The `EventTarget` and `Event` objects are a Node.js-specific implementation of the `EventTarget Web API` that are exposed by some Node.js core APIs.

```
const target = new EventTarget();

target.addEventListener('foo', (event) => {
  console.log('foo event happened!');
});
```

Node.js EventTarget vs. DOM EventTarget

There are two key differences between the Node.js `EventTarget` and the `EventTarget Web API`:

1. Whereas DOM `EventTarget` instances *may* be hierarchical, there is no concept of hierarchy and event propagation in Node.js. That is, an event dispatched to an `EventTarget` does not propagate through a hierarchy of nested target objects that may each have their own set of handlers for the event.
2. In the Node.js `EventTarget`, if an event listener is an async function or returns a `Promise`, and the returned `Promise` rejects, the rejection is automatically captured and handled the same way as a listener that throws synchronously (see [EventTarget error handling](#) for details).

NodeEventTarget vs. EventEmitter

The `NodeEventTarget` object implements a modified subset of the `EventEmitter` API that allows it to closely *emulate* an `EventEmitter` in certain situations. A `NodeEventTarget` is *not* an instance of `EventEmitter` and cannot be used in place of an `EventEmitter` in most cases.

1. Unlike `EventEmitter`, any given `listener` can be registered at most once per event `type`. Attempts to register a `listener` multiple times are ignored.
2. The `NodeEventTarget` does not emulate the full `EventEmitter` API. Specifically the `prependListener()`, `prependOnceListener()`, `rawListeners()`, `setMaxListeners()`, `getMaxListeners()`, and `errorMonitor` APIs are not emulated. The `'newListener'` and `'removeListener'` events will also not be emitted.
3. The `NodeEventTarget` does not implement any special default behavior for events with type `'error'`.
4. The `NodeEventTarget` supports `EventListener` objects as well as functions as handlers for all event types.

Event listener

Event listeners registered for an event `type` may either be JavaScript functions or objects with a `handleEvent` property whose value is a function.

In either case, the handler function is invoked with the `event` argument passed to the `eventTarget.dispatchEvent()` function.

Async functions may be used as event listeners. If an async handler function rejects, the rejection is captured and handled as described in [EventTarget error handling](#).

An error thrown by one handler function does not prevent the other handlers from being invoked.

The return value of a handler function is ignored.

Handlers are always invoked in the order they were added.

Handler functions may mutate the `event` object.

```
function handler1(event) {
  console.log(event.type); // Prints 'foo'
  event.a = 1;
}
```

```

async function handler2(event) {
  console.log(event.type); // Prints 'foo'
  console.log(event.a); // Prints 1
}

const handler3 = {
  handleEvent(event) {
    console.log(event.type); // Prints 'foo'
  }
};

const handler4 = {
  async handleEvent(event) {
    console.log(event.type); // Prints 'foo'
  }
};

const target = new EventTarget();

target.addEventListener('foo', handler1);
target.addEventListener('foo', handler2);
target.addEventListener('foo', handler3);
target.addEventListener('foo', handler4, { once: true });

```

EventTarget error handling

When a registered event listener throws (or returns a Promise that rejects), by default the error is treated as an uncaught exception on `process.nextTick()`. This means uncaught exceptions in `EventTarget` s will terminate the Node.js process by default.

Throwing within an event listener will *not* stop the other registered handlers from being invoked.

The `EventTarget` does not implement any special default handling for `'error'` type events like `EventEmitter`.

Currently errors are first forwarded to the `process.on('error')` event before reaching `process.on('uncaughtException')`. This behavior is deprecated and will change in a future release to align `EventTarget` with other Node.js APIs. Any code relying on the `process.on('error')` event should be aligned with the new behavior.

Class: Event

The `Event` object is an adaptation of the [Event Web API](#). Instances are created internally by Node.js.

event.bubbles

- Type: `<boolean>` Always returns `false`.

This is not used in Node.js and is provided purely for completeness.

event.cancelBubble()

Alias for `event.stopPropagation()`. This is not used in Node.js and is provided purely for completeness.

event.cancelable

- Type: `<boolean>` True if the event was created with the `cancelable` option.

event.composed

- Type: `<boolean>` Always returns `false`.

This is not used in Node.js and is provided purely for completeness.

event.composedPath()

Returns an array containing the current `EventTarget` as the only entry or empty if the event is not being dispatched. This is not used in Node.js and is provided purely for completeness.

event.currentTarget

- Type: `<EventTarget>` The `EventTarget` dispatching the event.

Alias for `event.target`.

event.defaultPrevented

- Type: `<boolean>`

Is `true` if `cancelable` is `true` and `event.preventDefault()` has been called.

event.eventPhase

- Type: `<number>` Returns `0` while an event is not being dispatched, `2` while it is being dispatched.

This is not used in Node.js and is provided purely for completeness.

event.isTrusted

- Type: `<boolean>`

The `<AbortSignal>` "abort" event is emitted with `isTrusted` set to `true`. The value is `false` in all other cases.

event.preventDefault()

Sets the `defaultPrevented` property to `true` if `cancelable` is `true`.

event.returnValue

- Type: `<boolean>` True if the event has not been canceled.

This is not used in Node.js and is provided purely for completeness.

event.srcElement

- Type: `<EventTarget>` The `EventTarget` dispatching the event.

Alias for `event.target`.

event.stopImmediatePropagation()

Stops the invocation of event listeners after the current one completes.

event.stopPropagation()

This is not used in Node.js and is provided purely for completeness.

event.target

- Type: `<EventTarget>` The `EventTarget` dispatching the event.

event.timeStamp

- Type: `<number>`

The millisecond timestamp when the `Event` object was created.

event.type

- Type: `<string>`

The event type identifier.

Class: EventTarget

eventTarget.addEventListener(type, listener[, options])

- `type` `<string>`
- `listener` `<Function>` | `<EventListener>`
- `options` `<Object>`
 - `once` `<boolean>` When `true`, the listener is automatically removed when it is first invoked. **Default:** `false`.
 - `passive` `<boolean>` When `true`, serves as a hint that the listener will not call the `Event` object's `preventDefault()` method. **Default:** `false`.
 - `capture` `<boolean>` Not directly used by Node.js. Added for API completeness. **Default:** `false`.

Adds a new handler for the `type` event. Any given `listener` is added only once per `type` and per `capture` option value.

If the `once` option is `true`, the `listener` is removed after the next time a `type` event is dispatched.

The `capture` option is not used by Node.js in any functional way other than tracking registered event listeners per the `EventTarget` specification. Specifically, the `capture` option is used as part of the key when registering a `listener`. Any individual `listener` may be added once with `capture = false`, and once with `capture = true`.

```
function handler(event) {}

const target = new EventTarget();
target.addEventListener('foo', handler, { capture: true }); // first
target.addEventListener('foo', handler, { capture: false }); // second

// Removes the second instance of handler
target.removeEventListener('foo', handler);

// Removes the first instance of handler
target.removeEventListener('foo', handler, { capture: true });
```

eventTarget.dispatchEvent(event)

- `event` `<Event>`
- Returns: `<boolean>` `true` if either event's `cancelable` attribute value is `false` or its `preventDefault()` method was not invoked, otherwise `false`.

Dispatches the `event` to the list of handlers for `event.type`.

The registered event listeners is synchronously invoked in the order they were registered.

eventTarget.removeEventListener(type, listener)

- `type` `<string>`
- `listener` `<Function>` | `<EventListener>`
- `options` `<Object>`
 - `capture` `<boolean>`

Removes the `listener` from the list of handlers for event `type`.

Class: `NodeEventTarget`

- Extends: `<EventTarget>`

The `NodeEventTarget` is a Node.js-specific extension to `EventTarget` that emulates a subset of the `EventEmitter` API.

`nodeEventTarget.addListener(type, listener[, options])`

- `type` `<string>`
- `listener` `<Function>` | `<EventListener>`
- `options` `<Object>`
 - `once` `<boolean>`
- Returns: `<EventTarget>` this

Node.js-specific extension to the `EventTarget` class that emulates the equivalent `EventEmitter` API. The only difference between `addListener()` and `addEventListener()` is that `addListener()` will return a reference to the `EventTarget`.

`nodeEventTarget.eventNames()`

- Returns: `<string[]>`

Node.js-specific extension to the `EventTarget` class that returns an array of event `type` names for which event listeners are registered.

`nodeEventTarget.listenerCount(type)`

- `type` `<string>`
- Returns: `<number>`

Node.js-specific extension to the `EventTarget` class that returns the number of event listeners registered for the `type`.

`nodeEventTarget.off(type, listener)`

- `type` `<string>`
- `listener` `<Function>` | `<EventListener>`
- Returns: `<EventTarget>` this

Node.js-specific alias for `eventTarget.removeListener()`.

`nodeEventTarget.on(type, listener[, options])`

- `type` `<string>`
- `listener` `<Function>` | `<EventListener>`
- `options` `<Object>`

- `once` `<boolean>`
- Returns: `<EventTarget>` this

Node.js-specific alias for `eventTarget.addListener()`.

`nodeEventTarget.once(type, listener[, options])`

- `type` `<string>`
- `listener` `<Function>` | `<EventListener>`
- `options` `<Object>`
- Returns: `<EventTarget>` this

Node.js-specific extension to the `EventTarget` class that adds a `once` listener for the given event `type`. This is equivalent to calling `on` with the `once` option set to `true`.

`nodeEventTarget.removeAllListeners([type])`

- `type` `<string>`
- Returns: `<EventTarget>` this

Node.js-specific extension to the `EventTarget` class. If `type` is specified, removes all registered listeners for `type`, otherwise removes all registered listeners.

`nodeEventTarget.removeListener(type, listener)`

- `type` `<string>`
- `listener` `<Function>` | `<EventListener>`
- Returns: `<EventTarget>` this

Node.js-specific extension to the `EventTarget` class that removes the `listener` for the given `type`. The only difference between `removeListener()` and `removeEventListener()` is that `removeListener()` will return a reference to the `EventTarget`.

File system

Stability: 2 - Stable

Source Code: [lib/fs.js](#)

The `fs` module enables interacting with the file system in a way modeled on standard POSIX functions.

To use the promise-based APIs:

```
import * as fs from 'fs/promises'; const fs = require('fs/promises');
```

To use the callback and sync APIs:

```
import * as fs from 'fs'; const fs = require('fs');
```

All file system operations have synchronous, callback, and promise-based forms, and are accessible using both CommonJS syntax and ES6 Modules (ESM).

Promise example

Promise-based operations return a promise that is fulfilled when the asynchronous operation is complete.

```
import { unlink } from 'fs/promises';

try {
  await unlink('/tmp/hello');
  console.log('successfully deleted /tmp/hello');
} catch (error) {
  console.error('there was an error:', error.message);
}const { unlink } = require('fs/promises');

(async function(path) {
  try {
    await unlink(path);
    console.log(`successfully deleted ${path}`);
  } catch (error) {
    console.error('there was an error:', error.message);
  }
})('/tmp/hello');
```

Callback example

The callback form takes a completion callback function as its last argument and invokes the operation asynchronously. The arguments passed to the completion callback depend on the method, but the first argument is always reserved for an exception. If the operation is completed successfully, then the first argument is `null` or `undefined`.

```
import { unlink } from 'fs';

unlink('/tmp/hello', (err) => {
  if (err) throw err;
  console.log('successfully deleted /tmp/hello');
});const { unlink } = require('fs');

unlink('/tmp/hello', (err) => {
  if (err) throw err;
  console.log('successfully deleted /tmp/hello');
});
```

The callback-based versions of the `fs` module APIs are preferable over the use of the promise APIs when maximal performance (both in terms of execution time and memory allocation are required).

Synchronous example

The synchronous APIs block the Node.js event loop and further JavaScript execution until the operation is complete. Exceptions are thrown immediately and can be handled using `try...catch`, or can be allowed to bubble up.

```
import { unlinkSync } from 'fs';

try {
  unlinkSync('/tmp/hello');
  console.log('successfully deleted /tmp/hello');
} catch (err) {
  // handle the error
}const { unlinkSync } = require('fs');

try {
  unlinkSync('/tmp/hello');
  console.log('successfully deleted /tmp/hello');
} catch (err) {
  // handle the error
}
```

Promises API

The `fs/promises` API provides asynchronous file system methods that return promises.

The promise APIs use the underlying Node.js threadpool to perform file system operations off the event loop thread. These operations are not synchronized or threadsafe. Care must be taken when performing multiple concurrent modifications on the same file or data corruption may occur.

Class: `FileHandle`

A `<FileHandle>` object is an object wrapper for a numeric file descriptor.

Instances of the `<FileHandle>` object are created by the `fsPromises.open()` method.

All `<FileHandle>` objects are `<EventEmitter>`s.

If a `<FileHandle>` is not closed using the `filehandle.close()` method, it will try to automatically close the file descriptor and emit a process warning, helping to prevent memory leaks. Please do not rely on this behavior because it can be unreliable and the file may not be closed. Instead, always explicitly close `<FileHandle>`s. Node.js may change this behavior in the future.

Event: 'close'

The 'close' event is emitted when the `<FileHandle>` has been closed and can no longer be used.

`filehandle.appendFile(data[, options])`

- `data` `<string>` | `<Buffer>` | `<TypedArray>` | `<DataView>`
- `options` `<Object>` | `<string>`
 - `encoding` `<string>` | `<null>` **Default:** 'utf8'
- Returns: `<Promise>` Fulfills with `undefined` upon success.

Alias of `filehandle.writeFile()`.

When operating on file handles, the mode cannot be changed from what it was set to with `fsPromises.open()`. Therefore, this is equivalent to `filehandle.writeFile()`.

`filehandle.chmod(mode)`

- `mode` `<integer>` the file mode bit mask.
- Returns: `<Promise>` Fulfills with `undefined` upon success.

Modifies the permissions on the file. See `chmod(2)`.

`filehandle.chown(uid, gid)`

- `uid` `<integer>` The file's new owner's user id.
- `gid` `<integer>` The file's new group's group id.
- Returns: `<Promise>` Fulfills with `undefined` upon success.

Changes the ownership of the file. A wrapper for `chown(2)`.

`filehandle.close()`

- Returns: `<Promise>` Fulfills with `undefined` upon success.

Closes the file handle after waiting for any pending operation on the handle to complete.

```
import { open } from 'fs/promises';

let filehandle;
try {
  filehandle = await open('thefile.txt', 'r');
} finally {
  await filehandle?.close();
}
```

`filehandle.datasync()`

- Returns: `<Promise>` Fulfills with `undefined` upon success.

Forces all currently queued I/O operations associated with the file to the operating system's synchronized I/O completion state. Refer to the POSIX `fdatasync(2)` documentation for details.

Unlike `filehandle.sync` this method does not flush modified metadata.

`filehandle.fd`

- `<number>` The numeric file descriptor managed by the `<FileHandle>` object.

`filehandle.read(buffer, offset, length, position)`

- `buffer` `<Buffer>` | `<TypedArray>` | `<DataView>` A buffer that will be filled with the file data read.
- `offset` `<integer>` The location in the buffer at which to start filling. **Default:** `0`
- `length` `<integer>` The number of bytes to read. **Default:** `buffer.byteLength`
- `position` `<integer>` The location where to begin reading data from the file. If `null`, data will be read from the current file position, and the position will be updated. If `position` is an integer, the current file position will remain unchanged.
- Returns: `<Promise>` Fulfills upon success with an object with two properties:

- `bytesRead` `<integer>` The number of bytes read
- `buffer` `<Buffer>` | `<TypedArray>` | `<DataView>` A reference to the passed in `buffer` argument.

Reads data from the file and stores that in the given buffer.

If the file is not modified concurrently, the end-of-file is reached when the number of bytes read is zero.

`filehandle.read([options])`

- `options` `<Object>`
 - `buffer` `<Buffer>` | `<TypedArray>` | `<DataView>` A buffer that will be filled with the file data read. **Default:** `Buffer.alloc(16384)`
 - `offset` `<integer>` The location in the buffer at which to start filling. **Default:** `0`
 - `length` `<integer>` The number of bytes to read. **Default:** `buffer.byteLength`
 - `position` `<integer>` The location where to begin reading data from the file. If `null`, data will be read from the current file position, and the position will be updated. If `position` is an integer, the current file position will remain unchanged. **Default:** `null`
- Returns: `<Promise>` Fulfills upon success with an object with two properties:
 - `bytesRead` `<integer>` The number of bytes read
 - `buffer` `<Buffer>` | `<TypedArray>` | `<DataView>` A reference to the passed in `buffer` argument.

Reads data from the file and stores that in the given buffer.

If the file is not modified concurrently, the end-of-file is reached when the number of bytes read is zero.

`filehandle.readFile(options)`

- `options` `<Object>` | `<string>`
 - `encoding` `<string>` | `<null>` **Default:** `null`
 - `signal` `<AbortSignal>` allows aborting an in-progress `readFile`
- Returns: `<Promise>` Fulfills upon a successful read with the contents of the file. If no encoding is specified (using `options.encoding`), the data is returned as a `<Buffer>` object. Otherwise, the data will be a string.

Asynchronously reads the entire contents of a file.

If `options` is a string, then it specifies the `encoding`.

The `<FileHandle>` has to support reading.

If one or more `filehandle.read()` calls are made on a file handle and then a `filehandle.readFile()` call is made, the data will be read from the current position till the end of the file. It doesn't always read from the beginning of the file.

`filehandle.readv(buffers[, position])`

- `buffers` `<Buffer[]>` | `<TypedArray[]>` | `<DataView[]>`
- `position` `<integer>` The offset from the beginning of the file where the data should be read from. If `position` is not a `number`, the data will be read from the current position.
- Returns: `<Promise>` Fulfills upon success an object containing two properties:
 - `bytesRead` `<integer>` the number of bytes read
 - `buffers` `<Buffer[]>` | `<TypedArray[]>` | `<DataView[]>` property containing a reference to the `buffers` input.

Read from a file and write to an array of `<ArrayBufferView>`s

`filehandle.stat([options])`

- `options` `<Object>`

- `bigint` `<boolean>` Whether the numeric values in the returned `<fs.Stats>` object should be `bigint`. **Default:** `false`.
- Returns: `<Promise>` Fulfills with an `<fs.Stats>` for the file.

filehandle.sync()

- Returns: `<Promise>` Fulfills with `undefined` upon success.

Request that all data for the open file descriptor is flushed to the storage device. The specific implementation is operating system and device specific. Refer to the POSIX `fsync(2)` documentation for more detail.

filehandle.truncate(len)

- `len` `<integer>` **Default:** `0`
- Returns: `<Promise>` Fulfills with `undefined` upon success.

Truncates the file.

If the file was larger than `len` bytes, only the first `len` bytes will be retained in the file.

The following example retains only the first four bytes of the file:

```
import { open } from 'fs/promises';

let filehandle = null;
try {
  filehandle = await open('temp.txt', 'r+');
  await filehandle.truncate(4);
} finally {
  await filehandle?.close();
}
```

If the file previously was shorter than `len` bytes, it is extended, and the extended part is filled with null bytes (`'\0'`):

If `len` is negative then `0` will be used.

filehandle.utimes(atime, mtime)

- `atime` `<number>` | `<string>` | `<Date>`
- `mtime` `<number>` | `<string>` | `<Date>`
- Returns: `<Promise>`

Change the file system timestamps of the object referenced by the `<FileHandle>` then resolves the promise with no arguments upon success.

filehandle.write(buffer[, offset[, length[, position]]])

- `buffer` `<Buffer>` | `<TypedArray>` | `<DataView>` | `<string>` | `<Object>`
- `offset` `<integer>` The start position from within `buffer` where the data to write begins. **Default:** `0`
- `length` `<integer>` The number of bytes from `buffer` to write. **Default:** `buffer.byteLength`
- `position` `<integer>` The offset from the beginning of the file where the data from `buffer` should be written. If `position` is not a `number`, the data will be written at the current position. See the POSIX `pwrite(2)` documentation for more detail.
- Returns: `<Promise>`

Write `buffer` to the file.

If `buffer` is a plain object, it must have an own (not inherited) `toString` function property.

The promise is resolved with an object containing two properties:

- `bytesWritten` `<integer>` the number of bytes written
- `buffer` `<Buffer> | <TypedArray> | <DataView> | <string> | <Object>` a reference to the `buffer` written.

It is unsafe to use `filehandle.write()` multiple times on the same file without waiting for the promise to be resolved (or rejected). For this scenario, use `fs.createWriteStream()`.

On Linux, positional writes do not work when the file is opened in append mode. The kernel ignores the position argument and always appends the data to the end of the file.

`filehandle.write(string[, position[, encoding]])`

- `string` `<string> | <Object>`
- `position` `<integer>` The offset from the beginning of the file where the data from `string` should be written. If `position` is not a `number` the data will be written at the current position. See the POSIX `pwrite(2)` documentation for more detail.
- `encoding` `<string>` The expected string encoding. **Default:** `'utf8'`
- Returns: `<Promise>`

Write `string` to the file. If `string` is not a string, or an object with an own `toString` function property, the promise is rejected with an error.

The promise is resolved with an object containing two properties:

- `bytesWritten` `<integer>` the number of bytes written
- `buffer` `<string> | <Object>` a reference to the `string` written.

It is unsafe to use `filehandle.write()` multiple times on the same file without waiting for the promise to be resolved (or rejected). For this scenario, use `fs.createWriteStream()`.

On Linux, positional writes do not work when the file is opened in append mode. The kernel ignores the position argument and always appends the data to the end of the file.

`filehandle.writeFile(data, options)`

- `data` `<string> | <Buffer> | <TypedArray> | <DataView> | <Object>`
- `options` `<Object> | <string>`
 - `encoding` `<string> | <null>` The expected character encoding when `data` is a string. **Default:** `'utf8'`
- Returns: `<Promise>`

Asynchronously writes data to a file, replacing the file if it already exists. `data` can be a string, a buffer, or an object with an own `toString` function property. The promise is resolved with no arguments upon success.

If `options` is a string, then it specifies the `encoding`.

The `<FileHandle>` has to support writing.

It is unsafe to use `filehandle.writeFile()` multiple times on the same file without waiting for the promise to be resolved (or rejected).

If one or more `filehandle.write()` calls are made on a file handle and then a `filehandle.writeFile()` call is made, the data will be written from the current position till the end of the file. It doesn't always write from the beginning of the file.

`filehandle.writev(buffers[, position])`

- `buffers` `<Buffer[]> | <TypedArray[]> | <DataView[]>`

- `position` `<integer>` The offset from the beginning of the file where the data from `buffers` should be written. If `position` is not a `number`, the data will be written at the current position.
- Returns: `<Promise>`

Write an array of `<ArrayBufferView>`s to the file.

The promise is resolved with an object containing a two properties:

- `bytesWritten` `<integer>` the number of bytes written
- `buffers` `<Buffer[]> | <TypedArray[]> | <DataView[]>` a reference to the `buffers` input.

It is unsafe to call `writev()` multiple times on the same file without waiting for the promise to be resolved (or rejected).

On Linux, positional writes don't work when the file is opened in append mode. The kernel ignores the position argument and always appends the data to the end of the file.

fsPromises.access(path[, mode])

- `path` `<string> | <Buffer> | <URL>`
- `mode` `<integer>` **Default:** `fs.constants.F_OK`
- Returns: `<Promise>` Fulfills with `undefined` upon success.

Tests a user's permissions for the file or directory specified by `path`. The `mode` argument is an optional integer that specifies the accessibility checks to be performed. Check [File access constants](#) for possible values of `mode`. It is possible to create a mask consisting of the bitwise OR of two or more values (e.g. `fs.constants.W_OK | fs.constants.R_OK`).

If the accessibility check is successful, the promise is resolved with no value. If any of the accessibility checks fail, the promise is rejected with an `<Error>` object. The following example checks if the file `/etc/passwd` can be read and written by the current process.

```
import { access } from 'fs/promises';
import { constants } from 'fs';

try {
  await access('/etc/passwd', constants.R_OK | constants.W_OK);
  console.log('can access');
} catch {
  console.error('cannot access');
}
```

Using `fsPromises.access()` to check for the accessibility of a file before calling `fsPromises.open()` is not recommended. Doing so introduces a race condition, since other processes may change the file's state between the two calls. Instead, user code should open/read/write the file directly and handle the error raised if the file is not accessible.

fsPromises.appendFile(path, data[, options])

- `path` `<string> | <Buffer> | <URL> | <FileHandle>` filename or `<FileHandle>`
- `data` `<string> | <Buffer>`
- `options` `<Object> | <string>`
 - `encoding` `<string> | <null>` **Default:** `'utf8'`
 - `mode` `<integer>` **Default:** `0o666`
 - `flag` `<string>` See [support of file system flags](#). **Default:** `'a'`.
- Returns: `<Promise>` Fulfills with `undefined` upon success.

Asynchronously append data to a file, creating the file if it does not yet exist. `data` can be a string or a `<Buffer>`.

If `options` is a string, then it specifies the `encoding`.

The `path` may be specified as a `<FileHandle>` that has been opened for appending (using `fsPromises.open()`).

`fsPromises.chmod(path, mode)`

- `path` `<string>` | `<Buffer>` | `<URL>`
- `mode` `<string>` | `<integer>`
- Returns: `<Promise>` Fulfills with `undefined` upon success.

Changes the permissions of a file.

`fsPromises.chown(path, uid, gid)`

- `path` `<string>` | `<Buffer>` | `<URL>`
- `uid` `<integer>`
- `gid` `<integer>`
- Returns: `<Promise>` Fulfills with `undefined` upon success.

Changes the ownership of a file.

`fsPromises.copyFile(src, dest[, mode])`

- `src` `<string>` | `<Buffer>` | `<URL>` source filename to copy
- `dest` `<string>` | `<Buffer>` | `<URL>` destination filename of the copy operation
- `mode` `<integer>` Optional modifiers that specify the behavior of the copy operation. It is possible to create a mask consisting of the bitwise OR of two or more values (e.g. `fs.constants.COPYFILE_EXCL` | `fs.constants.COPYFILE_FICLONE`) **Default:** `0`.
 - `fs.constants.COPYFILE_EXCL`: The copy operation will fail if `dest` already exists.
 - `fs.constants.COPYFILE_FICLONE`: The copy operation will attempt to create a copy-on-write reflink. If the platform does not support copy-on-write, then a fallback copy mechanism is used.
 - `fs.constants.COPYFILE_FICLONE_FORCE`: The copy operation will attempt to create a copy-on-write reflink. If the platform does not support copy-on-write, then the operation will fail.
- Returns: `<Promise>` Fulfills with `undefined` upon success.

Asynchronously copies `src` to `dest`. By default, `dest` is overwritten if it already exists.

No guarantees are made about the atomicity of the copy operation. If an error occurs after the destination file has been opened for writing, an attempt will be made to remove the destination.

```
import { constants } from 'fs';
import { copyFile } from 'fs/promises';

try {
  await copyFile('source.txt', 'destination.txt');
  console.log('source.txt was copied to destination.txt');
} catch {
  console.log('The file could not be copied');
}

// By using COPYFILE_EXCL, the operation will fail if destination.txt exists.
try {
```

```

await copyFile('source.txt', 'destination.txt', constants.COPYFILE_EXCL);
console.log('source.txt was copied to destination.txt');
} catch {
  console.log('The file could not be copied');
}

```

fsPromises.cp(src, dest[, options])

Stability: 1 - Experimental

- `src` `<string>` | `<URL>` source path to copy.
- `dest` `<string>` | `<URL>` destination path to copy to.
- `options` `<Object>`
 - `dereference` `<boolean>` dereference symlinks. **Default:** `false`.
 - `errorOnExist` `<boolean>` when `force` is `false`, and the destination exists, throw an error. **Default:** `false`.
 - `filter` `<Function>` Function to filter copied files/directories. Return `true` to copy the item, `false` to ignore it. Can also return a `Promise` that resolves to `true` or `false` **Default:** `undefined`.
 - `force` `<boolean>` overwrite existing file or directory. _The copy operation will ignore errors if you set this to false and the destination exists. Use the `errorOnExist` option to change this behavior. **Default:** `true`.
 - `preserveTimestamps` `<boolean>` When `true` timestamps from `src` will be preserved. **Default:** `false`.
 - `recursive` `<boolean>` copy directories recursively **Default:** `false`
- Returns: `<Promise>` Fulfills with `undefined` upon success.

Asynchronously copies the entire directory structure from `src` to `dest`, including subdirectories and files.

When copying a directory to another directory, globs are not supported and behavior is similar to `cp dir1/ dir2/`.

fsPromises.lchmod(path, mode)

- `path` `<string>` | `<Buffer>` | `<URL>`
- `mode` `<integer>`
- Returns: `<Promise>` Fulfills with `undefined` upon success.

Changes the permissions on a symbolic link.

This method is only implemented on macOS.

fsPromises.lchown(path, uid, gid)

- `path` `<string>` | `<Buffer>` | `<URL>`
- `uid` `<integer>`
- `gid` `<integer>`
- Returns: `<Promise>` Fulfills with `undefined` upon success.

Changes the ownership on a symbolic link.

fsPromises.lutimes(path, atime, mtime)

- `path` `<string>` | `<Buffer>` | `<URL>`
- `atime` `<number>` | `<string>` | `<Date>`

- `mtime` `<number>` | `<string>` | `<Date>`
- Returns: `<Promise>` Fulfills with `undefined` upon success.

Changes the access and modification times of a file in the same way as `fsPromises.utimes()`, with the difference that if the path refers to a symbolic link, then the link is not dereferenced: instead, the timestamps of the symbolic link itself are changed.

`fsPromises.link(existingPath, newPath)`

- `existingPath` `<string>` | `<Buffer>` | `<URL>`
- `newPath` `<string>` | `<Buffer>` | `<URL>`
- Returns: `<Promise>` Fulfills with `undefined` upon success.

Creates a new link from the `existingPath` to the `newPath`. See the POSIX `link(2)` documentation for more detail.

`fsPromises.lstat(path[, options])`

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<Object>`
 - `bigint` `<boolean>` Whether the numeric values in the returned `<fs.Stats>` object should be `bigint`. **Default:** `false`.
- Returns: `<Promise>` Fulfills with the `<fs.Stats>` object for the given symbolic link `path`.

Equivalent to `fsPromises.stat()` unless `path` refers to a symbolic link, in which case the link itself is stat-ed, not the file that it refers to. Refer to the POSIX `lstat(2)` document for more detail.

`fsPromises.mkdir(path[, options])`

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<Object>` | `<integer>`
 - `recursive` `<boolean>` **Default:** `false`
 - `mode` `<string>` | `<integer>` Not supported on Windows. **Default:** `0o777`.
- Returns: `<Promise>` Upon success, fulfills with `undefined` if `recursive` is `false`, or the first directory path created if `recursive` is `true`.

Asynchronously creates a directory.

The optional `options` argument can be an integer specifying `mode` (permission and sticky bits), or an object with a `mode` property and a `recursive` property indicating whether parent directories should be created. Calling `fsPromises.mkdir()` when `path` is a directory that exists results in a rejection only when `recursive` is `false`.

`fsPromises.mkdtemp(prefix[, options])`

- `prefix` `<string>`
- `options` `<string>` | `<Object>`
 - `encoding` `<string>` **Default:** `'utf8'`
- Returns: `<Promise>` Fulfills with a string containing the filesystem path of the newly created temporary directory.

Creates a unique temporary directory. A unique directory name is generated by appending six random characters to the end of the provided `prefix`. Due to platform inconsistencies, avoid trailing `X` characters in `prefix`. Some platforms, notably the BSDs, can return more than six random characters, and replace trailing `X` characters in `prefix` with random characters.

The optional `options` argument can be a string specifying an encoding, or an object with an `encoding` property specifying the character encoding to use.

```
import { mkdtemp } from 'fs/promises';

try {
  await mkdtemp(path.join(os.tmpdir(), 'foo-'));
} catch (err) {
  console.error(err);
}
```

The `fsPromises.mkdtemp()` method will append the six randomly selected characters directly to the `prefix` string. For instance, given a directory `/tmp`, if the intention is to create a temporary directory *within* `/tmp`, the `prefix` must end with a trailing platform-specific path separator (`require('path').sep`).

fsPromises.open(path, flags[, mode])

- `path` `<string>` | `<Buffer>` | `<URL>`
- `flags` `<string>` | `<number>` See [support of file system flags](#). **Default:** `'r'`.
- `mode` `<string>` | `<integer>` Sets the file mode (permission and sticky bits) if the file is created. **Default:** `0o666` (readable and writable)
- Returns: `<Promise>` Fulfills with a `<FileHandle>` object.

Opens a `<FileHandle>`.

Refer to the POSIX [open\(2\)](#) documentation for more detail.

Some characters (`<` `>` `:` `"` `/` `\` `|` `?` `*`) are reserved under Windows as documented by [Naming Files, Paths, and Namespaces](#). Under NTFS, if the filename contains a colon, Node.js will open a file system stream, as described by [this MSDN page](#).

fsPromises.opendir(path[, options])

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<Object>`
 - `encoding` `<string>` | `<null>` **Default:** `'utf8'`
 - `bufferSize` `<number>` Number of directory entries that are buffered internally when reading from the directory. Higher values lead to better performance but higher memory usage. **Default:** `32`
- Returns: `<Promise>` Fulfills with an `<fs.Dir>`.

Asynchronously open a directory for iterative scanning. See the POSIX [opendir\(3\)](#) documentation for more detail.

Creates an `<fs.Dir>`, which contains all further functions for reading from and cleaning up the directory.

The `encoding` option sets the encoding for the `path` while opening the directory and subsequent read operations.

Example using async iteration:

```
import { opendir } from 'fs/promises';

try {
  const dir = await opendir('.');
  for await (const dirent of dir)
    console.log(dirent.name);
} catch (err) {
```

```
console.error(err);
}
```

When using the async iterator, the `<fs.Dir>` object will be automatically closed after the iterator exits.

fsPromises.readdir(path[, options])

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<string>` | `<Object>`
 - `encoding` `<string>` **Default:** `'utf8'`
 - `withFileTypes` `<boolean>` **Default:** `false`
- Returns: `<Promise>` Fulfills with an array of the names of the files in the directory excluding `'.'` and `'..'`.

Reads the contents of a directory.

The optional `options` argument can be a string specifying an encoding, or an object with an `encoding` property specifying the character encoding to use for the filenames. If the `encoding` is set to `'buffer'`, the filenames returned will be passed as `<Buffer>` objects.

If `options.withFileTypes` is set to `true`, the resolved array will contain `<fs.Dirent>` objects.

```
import { readdir } from 'fs/promises';

try {
  const files = await readdir(path);
  for (const file of files)
    console.log(file);
} catch (err) {
  console.error(err);
}
```

fsPromises.readFile(path[, options])

- `path` `<string>` | `<Buffer>` | `<URL>` | `<FileHandle>` filename or `FileHandle`
- `options` `<Object>` | `<string>`
 - `encoding` `<string>` | `<null>` **Default:** `null`
 - `flag` `<string>` See [support of file system flags](#). **Default:** `'r'`.
 - `signal` `<AbortSignal>` allows aborting an in-progress `readFile`
- Returns: `<Promise>` Fulfills with the contents of the file.

Asynchronously reads the entire contents of a file.

If no encoding is specified (using `options.encoding`), the data is returned as a `<Buffer>` object. Otherwise, the data will be a string.

If `options` is a string, then it specifies the encoding.

When the `path` is a directory, the behavior of `fsPromises.readFile()` is platform-specific. On macOS, Linux, and Windows, the promise will be rejected with an error. On FreeBSD, a representation of the directory's contents will be returned.

It is possible to abort an ongoing `readFile` using an `<AbortSignal>`. If a request is aborted the promise returned is rejected with an `AbortError`:


```
import { readFile } from 'fs/promises';

try {
  const controller = new AbortController();
  const { signal } = controller;
  const promise = readFile(fileName, { signal });

  // Abort the request before the promise settles.
  controller.abort();

  await promise;
} catch (err) {
  // When a request is aborted - err is an AbortError
  console.error(err);
}
```

Aborting an ongoing request does not abort individual operating system requests but rather the internal buffering `fs.readFile` performs.

Any specified `<FileHandle>` has to support reading.

fsPromises.readlink(path[, options])

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<string>` | `<Object>`
 - `encoding` `<string>` **Default:** `'utf8'`
- Returns: `<Promise>` Fulfills with the `linkString` upon success.

Reads the contents of the symbolic link referred to by `path`. See the POSIX `readlink(2)` documentation for more detail. The promise is resolved with the `linkString` upon success.

The optional `options` argument can be a string specifying an encoding, or an object with an `encoding` property specifying the character encoding to use for the link path returned. If the `encoding` is set to `'buffer'`, the link path returned will be passed as a `<Buffer>` object.

fsPromises.realpath(path[, options])

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<string>` | `<Object>`
 - `encoding` `<string>` **Default:** `'utf8'`
- Returns: `<Promise>` Fulfills with the resolved path upon success.

Determines the actual location of `path` using the same semantics as the `fs.realpath.native()` function.

Only paths that can be converted to UTF8 strings are supported.

The optional `options` argument can be a string specifying an encoding, or an object with an `encoding` property specifying the character encoding to use for the path. If the `encoding` is set to `'buffer'`, the path returned will be passed as a `<Buffer>` object.

On Linux, when Node.js is linked against musl libc, the procfs file system must be mounted on `/proc` in order for this function to work. Glibc does not have this restriction.

fsPromises.rename(oldPath, newPath)

- `oldPath` `<string>` | `<Buffer>` | `<URL>`

- `newPath` `<string>` | `<Buffer>` | `<URL>`
- Returns: `<Promise>` Fulfills with `undefined` upon success.

Renames `oldPath` to `newPath`.

`fsPromises.rmdir(path[, options])`

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<Object>`
 - `maxRetries` `<integer>` If an `EBUSY`, `EMFILE`, `ENFILE`, `ENOTEMPTY`, or `EPERM` error is encountered, Node.js retries the operation with a linear backoff wait of `retryDelay` milliseconds longer on each try. This option represents the number of retries. This option is ignored if the `recursive` option is not `true`. **Default:** `0`.
 - `recursive` `<boolean>` If `true`, perform a recursive directory removal. In recursive mode, operations are retried on failure. **Default:** `false`. **Deprecated.**
 - `retryDelay` `<integer>` The amount of time in milliseconds to wait between retries. This option is ignored if the `recursive` option is not `true`. **Default:** `100`.
- Returns: `<Promise>` Fulfills with `undefined` upon success.

Removes the directory identified by `path`.

Using `fsPromises.rmdir()` on a file (not a directory) results in the promise being rejected with an `ENOENT` error on Windows and an `ENOTDIR` error on POSIX.

To get a behavior similar to the `rm -rf` Unix command, use `fsPromises.rm()` with options `{ recursive: true, force: true }`.

`fsPromises.rm(path[, options])`

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<Object>`
 - `force` `<boolean>` When `true`, exceptions will be ignored if `path` does not exist. **Default:** `false`.
 - `maxRetries` `<integer>` If an `EBUSY`, `EMFILE`, `ENFILE`, `ENOTEMPTY`, or `EPERM` error is encountered, Node.js will retry the operation with a linear backoff wait of `retryDelay` milliseconds longer on each try. This option represents the number of retries. This option is ignored if the `recursive` option is not `true`. **Default:** `0`.
 - `recursive` `<boolean>` If `true`, perform a recursive directory removal. In recursive mode operations are retried on failure. **Default:** `false`.
 - `retryDelay` `<integer>` The amount of time in milliseconds to wait between retries. This option is ignored if the `recursive` option is not `true`. **Default:** `100`.
- Returns: `<Promise>` Fulfills with `undefined` upon success.

Removes files and directories (modeled on the standard POSIX `rm` utility).

`fsPromises.stat(path[, options])`

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<Object>`
 - `bigint` `<boolean>` Whether the numeric values in the returned `<fs.Stats>` object should be `bigint`. **Default:** `false`.
- Returns: `<Promise>` Fulfills with the `<fs.Stats>` object for the given `path`.

`fsPromises.symlink(target, path[, type])`

- `target` `<string>` | `<Buffer>` | `<URL>`
- `path` `<string>` | `<Buffer>` | `<URL>`

- `type` `<string>` **Default:** `'file'`
- Returns: `<Promise>` Fulfills with `undefined` upon success.

Creates a symbolic link.

The `type` argument is only used on Windows platforms and can be one of `'dir'`, `'file'`, or `'junction'`. Windows junction points require the destination path to be absolute. When using `'junction'`, the `target` argument will automatically be normalized to absolute path.

fsPromises.truncate(path[, len])

- `path` `<string>` | `<Buffer>` | `<URL>`
- `len` `<integer>` **Default:** `0`
- Returns: `<Promise>` Fulfills with `undefined` upon success.

Truncates (shortens or extends the length) of the content at `path` to `len` bytes.

fsPromises.unlink(path)

- `path` `<string>` | `<Buffer>` | `<URL>`
- Returns: `<Promise>` Fulfills with `undefined` upon success.

If `path` refers to a symbolic link, then the link is removed without affecting the file or directory to which that link refers. If the `path` refers to a file path that is not a symbolic link, the file is deleted. See the POSIX `unlink(2)` documentation for more detail.

fsPromises.utimes(path, atime, mtime)

- `path` `<string>` | `<Buffer>` | `<URL>`
- `atime` `<number>` | `<string>` | `<Date>`
- `mtime` `<number>` | `<string>` | `<Date>`
- Returns: `<Promise>` Fulfills with `undefined` upon success.

Change the file system timestamps of the object referenced by `path`.

The `atime` and `mtime` arguments follow these rules:

- Values can be either numbers representing Unix epoch time, `Date` s, or a numeric string like `'123456789.0'`.
- If the value can not be converted to a number, or is `NaN`, `Infinity` or `-Infinity`, an `Error` will be thrown.

fsPromises.watch(filename[, options])

- `filename` `<string>` | `<Buffer>` | `<URL>`
- `options` `<string>` | `<Object>`
 - `persistent` `<boolean>` Indicates whether the process should continue to run as long as files are being watched. **Default:** `true`.
 - `recursive` `<boolean>` Indicates whether all subdirectories should be watched, or only the current directory. This applies when a directory is specified, and only on supported platforms (See [caveats](#)). **Default:** `false`.
 - `encoding` `<string>` Specifies the character encoding to be used for the filename passed to the listener. **Default:** `'utf8'`.
 - `signal` `<AbortSignal>` An `<AbortSignal>` used to signal when the watcher should stop.
- Returns: `<AsyncIterator>` of objects with the properties:
 - `eventType` `<string>` The type of change
 - `filename` `<string>` | `<Buffer>` The name of the file changed.

Returns an async iterator that watches for changes on `filename`, where `filename` is either a file or a directory.

```

const { watch } = require('fs/promises');

const ac = new AbortController();
const { signal } = ac;
setTimeout(() => ac.abort(), 10000);

(async () => {
  try {
    const watcher = watch(__filename, { signal });
    for await (const event of watcher)
      console.log(event);
  } catch (err) {
    if (err.name === 'AbortError')
      return;
    throw err;
  }
})();

```

On most platforms, `'rename'` is emitted whenever a filename appears or disappears in the directory.

All the `caveats` for `fs.watch()` also apply to `fsPromises.watch()`.

fsPromises.writeFile(file, data[, options])

- `file` `<string>` | `<Buffer>` | `<URL>` | `<FileHandle>` filename or `FileHandle`
- `data` `<string>` | `<Buffer>` | `<TypedArray>` | `<DataView>` | `<Object>` | `<AsyncIterable>` | `<Iterable>` | `<Stream>`
- `options` `<Object>` | `<string>`
 - `encoding` `<string>` | `<null>` **Default:** `'utf8'`
 - `mode` `<integer>` **Default:** `0o666`
 - `flag` `<string>` See [support of file system flags](#). **Default:** `'w'`.
 - `signal` `<AbortSignal>` allows aborting an in-progress write
- Returns: `<Promise>` Fulfills with `undefined` upon success.

Asynchronously writes data to a file, replacing the file if it already exists. `data` can be a string, a `<Buffer>`, or, an object with an own (not inherited) `toString` function property.

The `encoding` option is ignored if `data` is a buffer.

If `options` is a string, then it specifies the encoding.

Any specified `<FileHandle>` has to support writing.

It is unsafe to use `fsPromises.writeFile()` multiple times on the same file without waiting for the promise to be settled.

Similarly to `fsPromises.readFile` - `fsPromises.writeFile` is a convenience method that performs multiple `write` calls internally to write the buffer passed to it. For performance sensitive code consider using `fs.createWriteStream()`.

It is possible to use an `<AbortSignal>` to cancel an `fsPromises.writeFile()`. Cancellation is "best effort", and some amount of data is likely still to be written.

```
import { writeFile } from 'fs/promises';
import { Buffer } from 'buffer';

try {
  const controller = new AbortController();
  const { signal } = controller;
  const data = new Uint8Array(Buffer.from('Hello Node.js'));
  const promise = writeFile('message.txt', data, { signal });

  // Abort the request before the promise settles.
  controller.abort();

  await promise;
} catch (err) {
  // When a request is aborted - err is an AbortError
  console.error(err);
}
```

Aborting an ongoing request does not abort individual operating system requests but rather the internal buffering `fs.writeFile` performs.

Callback API

The callback APIs perform all operations asynchronously, without blocking the event loop, then invoke a callback function upon completion or error.

The callback APIs use the underlying Node.js threadpool to perform file system operations off the event loop thread. These operations are not synchronized or threadsafe. Care must be taken when performing multiple concurrent modifications on the same file or data corruption may occur.

`fs.access(path[, mode], callback)`

- `path` `<string> | <Buffer> | <URL>`
- `mode` `<integer>` **Default:** `fs.constants.F_OK`
- `callback` `<Function>`
 - `err` `<Error>`

Tests a user's permissions for the file or directory specified by `path`. The `mode` argument is an optional integer that specifies the accessibility checks to be performed. Check [File access constants](#) for possible values of `mode`. It is possible to create a mask consisting of the bitwise OR of two or more values (e.g. `fs.constants.W_OK | fs.constants.R_OK`).

The final argument, `callback`, is a callback function that is invoked with a possible error argument. If any of the accessibility checks fail, the error argument will be an `Error` object. The following examples check if `package.json` exists, and if it is readable or writable.

```
import { access, constants } from 'fs';

const file = 'package.json';

// Check if the file exists in the current directory.
access(file, constants.F_OK, (err) => {
  console.log(`${file} ${err ? 'does not exist' : 'exists'}`);
});
```

```

// Check if the file is readable.
access(file, constants.R_OK, (err) => {
  console.log(`${file} ${err ? 'is not readable' : 'is readable'}`);
});

// Check if the file is writable.
access(file, constants.W_OK, (err) => {
  console.log(`${file} ${err ? 'is not writable' : 'is writable'}`);
});

// Check if the file exists in the current directory, and if it is writable.
access(file, constants.F_OK | constants.W_OK, (err) => {
  if (err) {
    console.error(
      `${file} ${err.code === 'ENOENT' ? 'does not exist' : 'is read-only'}`);
  } else {
    console.log(`${file} exists, and it is writable`);
  }
});

```

Do not use `fs.access()` to check for the accessibility of a file before calling `fs.open()`, `fs.readFile()` or `fs.writeFile()`. Doing so introduces a race condition, since other processes may change the file's state between the two calls. Instead, user code should open/read/write the file directly and handle the error raised if the file is not accessible.

write (NOT RECOMMENDED)

```

import { access, open, close } from 'fs';

access('myfile', (err) => {
  if (!err) {
    console.error('myfile already exists');
    return;
  }

  open('myfile', 'wx', (err, fd) => {
    if (err) throw err;

    try {
      writeMyData(fd);
    } finally {
      close(fd, (err) => {
        if (err) throw err;
      });
    }
  });
});

```

write (RECOMMENDED)

```
import { open, close } from 'fs';

open('myfile', 'wx', (err, fd) => {
  if (err) {
    if (err.code === 'EEXIST') {
      console.error('myfile already exists');
      return;
    }

    throw err;
  }

  try {
    writeMyData(fd);
  } finally {
    close(fd, (err) => {
      if (err) throw err;
    });
  }
});
```

read (NOT RECOMMENDED)

```
import { access, open, close } from 'fs';
access('myfile', (err) => {
  if (err) {
    if (err.code === 'ENOENT') {
      console.error('myfile does not exist');
      return;
    }

    throw err;
  }

  open('myfile', 'r', (err, fd) => {
    if (err) throw err;

    try {
      readMyData(fd);
    } finally {
      close(fd, (err) => {
        if (err) throw err;
      });
    }
  });
});
```

read (RECOMMENDED)

```
import { open, close } from 'fs';

open('myfile', 'r', (err, fd) => {
  if (err) {
    if (err.code === 'ENOENT') {
      console.error('myfile does not exist');
      return;
    }

    throw err;
  }

  try {
    readMyData(fd);
  } finally {
    close(fd, (err) => {
      if (err) throw err;
    });
  }
});
```

The "not recommended" examples above check for accessibility and then use the file; the "recommended" examples are better because they use the file directly and handle the error, if any.

In general, check for the accessibility of a file only if the file will not be used directly, for example when its accessibility is a signal from another process.

On Windows, access-control policies (ACLs) on a directory may limit access to a file or directory. The `fs.access()` function, however, does not check the ACL and therefore may report that a path is accessible even if the ACL restricts the user from reading or writing to it.

fs.appendFile(path, data[, options], callback)

- `path` `<string> | <Buffer> | <URL> | <number>` filename or file descriptor
- `data` `<string> | <Buffer>`
- `options` `<Object> | <string>`
 - `encoding` `<string> | <null>` **Default:** `'utf8'`
 - `mode` `<integer>` **Default:** `0o666`
 - `flag` `<string>` See [support of file system flags](#). **Default:** `'a'`.
- `callback` `<Function>`
 - `err` `<Error>`

Asynchronously append data to a file, creating the file if it does not yet exist. `data` can be a string or a `<Buffer>`.

```
import { appendFile } from 'fs';

appendFile('message.txt', 'data to append', (err) => {
  if (err) throw err;
  console.log('The "data to append" was appended to file!');
});
```


If `options` is a string, then it specifies the encoding:

```
import { appendFile } from 'fs';

appendFile('message.txt', 'data to append', 'utf8', callback);
```

The `path` may be specified as a numeric file descriptor that has been opened for appending (using `fs.open()` or `fs.openSync()`). The file descriptor will not be closed automatically.

```
import { open, close, appendFile } from 'fs';

function closeFd(fd) {
  close(fd, (err) => {
    if (err) throw err;
  });
}

open('message.txt', 'a', (err, fd) => {
  if (err) throw err;

  try {
    appendFile(fd, 'data to append', 'utf8', (err) => {
      closeFd(fd);
      if (err) throw err;
    });
  } catch (err) {
    closeFd(fd);
    throw err;
  }
});
```

fs.chmod(path, mode, callback)

- `path` `<string> | <Buffer> | <URL>`
- `mode` `<string> | <integer>`
- `callback` `<Function>`
 - `err` `<Error>`

Asynchronously changes the permissions of a file. No arguments other than a possible exception are given to the completion callback.

See the POSIX `chmod(2)` documentation for more detail.

```
import { chmod } from 'fs';

chmod('my_file.txt', 0o775, (err) => {
  if (err) throw err;
  console.log('The permissions for file "my_file.txt" have been changed!');
});
```

File modes

The `mode` argument used in both the `fs.chmod()` and `fs.chmodSync()` methods is a numeric bitmask created using a logical OR of the following constants:

Constant	Octal	Description
<code>fs.constants.S_IRUSR</code>	<code>0o400</code>	read by owner
<code>fs.constants.S_IWUSR</code>	<code>0o200</code>	write by owner
<code>fs.constants.S_IXUSR</code>	<code>0o100</code>	execute/search by owner
<code>fs.constants.S_IRGRP</code>	<code>0o40</code>	read by group
<code>fs.constants.S_IWGRP</code>	<code>0o20</code>	write by group
<code>fs.constants.S_IXGRP</code>	<code>0o10</code>	execute/search by group
<code>fs.constants.S_IROTH</code>	<code>0o4</code>	read by others
<code>fs.constants.S_IWOTH</code>	<code>0o2</code>	write by others
<code>fs.constants.S_IXOTH</code>	<code>0o1</code>	execute/search by others

An easier method of constructing the `mode` is to use a sequence of three octal digits (e.g. `765`). The left-most digit (`7` in the example), specifies the permissions for the file owner. The middle digit (`6` in the example), specifies permissions for the group. The right-most digit (`5` in the example), specifies the permissions for others.

Number	Description
<code>7</code>	read, write, and execute
<code>6</code>	read and write
<code>5</code>	read and execute
<code>4</code>	read only
<code>3</code>	write and execute
<code>2</code>	write only
<code>1</code>	execute only
<code>0</code>	no permission

For example, the octal value `0o765` means:

- The owner may read, write and execute the file.
- The group may read and write the file.
- Others may read and execute the file.

When using raw numbers where file modes are expected, any value larger than `0o777` may result in platform-specific behaviors that are not supported to work consistently. Therefore constants like `S_ISVTX`, `S_ISGID` or `S_ISUID` are not exposed in `fs.constants`.

Caveats: on Windows only the write permission can be changed, and the distinction among the permissions of group, owner or others is not implemented.

`fs.chown(path, uid, gid, callback)`

- `path` `<string>` | `<Buffer>` | `<URL>`
- `uid` `<integer>`
- `gid` `<integer>`
- `callback` `<Function>`
 - `err` `<Error>`

Asynchronously changes owner and group of a file. No arguments other than a possible exception are given to the completion callback.

See the POSIX `chown(2)` documentation for more detail.

`fs.close(fd[, callback])`

- `fd` `<integer>`
- `callback` `<Function>`
 - `err` `<Error>`

Closes the file descriptor. No arguments other than a possible exception are given to the completion callback.

Calling `fs.close()` on any file descriptor (`fd`) that is currently in use through any other `fs` operation may lead to undefined behavior.

See the POSIX `close(2)` documentation for more detail.

`fs.copyFile(src, dest[, mode], callback)`

- `src` `<string>` | `<Buffer>` | `<URL>` source filename to copy
- `dest` `<string>` | `<Buffer>` | `<URL>` destination filename of the copy operation
- `mode` `<integer>` modifiers for copy operation. **Default:** `0`.
- `callback` `<Function>`

Asynchronously copies `src` to `dest`. By default, `dest` is overwritten if it already exists. No arguments other than a possible exception are given to the callback function. Node.js makes no guarantees about the atomicity of the copy operation. If an error occurs after the destination file has been opened for writing, Node.js will attempt to remove the destination.

`mode` is an optional integer that specifies the behavior of the copy operation. It is possible to create a mask consisting of the bitwise OR of two or more values (e.g. `fs.constants.COPYFILE_EXCL` | `fs.constants.COPYFILE_FICLONE`).

- `fs.constants.COPYFILE_EXCL`: The copy operation will fail if `dest` already exists.
- `fs.constants.COPYFILE_FICLONE`: The copy operation will attempt to create a copy-on-write reflink. If the platform does not support copy-on-write, then a fallback copy mechanism is used.
- `fs.constants.COPYFILE_FICLONE_FORCE`: The copy operation will attempt to create a copy-on-write reflink. If the platform does not support copy-on-write, then the operation will fail.

```
import { copyFile, constants } from 'fs';

function callback(err) {
  if (err) throw err;
  console.log('source.txt was copied to destination.txt');
}

// destination.txt will be created or overwritten by default.
copyFile('source.txt', 'destination.txt', callback);

// By using COPYFILE_EXCL, the operation will fail if destination.txt exists.
copyFile('source.txt', 'destination.txt', constants.COPYFILE_EXCL, callback);
```

fs.cp(src, dest[, options], callback)

Stability: 1 - Experimental

- `src` `<string>` | `<URL>` source path to copy.
- `dest` `<string>` | `<URL>` destination path to copy to.
- `options` `<Object>`
 - `dereference` `<boolean>` dereference symlinks. **Default:** `false`.
 - `errorOnExist` `<boolean>` when `force` is `false`, and the destination exists, throw an error. **Default:** `false`.
 - `filter` `<Function>` Function to filter copied files/directories. Return `true` to copy the item, `false` to ignore it. Can also return a `Promise` that resolves to `true` or `false` **Default:** `undefined`.
 - `force` `<boolean>` overwrite existing file or directory. _The copy operation will ignore errors if you set this to false and the destination exists. Use the `errorOnExist` option to change this behavior. **Default:** `true`.
 - `preserveTimestamps` `<boolean>` When `true` timestamps from `src` will be preserved. **Default:** `false`.
 - `recursive` `<boolean>` copy directories recursively **Default:** `false`
- `callback` `<Function>`

Asynchronously copies the entire directory structure from `src` to `dest`, including subdirectories and files.

When copying a directory to another directory, globs are not supported and behavior is similar to `cp dir1/ dir2/`.

fs.createReadStream(path[, options])

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<string>` | `<Object>`
 - `flags` `<string>` See [support of file system flags](#). **Default:** `'r'`.
 - `encoding` `<string>` **Default:** `null`
 - `fd` `<integer>` | `<FileHandle>` **Default:** `null`
 - `mode` `<integer>` **Default:** `0o666`
 - `autoClose` `<boolean>` **Default:** `true`
 - `emitClose` `<boolean>` **Default:** `true`
 - `start` `<integer>`
 - `end` `<integer>` **Default:** `Infinity`
 - `highWaterMark` `<integer>` **Default:** `64 * 1024`
 - `fs` `<Object>` | `<>null>` **Default:** `null`
- Returns: `<fs.ReadStream>` See [Readable Stream](#).

Unlike the 16 kb default `highWaterMark` for a readable stream, the stream returned by this method has a default `highWaterMark` of 64 kb.

`options` can include `start` and `end` values to read a range of bytes from the file instead of the entire file. Both `start` and `end` are inclusive and start counting at 0, allowed values are in the `[0, Number.MAX_SAFE_INTEGER]` range. If `fd` is specified and `start` is omitted or `undefined`, `fs.createReadStream()` reads sequentially from the current file position. The `encoding` can be any one of those accepted by `<Buffer>`.

If `fd` is specified, `ReadStream` will ignore the `path` argument and will use the specified file descriptor. This means that no `'open'` event will be emitted. `fd` should be blocking; non-blocking `fd`s should be passed to `<net.Socket>`.

If `fd` points to a character device that only supports blocking reads (such as keyboard or sound card), read operations do not finish until data is available. This can prevent the process from exiting and the stream from closing naturally.

By default, the stream will emit a `'close'` event after it has been destroyed, like most `Readable` streams. Set the `emitClose` option to `false` to change this behavior.

By providing the `fs` option, it is possible to override the corresponding `fs` implementations for `open`, `read`, and `close`. When providing the `fs` option, overrides for `open`, `read`, and `close` are required.

```
import { createReadStream } from 'fs';

// Create a stream from some character device.
const stream = createReadStream('/dev/input/event0');
setTimeout(() => {
  stream.close(); // This may not close the stream.
  // Artificially marking end-of-stream, as if the underlying resource had
  // indicated end-of-file by itself, allows the stream to close.
  // This does not cancel pending read operations, and if there is such an
  // operation, the process may still not be able to exit successfully
  // until it finishes.
  stream.push(null);
  stream.read(0);
}, 100);
```

If `autoClose` is false, then the file descriptor won't be closed, even if there's an error. It is the application's responsibility to close it and make sure there's no file descriptor leak. If `autoClose` is set to true (default behavior), on `'error'` or `'end'` the file descriptor will be closed automatically.

`mode` sets the file mode (permission and sticky bits), but only if the file was created.

An example to read the last 10 bytes of a file which is 100 bytes long:

```
import { createReadStream } from 'fs';

createReadStream('sample.txt', { start: 90, end: 99 });
```

If `options` is a string, then it specifies the encoding.

`fs.createWriteStream(path[, options])`

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<string>` | `<Object>`
 - `flags` `<string>` See [support of file system flags](#). **Default:** `'w'`.
 - `encoding` `<string>` **Default:** `'utf8'`
 - `fd` `<integer>` | `<FileHandle>` **Default:** `null`
 - `mode` `<integer>` **Default:** `0o666`
 - `autoClose` `<boolean>` **Default:** `true`
 - `emitClose` `<boolean>` **Default:** `true`
 - `start` `<integer>`
 - `fs` `<Object>` | `<>null>` **Default:** `null`

- Returns: `<fs.WriteStream>` See `WritableStream`.

`options` may also include a `start` option to allow writing data at some position past the beginning of the file, allowed values are in the `[0, Number.MAX_SAFE_INTEGER]` range. Modifying a file rather than replacing it may require the `flags` option to be set to `r+` rather than the default `w`. The `encoding` can be any one of those accepted by `<Buffer>`.

If `autoClose` is set to `true` (default behavior) on `'error'` or `'finish'` the file descriptor will be closed automatically. If `autoClose` is `false`, then the file descriptor won't be closed, even if there's an error. It is the application's responsibility to close it and make sure there's no file descriptor leak.

By default, the stream will emit a `'close'` event after it has been destroyed, like most `writable` streams. Set the `emitClose` option to `false` to change this behavior.

By providing the `fs` option it is possible to override the corresponding `fs` implementations for `open`, `write`, `writv` and `close`. Overriding `write()` without `writv()` can reduce performance as some optimizations (`_writv()`) will be disabled. When providing the `fs` option, overrides for `open`, `close`, and at least one of `write` and `writv` are required.

Like `<fs.ReadStream>`, if `fd` is specified, `<fs.WriteStream>` will ignore the `path` argument and will use the specified file descriptor. This means that no `'open'` event will be emitted. `fd` should be blocking; non-blocking `fd`s should be passed to `<net.Socket>`.

If `options` is a string, then it specifies the encoding.

fs.exists(path, callback)

Stability: 0 - Deprecated: Use `fs.stat()` or `fs.access()` instead.

- `path` `<string>` | `<Buffer>` | `<URL>`
- `callback` `<Function>`
 - `exists` `<boolean>`

Test whether or not the given path exists by checking with the file system. Then call the `callback` argument with either `true` or `false`:

```
import { exists } from 'fs';

exists('/etc/passwd', (e) => {
  console.log(e ? 'it exists' : 'no passwd!');
});
```

The parameters for this callback are not consistent with other Node.js callbacks. Normally, the first parameter to a Node.js callback is an `err` parameter, optionally followed by other parameters. The `fs.exists()` callback has only one boolean parameter. This is one reason `fs.access()` is recommended instead of `fs.exists()`.

Using `fs.exists()` to check for the existence of a file before calling `fs.open()`, `fs.readFile()` or `fs.writeFile()` is not recommended. Doing so introduces a race condition, since other processes may change the file's state between the two calls. Instead, user code should open/read/write the file directly and handle the error raised if the file does not exist.

write (NOT RECOMMENDED)

```
import { exists, open, close } from 'fs';

exists('myfile', (e) => {
  if (e) {
```

```

    console.error('myfile already exists');
  } else {
    open('myfile', 'wx', (err, fd) => {
      if (err) throw err;

      try {
        writeMyData(fd);
      } finally {
        close(fd, (err) => {
          if (err) throw err;
        });
      }
    });
  }
});

```

write (RECOMMENDED)

```

import { open, close } from 'fs';
open('myfile', 'wx', (err, fd) => {
  if (err) {
    if (err.code === 'EEXIST') {
      console.error('myfile already exists');
      return;
    }

    throw err;
  }

  try {
    writeMyData(fd);
  } finally {
    close(fd, (err) => {
      if (err) throw err;
    });
  }
});

```

read (NOT RECOMMENDED)

```

import { open, close, exists } from 'fs';

exists('myfile', (e) => {
  if (e) {
    open('myfile', 'r', (err, fd) => {
      if (err) throw err;

      try {
        readMyData(fd);
      } finally {

```

```

        close(fd, (err) => {
            if (err) throw err;
        });
    }
});
} else {
    console.error('myfile does not exist');
}
});

```

read (RECOMMENDED)

```

import { open, close } from 'fs';

open('myfile', 'r', (err, fd) => {
    if (err) {
        if (err.code === 'ENOENT') {
            console.error('myfile does not exist');
            return;
        }

        throw err;
    }

    try {
        readMyData(fd);
    } finally {
        close(fd, (err) => {
            if (err) throw err;
        });
    }
});

```

The "not recommended" examples above check for existence and then use the file; the "recommended" examples are better because they use the file directly and handle the error, if any.

In general, check for the existence of a file only if the file won't be used directly, for example when its existence is a signal from another process.

fs.fchmod(fd, mode, callback)

- `fd` `<integer>`
- `mode` `<string> | <integer>`
- `callback` `<Function>`
 - `err` `<Error>`

Sets the permissions on the file. No arguments other than a possible exception are given to the completion callback.

See the POSIX `fchmod(2)` documentation for more detail.

fs.fchown(fd, uid, gid, callback)

- `fd` `<integer>`

- `uid` `<integer>`
- `gid` `<integer>`
- `callback` `<Function>`
 - `err` `<Error>`

Sets the owner of the file. No arguments other than a possible exception are given to the completion callback.

See the POSIX `fchown(2)` documentation for more detail.

`fs.fdatasync(fd, callback)`

- `fd` `<integer>`
- `callback` `<Function>`
 - `err` `<Error>`

Forces all currently queued I/O operations associated with the file to the operating system's synchronized I/O completion state. Refer to the POSIX `fdatasync(2)` documentation for details. No arguments other than a possible exception are given to the completion callback.

`fs.fstat(fd[, options], callback)`

- `fd` `<integer>`
- `options` `<Object>`
 - `bigint` `<boolean>` Whether the numeric values in the returned `<fs.Stats>` object should be `bigint`. **Default:** `false`.
- `callback` `<Function>`
 - `err` `<Error>`
 - `stats` `<fs.Stats>`

Invokes the callback with the `<fs.Stats>` for the file descriptor.

See the POSIX `fstat(2)` documentation for more detail.

`fs.fsync(fd, callback)`

- `fd` `<integer>`
- `callback` `<Function>`
 - `err` `<Error>`

Request that all data for the open file descriptor is flushed to the storage device. The specific implementation is operating system and device specific. Refer to the POSIX `fsync(2)` documentation for more detail. No arguments other than a possible exception are given to the completion callback.

`fs.ftruncate(fd[, len], callback)`

- `fd` `<integer>`
- `len` `<integer>` **Default:** `0`
- `callback` `<Function>`
 - `err` `<Error>`

Truncates the file descriptor. No arguments other than a possible exception are given to the completion callback.

See the POSIX `ftruncate(2)` documentation for more detail.

If the file referred to by the file descriptor was larger than `len` bytes, only the first `len` bytes will be retained in the file.

For example, the following program retains only the first four bytes of the file:

```
import { open, close, ftruncate } from 'fs';

function closeFd(fd) {
  close(fd, (err) => {
    if (err) throw err;
  });
}

open('temp.txt', 'r+', (err, fd) => {
  if (err) throw err;

  try {
    ftruncate(fd, 4, (err) => {
      closeFd(fd);
      if (err) throw err;
    });
  } catch (err) {
    closeFd(fd);
    if (err) throw err;
  }
});
```

If the file previously was shorter than `len` bytes, it is extended, and the extended part is filled with null bytes (`'\0'`):

If `len` is negative then `0` will be used.

fs.futimes(fd, atime, mtime, callback)

- `fd` `<integer>`
- `atime` `<number>` | `<string>` | `<Date>`
- `mtime` `<number>` | `<string>` | `<Date>`
- `callback` `<Function>`
 - `err` `<Error>`

Change the file system timestamps of the object referenced by the supplied file descriptor. See `fs.utimes()`.

fs.lchmod(path, mode, callback)

- `path` `<string>` | `<Buffer>` | `<URL>`
- `mode` `<integer>`
- `callback` `<Function>`
 - `err` `<Error>` | `<AggregateError>`

Changes the permissions on a symbolic link. No arguments other than a possible exception are given to the completion callback.

This method is only implemented on macOS.

See the POSIX `lchmod(2)` documentation for more detail.

fs.lchown(path, uid, gid, callback)

- `path` `<string>` | `<Buffer>` | `<URL>`

- `uid` `<integer>`
- `gid` `<integer>`
- `callback` `<Function>`
 - `err` `<Error>`

Set the owner of the symbolic link. No arguments other than a possible exception are given to the completion callback.

See the POSIX `lchown(2)` documentation for more detail.

`fs.lutimes(path, atime, mtime, callback)`

- `path` `<string>` | `<Buffer>` | `<URL>`
- `atime` `<number>` | `<string>` | `<Date>`
- `mtime` `<number>` | `<string>` | `<Date>`
- `callback` `<Function>`
 - `err` `<Error>`

Changes the access and modification times of a file in the same way as `fs.utimes()`, with the difference that if the path refers to a symbolic link, then the link is not dereferenced: instead, the timestamps of the symbolic link itself are changed.

No arguments other than a possible exception are given to the completion callback.

`fs.link(existingPath, newPath, callback)`

- `existingPath` `<string>` | `<Buffer>` | `<URL>`
- `newPath` `<string>` | `<Buffer>` | `<URL>`
- `callback` `<Function>`
 - `err` `<Error>`

Creates a new link from the `existingPath` to the `newPath`. See the POSIX `link(2)` documentation for more detail. No arguments other than a possible exception are given to the completion callback.

`fs.lstat(path[, options], callback)`

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<Object>`
 - `bigint` `<boolean>` Whether the numeric values in the returned `<fs.Stats>` object should be `bigint`. **Default:** `false`.
- `callback` `<Function>`
 - `err` `<Error>`
 - `stats` `<fs.Stats>`

Retrieves the `<fs.Stats>` for the symbolic link referred to by the path. The callback gets two arguments (`err`, `stats`) where `stats` is a `<fs.Stats>` object. `lstat()` is identical to `stat()`, except that if `path` is a symbolic link, then the link itself is stat-ed, not the file that it refers to.

See the POSIX `lstat(2)` documentation for more details.

`fs.mkdir(path[, options], callback)`

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<Object>` | `<integer>`
 - `recursive` `<boolean>` **Default:** `false`
 - `mode` `<string>` | `<integer>` Not supported on Windows. **Default:** `0o777`.
- `callback` `<Function>`

- `err` `<Error>`

Asynchronously creates a directory.

The callback is given a possible exception and, if `recursive` is `true`, the first directory path created, `(err, [path])`. `path` can still be `undefined` when `recursive` is `true`, if no directory was created.

The optional `options` argument can be an integer specifying `mode` (permission and sticky bits), or an object with a `mode` property and a `recursive` property indicating whether parent directories should be created. Calling `fs.mkdir()` when `path` is a directory that exists results in an error only when `recursive` is false.

```
import { mkdir } from 'fs';

// Creates /tmp/a/apple, regardless of whether `/tmp` and /tmp/a exist.
mkdir('/tmp/a/apple', { recursive: true }, (err) => {
  if (err) throw err;
});
```

On Windows, using `fs.mkdir()` on the root directory even with recursion will result in an error:

```
import { mkdir } from 'fs';

mkdir('/', { recursive: true }, (err) => {
  // => [Error: EPERM: operation not permitted, mkdir 'C:\']
});
```

See the POSIX `mkdir(2)` documentation for more details.

`fs.mkdtemp(prefix[, options], callback)`

- `prefix` `<string>`
- `options` `<string> | <Object>`
 - `encoding` `<string>` **Default:** `'utf8'`
- `callback` `<Function>`
 - `err` `<Error>`
 - `directory` `<string>`

Creates a unique temporary directory.

Generates six random characters to be appended behind a required `prefix` to create a unique temporary directory. Due to platform inconsistencies, avoid trailing `X` characters in `prefix`. Some platforms, notably the BSDs, can return more than six random characters, and replace trailing `X` characters in `prefix` with random characters.

The created directory path is passed as a string to the callback's second parameter.

The optional `options` argument can be a string specifying an encoding, or an object with an `encoding` property specifying the character encoding to use.

```
import { mkdtemp } from 'fs';

mkdtemp(path.join(os.tmpdir(), 'foo-'), (err, directory) => {
  if (err) throw err;
});
```

```

    console.log(directory);
    // Prints: /tmp/foo-itXde2 or C:\Users\...\AppData\Local\Temp\foo-itXde2
  });

```

The `fs.mkdtemp()` method will append the six randomly selected characters directly to the `prefix` string. For instance, given a directory `/tmp`, if the intention is to create a temporary directory *within* `/tmp`, the `prefix` must end with a trailing platform-specific path separator (`require('path').sep`).

```

import { tmpdir } from 'os';
import { mkdtemp } from 'fs';

// The parent directory for the new temporary directory
const tmpDir = tmpdir();

// This method is *INCORRECT*:
mkdtemp(tmpDir, (err, directory) => {
  if (err) throw err;
  console.log(directory);
  // Will print something similar to `/tmpabc123`.
  // A new temporary directory is created at the file system root
  // rather than *within* the /tmp directory.
});

// This method is *CORRECT*:
import { sep } from 'path';
mkdtemp(`${tmpDir}${sep}`, (err, directory) => {
  if (err) throw err;
  console.log(directory);
  // Will print something similar to `/tmp/abc123`.
  // A new temporary directory is created within
  // the /tmp directory.
});

```

fs.open(path[, flags[, mode]], callback)

- `path` `<string>` | `<Buffer>` | `<URL>`
- `flags` `<string>` | `<number>` See [support of file system flags](#). Default: `'r'`.
- `mode` `<string>` | `<integer>` Default: `0o666` (readable and writable)
- `callback` `<Function>`
 - `err` `<Error>`
 - `fd` `<integer>`

Asynchronous file open. See the POSIX [open\(2\)](#) documentation for more details.

`mode` sets the file mode (permission and sticky bits), but only if the file was created. On Windows, only the write permission can be manipulated; see [fs.chmod\(\)](#).

The callback gets two arguments (`err`, `fd`).

Some characters (`<` `>` `:` `"` `/` `\` `|` `?` `*`) are reserved under Windows as documented by [Naming Files, Paths, and Namespaces](#). Under NTFS, if the filename contains a colon, Node.js will open a file system stream, as described by [this MSDN page](#).

Functions based on `fs.open()` exhibit this behavior as well: `fs.writeFile()`, `fs.readFile()`, etc.

`fs.opendir(path[, options], callback)`

- `path` `<string> | <Buffer> | <URL>`
- `options` `<Object>`
 - `encoding` `<string> | <null>` **Default:** `'utf8'`
 - `bufferSize` `<number>` Number of directory entries that are buffered internally when reading from the directory. Higher values lead to better performance but higher memory usage. **Default:** `32`
- `callback` `<Function>`
 - `err` `<Error>`
 - `dir` `<fs.Dir>`

Asynchronously open a directory. See the POSIX `opendir(3)` documentation for more details.

Creates an `<fs.Dir>`, which contains all further functions for reading from and cleaning up the directory.

The `encoding` option sets the encoding for the `path` while opening the directory and subsequent read operations.

`fs.read(fd, buffer, offset, length, position, callback)`

- `fd` `<integer>`
- `buffer` `<Buffer> | <TypedArray> | <DataView>` The buffer that the data will be written to. **Default:** `Buffer.alloc(16384)`
- `offset` `<integer>` The position in `buffer` to write the data to. **Default:** `0`
- `length` `<integer>` The number of bytes to read. **Default:** `buffer.byteLength`
- `position` `<integer> | <bigint>` Specifies where to begin reading from in the file. If `position` is `null` or `-1`, data will be read from the current file position, and the file position will be updated. If `position` is an integer, the file position will be unchanged.
- `callback` `<Function>`
 - `err` `<Error>`
 - `bytesRead` `<integer>`
 - `buffer` `<Buffer>`

Read data from the file specified by `fd`.

The callback is given the three arguments, `(err, bytesRead, buffer)`.

If the file is not modified concurrently, the end-of-file is reached when the number of bytes read is zero.

If this method is invoked as its `util.promisify()` ed version, it returns a promise for an `Object` with `bytesRead` and `buffer` properties.

`fs.read(fd, [options], callback)`

- `fd` `<integer>`
- `options` `<Object>`
 - `buffer` `<Buffer> | <TypedArray> | <DataView>` **Default:** `Buffer.alloc(16384)`
 - `offset` `<integer>` **Default:** `0`
 - `length` `<integer>` **Default:** `buffer.byteLength`
 - `position` `<integer> | <bigint>` **Default:** `null`
- `callback` `<Function>`
 - `err` `<Error>`
 - `bytesRead` `<integer>`

- `buffer` `<Buffer>`

Similar to the `fs.read()` function, this version takes an optional `options` object. If no `options` object is specified, it will default with the above values.

`fs.readdir(path[, options], callback)`

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<string>` | `<Object>`
 - `encoding` `<string>` **Default:** `'utf8'`
 - `withFileTypes` `<boolean>` **Default:** `false`
- `callback` `<Function>`
 - `err` `<Error>`
 - `files` `<string[]>` | `<Buffer[]>` | `<fs.Dirent[]>`

Reads the contents of a directory. The callback gets two arguments `(err, files)` where `files` is an array of the names of the files in the directory excluding `'.'` and `'..'`.

See the POSIX `readdir(3)` documentation for more details.

The optional `options` argument can be a string specifying an encoding, or an object with an `encoding` property specifying the character encoding to use for the filenames passed to the callback. If the `encoding` is set to `'buffer'`, the filenames returned will be passed as `<Buffer>` objects.

If `options.withFileTypes` is set to `true`, the `files` array will contain `<fs.Dirent>` objects.

`fs.readFile(path[, options], callback)`

- `path` `<string>` | `<Buffer>` | `<URL>` | `<integer>` filename or file descriptor
- `options` `<Object>` | `<string>`
 - `encoding` `<string>` | `<null>` **Default:** `null`
 - `flag` `<string>` See [support of file system flags](#). **Default:** `'r'`.
 - `signal` `<AbortSignal>` allows aborting an in-progress `readFile`
- `callback` `<Function>`
 - `err` `<Error>` | `<AggregateError>`
 - `data` `<string>` | `<Buffer>`

Asynchronously reads the entire contents of a file.

```
import { readFile } from 'fs';

readFile('/etc/passwd', (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

The callback is passed two arguments `(err, data)`, where `data` is the contents of the file.

If no encoding is specified, then the raw buffer is returned.

If `options` is a string, then it specifies the encoding:

```
import { readFile } from 'fs';

readFile('/etc/passwd', 'utf8', callback);
```

When the path is a directory, the behavior of `fs.readFile()` and `fs.readFileSync()` is platform-specific. On macOS, Linux, and Windows, an error will be returned. On FreeBSD, a representation of the directory's contents will be returned.

```
import { readFile } from 'fs';

// macOS, Linux, and Windows
readFile('<directory>', (err, data) => {
  // => [Error: EISDIR: illegal operation on a directory, read <directory>]
});

// FreeBSD
readFile('<directory>', (err, data) => {
  // => null, <data>
});
```

It is possible to abort an ongoing request using an `AbortSignal`. If a request is aborted the callback is called with an `AbortError`:

```
import { readFile } from 'fs';

const controller = new AbortController();
const signal = controller.signal;
readFile(fileInfo[0].name, { signal }, (err, buf) => {
  // ...
});
// When you want to abort the request
controller.abort();
```

The `fs.readFile()` function buffers the entire file. To minimize memory costs, when possible prefer streaming via `fs.createReadStream()`.

Aborting an ongoing request does not abort individual operating system requests but rather the internal buffering `fs.readFile` performs.

File descriptors

1. Any specified file descriptor has to support reading.
2. If a file descriptor is specified as the `path`, it will not be closed automatically.
3. The reading will begin at the current position. For example, if the file already had `'Hello world'` and six bytes are read with the file descriptor, the call to `fs.readFile()` with the same file descriptor, would give `'World'`, rather than `'Hello world'`.

Performance Considerations

The `fs.readFile()` method asynchronously reads the contents of a file into memory one chunk at a time, allowing the event loop to turn between each chunk. This allows the read operation to have less impact on other activity that may be using the underlying libuv thread pool but means that it will take longer to read a complete file into memory.

The additional read overhead can vary broadly on different systems and depends on the type of file being read. If the file type is not a regular file (a pipe for instance) and Node.js is unable to determine an actual file size, each read operation will load on 64 KB of data. For regular files,

each read will process 512 KB of data.

For applications that require as-fast-as-possible reading of file contents, it is better to use `fs.read()` directly and for application code to manage reading the full contents of the file itself.

The Node.js GitHub issue [#25741](#) provides more information and a detailed analysis on the performance of `fs.readFile()` for multiple file sizes in different Node.js versions.

`fs.readlink(path[, options], callback)`

- `path` `<string> | <Buffer> | <URL>`
- `options` `<string> | <Object>`
 - `encoding` `<string>` Default: `'utf8'`
- `callback` `<Function>`
 - `err` `<Error>`
 - `linkString` `<string> | <Buffer>`

Reads the contents of the symbolic link referred to by `path`. The callback gets two arguments (`err`, `linkString`).

See the POSIX `readlink(2)` documentation for more details.

The optional `options` argument can be a string specifying an encoding, or an object with an `encoding` property specifying the character encoding to use for the link path passed to the callback. If the `encoding` is set to `'buffer'`, the link path returned will be passed as a `<Buffer>` object.

`fs.readv(fd, buffers[, position], callback)`

- `fd` `<integer>`
- `buffers` `<ArrayBufferView[]>`
- `position` `<integer>`
- `callback` `<Function>`
 - `err` `<Error>`
 - `bytesRead` `<integer>`
 - `buffers` `<ArrayBufferView[]>`

Read from a file specified by `fd` and write to an array of `ArrayBufferView`s using `readv()`.

`position` is the offset from the beginning of the file from where data should be read. If `typeof position !== 'number'`, the data will be read from the current position.

The callback will be given three arguments: `err`, `bytesRead`, and `buffers`. `bytesRead` is how many bytes were read from the file.

If this method is invoked as its `util.promisify()` ed version, it returns a promise for an `Object` with `bytesRead` and `buffers` properties.

`fs.realpath(path[, options], callback)`

- `path` `<string> | <Buffer> | <URL>`
- `options` `<string> | <Object>`
 - `encoding` `<string>` Default: `'utf8'`
- `callback` `<Function>`
 - `err` `<Error>`
 - `resolvedPath` `<string> | <Buffer>`

Asynchronously computes the canonical pathname by resolving `.`, `..` and symbolic links.

A canonical pathname is not necessarily unique. Hard links and bind mounts can expose a file system entity through many pathnames.

This function behaves like `realpath(3)`, with some exceptions:

1. No case conversion is performed on case-insensitive file systems.
2. The maximum number of symbolic links is platform-independent and generally (much) higher than what the native `realpath(3)` implementation supports.

The `callback` gets two arguments `(err, resolvedPath)`. May use `process.cwd` to resolve relative paths.

Only paths that can be converted to UTF8 strings are supported.

The optional `options` argument can be a string specifying an encoding, or an object with an `encoding` property specifying the character encoding to use for the path passed to the callback. If the `encoding` is set to `'buffer'`, the path returned will be passed as a `<Buffer>` object.

If `path` resolves to a socket or a pipe, the function will return a system dependent name for that object.

`fs.realpath.native(path[, options], callback)`

- `path` `<string> | <Buffer> | <URL>`
- `options` `<string> | <Object>`
 - `encoding` `<string>` **Default:** `'utf8'`
- `callback` `<Function>`
 - `err` `<Error>`
 - `resolvedPath` `<string> | <Buffer>`

Asynchronous `realpath(3)`.

The `callback` gets two arguments `(err, resolvedPath)`.

Only paths that can be converted to UTF8 strings are supported.

The optional `options` argument can be a string specifying an encoding, or an object with an `encoding` property specifying the character encoding to use for the path passed to the callback. If the `encoding` is set to `'buffer'`, the path returned will be passed as a `<Buffer>` object.

On Linux, when Node.js is linked against musl libc, the procfs file system must be mounted on `/proc` in order for this function to work. Glibc does not have this restriction.

`fs.rename(oldPath, newPath, callback)`

- `oldPath` `<string> | <Buffer> | <URL>`
- `newPath` `<string> | <Buffer> | <URL>`
- `callback` `<Function>`
 - `err` `<Error>`

Asynchronously rename file at `oldPath` to the pathname provided as `newPath`. In the case that `newPath` already exists, it will be overwritten. If there is a directory at `newPath`, an error will be raised instead. No arguments other than a possible exception are given to the completion callback.

See also: `rename(2)`.

```
import { rename } from 'fs';
```

```
rename('oldFile.txt', 'newFile.txt', (err) => {
  if (err) throw err;
  console.log('Rename complete!');
});
```

fs.rmdir(path[, options], callback)

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<Object>`
 - `maxRetries` `<integer>` If an `EBUSY`, `EMFILE`, `ENFILE`, `ENOTEMPTY`, or `EPERM` error is encountered, Node.js retries the operation with a linear backoff wait of `retryDelay` milliseconds longer on each try. This option represents the number of retries. This option is ignored if the `recursive` option is not `true`. **Default:** `0`.
 - `recursive` `<boolean>` If `true`, perform a recursive directory removal. In recursive mode, operations are retried on failure. **Default:** `false`. **Deprecated.**
 - `retryDelay` `<integer>` The amount of time in milliseconds to wait between retries. This option is ignored if the `recursive` option is not `true`. **Default:** `100`.
- `callback` `<Function>`
 - `err` `<Error>`

Asynchronous `rmdir(2)`. No arguments other than a possible exception are given to the completion callback.

Using `fs.rmdir()` on a file (not a directory) results in an `ENOENT` error on Windows and an `ENOTDIR` error on POSIX.

To get a behavior similar to the `rm -rf` Unix command, use `fs.rm()` with options `{ recursive: true, force: true }`.

fs.rm(path[, options], callback)

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<Object>`
 - `force` `<boolean>` When `true`, exceptions will be ignored if `path` does not exist. **Default:** `false`.
 - `maxRetries` `<integer>` If an `EBUSY`, `EMFILE`, `ENFILE`, `ENOTEMPTY`, or `EPERM` error is encountered, Node.js will retry the operation with a linear backoff wait of `retryDelay` milliseconds longer on each try. This option represents the number of retries. This option is ignored if the `recursive` option is not `true`. **Default:** `0`.
 - `recursive` `<boolean>` If `true`, perform a recursive removal. In recursive mode operations are retried on failure. **Default:** `false`.
 - `retryDelay` `<integer>` The amount of time in milliseconds to wait between retries. This option is ignored if the `recursive` option is not `true`. **Default:** `100`.
- `callback` `<Function>`
 - `err` `<Error>`

Asynchronously removes files and directories (modeled on the standard POSIX `rm` utility). No arguments other than a possible exception are given to the completion callback.

fs.stat(path[, options], callback)

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<Object>`
 - `bigint` `<boolean>` Whether the numeric values in the returned `<fs.Stats>` object should be `bigint`. **Default:** `false`.
- `callback` `<Function>`
 - `err` `<Error>`
 - `stats` `<fs.Stats>`

Asynchronous `stat(2)`. The callback gets two arguments `(err, stats)` where `stats` is an `<fs.Stats>` object.

In case of an error, the `err.code` will be one of [Common System Errors](#).

Using `fs.stat()` to check for the existence of a file before calling `fs.open()`, `fs.readFile()` or `fs.writeFile()` is not recommended. Instead, user code should open/read/write the file directly and handle the error raised if the file is not available.

To check if a file exists without manipulating it afterwards, `fs.access()` is recommended.

For example, given the following directory structure:

```
- txtDir
-- file.txt
- app.js
```

The next program will check for the stats of the given paths:

```
import { stat } from 'fs';

const pathsToCheck = ['./txtDir', './txtDir/file.txt'];

for (let i = 0; i < pathsToCheck.length; i++) {
  stat(pathsToCheck[i], (err, stats) => {
    console.log(stats.isDirectory());
    console.log(stats);
  });
}
```

The resulting output will resemble:

```
true
Stats {
  dev: 16777220,
  mode: 16877,
  nlink: 3,
  uid: 501,
  gid: 20,
  rdev: 0,
  blksize: 4096,
  ino: 14214262,
  size: 96,
  blocks: 0,
  atimeMs: 1561174653071.963,
  mtimeMs: 1561174614583.3518,
  ctimeMs: 1561174626623.5366,
  birthtimeMs: 1561174126937.2893,
  atime: 2019-06-22T03:37:33.072Z,
  mtime: 2019-06-22T03:36:54.583Z,
  ctime: 2019-06-22T03:37:06.624Z,
  birthtime: 2019-06-22T03:28:46.937Z
}
false
Stats {
```

```
dev: 16777220,
mode: 33188,
nlink: 1,
uid: 501,
gid: 20,
rdev: 0,
blksize: 4096,
ino: 14214074,
size: 8,
blocks: 8,
atimeMs: 1561174616618.8555,
mtimeMs: 1561174614584,
ctimeMs: 1561174614583.8145,
birthtimeMs: 1561174007710.7478,
atime: 2019-06-22T03:36:56.619Z,
mtime: 2019-06-22T03:36:54.584Z,
ctime: 2019-06-22T03:36:54.584Z,
birthtime: 2019-06-22T03:26:47.711Z
}
```

fs.symlink(target, path[, type], callback)

- **target** `<string> | <Buffer> | <URL>`
- **path** `<string> | <Buffer> | <URL>`
- **type** `<string>`
- **callback** `<Function>`
 - **err** `<Error>`

Creates the link called **path** pointing to **target**. No arguments other than a possible exception are given to the completion callback.

See the POSIX [symlink\(2\)](#) documentation for more details.

The **type** argument is only available on Windows and ignored on other platforms. It can be set to `'dir'`, `'file'`, or `'junction'`. If the **type** argument is not set, Node.js will autodetect **target** type and use `'file'` or `'dir'`. If the **target** does not exist, `'file'` will be used. Windows junction points require the destination path to be absolute. When using `'junction'`, the **target** argument will automatically be normalized to absolute path.

Relative targets are relative to the link's parent directory.

```
import { symlink } from 'fs';

symlink('./mew', './example/mewtwo', callback);
```

The above example creates a symbolic link `mewtwo` in the `example` which points to `mew` in the same directory:

```
$ tree example/
example/
├─ mew
└─ mewtwo -> ./mew
```

fs.truncate(path[, len], callback)

- `path` `<string>` | `<Buffer>` | `<URL>`
- `len` `<integer>` **Default:** 0
- `callback` `<Function>`
 - `err` `<Error>` | `<AggregateError>`

Truncates the file. No arguments other than a possible exception are given to the completion callback. A file descriptor can also be passed as the first argument. In this case, `fs.ftruncate()` is called.

```
import { truncate } from 'fs';
// Assuming that 'path/file.txt' is a regular file.
truncate('path/file.txt', (err) => {
  if (err) throw err;
  console.log('path/file.txt was truncated');
});const { truncate } = require('fs');
// Assuming that 'path/file.txt' is a regular file.
truncate('path/file.txt', (err) => {
  if (err) throw err;
  console.log('path/file.txt was truncated');
});
```

Passing a file descriptor is deprecated and may result in an error being thrown in the future.

See the POSIX `truncate(2)` documentation for more details.

fs.unlink(path, callback)

- `path` `<string>` | `<Buffer>` | `<URL>`
- `callback` `<Function>`
 - `err` `<Error>`

Asynchronously removes a file or symbolic link. No arguments other than a possible exception are given to the completion callback.

```
import { unlink } from 'fs';
// Assuming that 'path/file.txt' is a regular file.
unlink('path/file.txt', (err) => {
  if (err) throw err;
  console.log('path/file.txt was deleted');
});
```

`fs.unlink()` will not work on a directory, empty or otherwise. To remove a directory, use `fs.rmdir()`.

See the POSIX `unlink(2)` documentation for more details.

fs.unwatchFile(filename[, listener])

- `filename` `<string>` | `<Buffer>` | `<URL>`
- `listener` `<Function>` Optional, a listener previously attached using `fs.watchFile()`

Stop watching for changes on `filename`. If `listener` is specified, only that particular listener is removed. Otherwise, *all* listeners are removed, effectively stopping watching of `filename`.

Calling `fs.unwatchFile()` with a filename that is not being watched is a no-op, not an error.

Using `fs.watch()` is more efficient than `fs.watchFile()` and `fs.unwatchFile()`. `fs.watch()` should be used instead of `fs.watchFile()` and `fs.unwatchFile()` when possible.

`fs.utimes(path, atime, mtime, callback)`

- `path` `<string>` | `<Buffer>` | `<URL>`
- `atime` `<number>` | `<string>` | `<Date>`
- `mtime` `<number>` | `<string>` | `<Date>`
- `callback` `<Function>`
 - `err` `<Error>`

Change the file system timestamps of the object referenced by `path`.

The `atime` and `mtime` arguments follow these rules:

- Values can be either numbers representing Unix epoch time in seconds, `Date`s, or a numeric string like `'123456789.0'`.
- If the value can not be converted to a number, or is `NaN`, `Infinity` or `-Infinity`, an `Error` will be thrown.

`fs.watch(filename[, options][, listener])`

- `filename` `<string>` | `<Buffer>` | `<URL>`
- `options` `<string>` | `<Object>`
 - `persistent` `<boolean>` Indicates whether the process should continue to run as long as files are being watched. **Default:** `true`.
 - `recursive` `<boolean>` Indicates whether all subdirectories should be watched, or only the current directory. This applies when a directory is specified, and only on supported platforms (See [caveats](#)). **Default:** `false`.
 - `encoding` `<string>` Specifies the character encoding to be used for the filename passed to the listener. **Default:** `'utf8'`.
 - `signal` `<AbortSignal>` allows closing the watcher with an `AbortSignal`.
- `listener` `<Function>` | `<undefined>` **Default:** `undefined`
 - `eventType` `<string>`
 - `filename` `<string>` | `<Buffer>`
- Returns: `<fs.FSWatcher>`

Watch for changes on `filename`, where `filename` is either a file or a directory.

The second argument is optional. If `options` is provided as a string, it specifies the `encoding`. Otherwise `options` should be passed as an object.

The listener callback gets two arguments (`eventType`, `filename`). `eventType` is either `'rename'` or `'change'`, and `filename` is the name of the file which triggered the event.

On most platforms, `'rename'` is emitted whenever a filename appears or disappears in the directory.

The listener callback is attached to the `'change'` event fired by `<fs.FSWatcher>`, but it is not the same thing as the `'change'` value of `eventType`.

If a `signal` is passed, aborting the corresponding `AbortController` will close the returned `<fs.FSWatcher>`.

Caveats

The `fs.watch` API is not 100% consistent across platforms, and is unavailable in some situations.

The recursive option is only supported on macOS and Windows. An `ERR_FEATURE_UNAVAILABLE_ON_PLATFORM` exception will be thrown when the option is used on a platform that does not support it.

On Windows, no events will be emitted if the watched directory is moved or renamed. An `EPERM` error is reported when the watched directory is deleted.

Availability

This feature depends on the underlying operating system providing a way to be notified of filesystem changes.

- On Linux systems, this uses `inotify(7)`.
- On BSD systems, this uses `kqueue(2)`.
- On macOS, this uses `kqueue(2)` for files and `FSEvents` for directories.
- On SunOS systems (including Solaris and SmartOS), this uses `event ports`.
- On Windows systems, this feature depends on `ReadDirectoryChangesW`.
- On AIX systems, this feature depends on `AHAFS`, which must be enabled.
- On IBM i systems, this feature is not supported.

If the underlying functionality is not available for some reason, then `fs.watch()` will not be able to function and may throw an exception. For example, watching files or directories can be unreliable, and in some cases impossible, on network file systems (NFS, SMB, etc) or host file systems when using virtualization software such as Vagrant or Docker.

It is still possible to use `fs.watchFile()`, which uses stat polling, but this method is slower and less reliable.

Inodes

On Linux and macOS systems, `fs.watch()` resolves the path to an `inode` and watches the inode. If the watched path is deleted and recreated, it is assigned a new inode. The watch will emit an event for the delete but will continue watching the *original* inode. Events for the new inode will not be emitted. This is expected behavior.

AIX files retain the same inode for the lifetime of a file. Saving and closing a watched file on AIX will result in two notifications (one for adding new content, and one for truncation).

Filename argument

Providing `filename` argument in the callback is only supported on Linux, macOS, Windows, and AIX. Even on supported platforms, `filename` is not always guaranteed to be provided. Therefore, don't assume that `filename` argument is always provided in the callback, and have some fallback logic if it is `null`.

```
import { watch } from 'fs';
watch('somedir', (eventType, filename) => {
  console.log(`event type is: ${eventType}`);
  if (filename) {
    console.log(`filename provided: ${filename}`);
  } else {
    console.log('filename not provided');
  }
});
```

`fs.watchFile(filename[, options], listener)`

- `filename` `<string> | <Buffer> | <URL>`
- `options` `<Object>`

- `bigint` `<boolean>` Default: `false`
- `persistent` `<boolean>` Default: `true`
- `interval` `<integer>` Default: `5007`
- `listener` `<Function>`
 - `current` `<fs.Stats>`
 - `previous` `<fs.Stats>`
- Returns: `<fs.StatWatcher>`

Watch for changes on `filename`. The callback `listener` will be called each time the file is accessed.

The `options` argument may be omitted. If provided, it should be an object. The `options` object may contain a boolean named `persistent` that indicates whether the process should continue to run as long as files are being watched. The `options` object may specify an `interval` property indicating how often the target should be polled in milliseconds.

The `listener` gets two arguments the current stat object and the previous stat object:

```
import { watchFile } from 'fs';

watchFile('message.text', (curr, prev) => {
  console.log(`the current mtime is: ${curr.mtime}`);
  console.log(`the previous mtime was: ${prev.mtime}`);
});
```

These stat objects are instances of `fs.Stat`. If the `bigint` option is `true`, the numeric values in these objects are specified as `BigInt` s.

To be notified when the file was modified, not just accessed, it is necessary to compare `curr.mtime` and `prev.mtime`.

When an `fs.watchFile` operation results in an `ENOENT` error, it will invoke the listener once, with all the fields zeroed (or, for dates, the Unix Epoch). If the file is created later on, the listener will be called again, with the latest stat objects. This is a change in functionality since v0.10.

Using `fs.watch()` is more efficient than `fs.watchFile` and `fs.unwatchFile`. `fs.watch` should be used instead of `fs.watchFile` and `fs.unwatchFile` when possible.

When a file being watched by `fs.watchFile()` disappears and reappears, then the contents of `previous` in the second callback event (the file's reappearance) will be the same as the contents of `previous` in the first callback event (its disappearance).

This happens when:

- the file is deleted, followed by a restore
- the file is renamed and then renamed a second time back to its original name

`fs.write(fd, buffer[, offset[, length[, position]]], callback)`

- `fd` `<integer>`
- `buffer` `<Buffer>` | `<TypedArray>` | `<DataView>` | `<string>` | `<Object>`
- `offset` `<integer>`
- `length` `<integer>`
- `position` `<integer>`
- `callback` `<Function>`
 - `err` `<Error>`
 - `bytesWritten` `<integer>`

- `buffer` `<Buffer>` | `<TypedArray>` | `<DataView>`

Write `buffer` to the file specified by `fd`. If `buffer` is a normal object, it must have an own `toString` function property.

`offset` determines the part of the buffer to be written, and `length` is an integer specifying the number of bytes to write.

`position` refers to the offset from the beginning of the file where this data should be written. If `typeof position !== 'number'`, the data will be written at the current position. See `pwrite(2)`.

The callback will be given three arguments `(err, bytesWritten, buffer)` where `bytesWritten` specifies how many *bytes* were written from `buffer`.

If this method is invoked as its `util.promisify()` ed version, it returns a promise for an `Object` with `bytesWritten` and `buffer` properties.

It is unsafe to use `fs.write()` multiple times on the same file without waiting for the callback. For this scenario, `fs.createWriteStream()` is recommended.

On Linux, positional writes don't work when the file is opened in append mode. The kernel ignores the position argument and always appends the data to the end of the file.

`fs.write(fd, string[, position[, encoding]], callback)`

- `fd` `<integer>`
- `string` `<string>` | `<Object>`
- `position` `<integer>`
- `encoding` `<string>` Default: `'utf8'`
- `callback` `<Function>`
 - `err` `<Error>`
 - `written` `<integer>`
 - `string` `<string>`

Write `string` to the file specified by `fd`. If `string` is not a string, or an object with an own `toString` function property, then an exception is thrown.

`position` refers to the offset from the beginning of the file where this data should be written. If `typeof position !== 'number'` the data will be written at the current position. See `pwrite(2)`.

`encoding` is the expected string encoding.

The callback will receive the arguments `(err, written, string)` where `written` specifies how many *bytes* the passed string required to be written. Bytes written is not necessarily the same as string characters written. See `Buffer.byteLength`.

It is unsafe to use `fs.write()` multiple times on the same file without waiting for the callback. For this scenario, `fs.createWriteStream()` is recommended.

On Linux, positional writes don't work when the file is opened in append mode. The kernel ignores the position argument and always appends the data to the end of the file.

On Windows, if the file descriptor is connected to the console (e.g. `fd == 1` or `stdout`) a string containing non-ASCII characters will not be rendered properly by default, regardless of the encoding used. It is possible to configure the console to render UTF-8 properly by changing the active codepage with the `chcp 65001` command. See the `chcp` docs for more details.

`fs.writeFile(file, data[, options], callback)`

- `file` `<string>` | `<Buffer>` | `<URL>` | `<integer>` filename or file descriptor

- `data` `<string>` | `<Buffer>` | `<TypedArray>` | `<DataView>` | `<Object>`
- `options` `<Object>` | `<string>`
 - `encoding` `<string>` | `<null>` **Default:** `'utf8'`
 - `mode` `<integer>` **Default:** `0o666`
 - `flag` `<string>` See [support of file system flags](#) . **Default:** `'w'` .
 - `signal` `<AbortSignal>` allows aborting an in-progress `writeFile`
- `callback` `<Function>`
 - `err` `<Error>` | `<AggregateError>`

When `file` is a filename, asynchronously writes data to the file, replacing the file if it already exists. `data` can be a string or a buffer.

When `file` is a file descriptor, the behavior is similar to calling `fs.write()` directly (which is recommended). See the notes below on using a file descriptor.

The `encoding` option is ignored if `data` is a buffer.

If `data` is a plain object, it must have an own (not inherited) `toString` function property.

```
import { writeFile } from 'fs';
import { Buffer } from 'buffer';

const data = new Uint8Array(Buffer.from('Hello Node.js'));
writeFile('message.txt', data, (err) => {
  if (err) throw err;
  console.log('The file has been saved!');
});
```

If `options` is a string, then it specifies the encoding:

```
import { writeFile } from 'fs';

writeFile('message.txt', 'Hello Node.js', 'utf8', callback);
```

It is unsafe to use `fs.writeFile()` multiple times on the same file without waiting for the callback. For this scenario, `fs.createWriteStream()` is recommended.

Similarly to `fs.readFile` - `fs.writeFile` is a convenience method that performs multiple `write` calls internally to write the buffer passed to it. For performance sensitive code consider using `fs.createWriteStream()` .

It is possible to use an `<AbortSignal>` to cancel an `fs.writeFile()` . Cancellation is "best effort", and some amount of data is likely still to be written.

```
import { writeFile } from 'fs';
import { Buffer } from 'buffer';

const controller = new AbortController();
const { signal } = controller;
const data = new Uint8Array(Buffer.from('Hello Node.js'));
writeFile('message.txt', data, { signal }, (err) => {
  // When a request is aborted - the callback is called with an AbortError
});
```

```
// When the request should be aborted
controller.abort();
```

Aborting an ongoing request does not abort individual operating system requests but rather the internal buffering `fs.writeFile` performs.

Using `fs.writeFile()` with file descriptors

When `file` is a file descriptor, the behavior is almost identical to directly calling `fs.write()` like:

```
import { write } from 'fs';
import { Buffer } from 'buffer';

write(fd, Buffer.from(data, options.encoding), callback);
```

The difference from directly calling `fs.write()` is that under some unusual conditions, `fs.write()` might write only part of the buffer and need to be retried to write the remaining data, whereas `fs.writeFile()` retries until the data is entirely written (or an error occurs).

The implications of this are a common source of confusion. In the file descriptor case, the file is not replaced! The data is not necessarily written to the beginning of the file, and the file's original data may remain before and/or after the newly written data.

For example, if `fs.writeFile()` is called twice in a row, first to write the string `'Hello'`, then to write the string `', world'`, the file would contain `'Hello, world'`, and might contain some of the file's original data (depending on the size of the original file, and the position of the file descriptor). If a file name had been used instead of a descriptor, the file would be guaranteed to contain only `', world'`.

`fs.writev(fd, buffers[, position], callback)`

- `fd` `<integer>`
- `buffers` `<ArrayBufferView[]>`
- `position` `<integer>`
- `callback` `<Function>`
 - `err` `<Error>`
 - `bytesWritten` `<integer>`
 - `buffers` `<ArrayBufferView[]>`

Write an array of `ArrayBufferView`s to the file specified by `fd` using `writev()`.

`position` is the offset from the beginning of the file where this data should be written. If `typeof position !== 'number'`, the data will be written at the current position.

The callback will be given three arguments: `err`, `bytesWritten`, and `buffers`. `bytesWritten` is how many bytes were written from `buffers`.

If this method is `util.promisify()` ed, it returns a promise for an `Object` with `bytesWritten` and `buffers` properties.

It is unsafe to use `fs.writev()` multiple times on the same file without waiting for the callback. For this scenario, use `fs.createWriteStream()`.

On Linux, positional writes don't work when the file is opened in append mode. The kernel ignores the position argument and always appends the data to the end of the file.

Synchronous API

The synchronous APIs perform all operations synchronously, blocking the event loop until the operation completes or fails.

fs.accessSync(path[, mode])

- `path` `<string>` | `<Buffer>` | `<URL>`
- `mode` `<integer>` **Default:** `fs.constants.F_OK`

Synchronously tests a user's permissions for the file or directory specified by `path`. The `mode` argument is an optional integer that specifies the accessibility checks to be performed. Check [File access constants](#) for possible values of `mode`. It is possible to create a mask consisting of the bitwise OR of two or more values (e.g. `fs.constants.W_OK | fs.constants.R_OK`).

If any of the accessibility checks fail, an `Error` will be thrown. Otherwise, the method will return `undefined`.

```
import { accessSync, constants } from 'fs';

try {
  accessSync('etc/passwd', constants.R_OK | constants.W_OK);
  console.log('can read/write');
} catch (err) {
  console.error('no access!');
}
```

fs.appendFileSync(path, data[, options])

- `path` `<string>` | `<Buffer>` | `<URL>` | `<number>` filename or file descriptor
- `data` `<string>` | `<Buffer>`
- `options` `<Object>` | `<string>`
 - `encoding` `<string>` | `<null>` **Default:** `'utf8'`
 - `mode` `<integer>` **Default:** `0o666`
 - `flag` `<string>` See [support of file system flags](#). **Default:** `'a'`.

Synchronously append data to a file, creating the file if it does not yet exist. `data` can be a string or a `<Buffer>`.

```
import { appendFileSync } from 'fs';

try {
  appendFileSync('message.txt', 'data to append');
  console.log('The "data to append" was appended to file!');
} catch (err) {
  /* Handle the error */
}
```

If `options` is a string, then it specifies the encoding:

```
import { appendFileSync } from 'fs';

appendFileSync('message.txt', 'data to append', 'utf8');
```

The `path` may be specified as a numeric file descriptor that has been opened for appending (using `fs.open()` or `fs.openSync()`). The file descriptor will not be closed automatically.

```
import { openSync, closeSync, appendFileSync } from 'fs';

let fd;

try {
  fd = openSync('message.txt', 'a');
  appendFileSync(fd, 'data to append', 'utf8');
} catch (err) {
  /* Handle the error */
} finally {
  if (fd !== undefined)
    closeSync(fd);
}
```

fs.chmodSync(path, mode)

- `path` `<string> | <Buffer> | <URL>`
- `mode` `<string> | <integer>`

For detailed information, see the documentation of the asynchronous version of this API: `fs.chmod()`.

See the POSIX `chmod(2)` documentation for more detail.

fs.chownSync(path, uid, gid)

- `path` `<string> | <Buffer> | <URL>`
- `uid` `<integer>`
- `gid` `<integer>`

Synchronously changes owner and group of a file. Returns `undefined`. This is the synchronous version of `fs.chown()`.

See the POSIX `chown(2)` documentation for more detail.

fs.closeSync(fd)

- `fd` `<integer>`

Closes the file descriptor. Returns `undefined`.

Calling `fs.closeSync()` on any file descriptor (`fd`) that is currently in use through any other `fs` operation may lead to undefined behavior.

See the POSIX `close(2)` documentation for more detail.

fs.copyFileSync(src, dest[, mode])

- `src` `<string> | <Buffer> | <URL>` source filename to copy
- `dest` `<string> | <Buffer> | <URL>` destination filename of the copy operation
- `mode` `<integer>` modifiers for copy operation. **Default:** `0`.

Synchronously copies `src` to `dest`. By default, `dest` is overwritten if it already exists. Returns `undefined`. Node.js makes no guarantees about the atomicity of the copy operation. If an error occurs after the destination file has been opened for writing, Node.js will attempt to remove the destination.

`mode` is an optional integer that specifies the behavior of the copy operation. It is possible to create a mask consisting of the bitwise OR of two or more values (e.g. `fs.constants.COPYFILE_EXCL | fs.constants.COPYFILE_FICLONE`).

- `fs.constants.COPYFILE_EXCL` : The copy operation will fail if `dest` already exists.
- `fs.constants.COPYFILE_FICLONE` : The copy operation will attempt to create a copy-on-write reflink. If the platform does not support copy-on-write, then a fallback copy mechanism is used.
- `fs.constants.COPYFILE_FICLONE_FORCE` : The copy operation will attempt to create a copy-on-write reflink. If the platform does not support copy-on-write, then the operation will fail.

```
import { copyFileSync, constants } from 'fs';

// destination.txt will be created or overwritten by default.
copyFileSync('source.txt', 'destination.txt');
console.log('source.txt was copied to destination.txt');

// By using COPYFILE_EXCL, the operation will fail if destination.txt exists.
copyFileSync('source.txt', 'destination.txt', constants.COPYFILE_EXCL);
```

fs.cpSync(src, dest[, options])

Stability: 1 - Experimental

- `src` `<string> | <URL>` source path to copy.
- `dest` `<string> | <URL>` destination path to copy to.
- `options` `<Object>`
 - `dereference` `<boolean>` dereference symlinks. **Default:** `false`.
 - `errorOnExist` `<boolean>` when `force` is `false`, and the destination exists, throw an error. **Default:** `false`.
 - `filter` `<Function>` Function to filter copied files/directories. Return `true` to copy the item, `false` to ignore it. **Default:** `undefined`
 - `force` `<boolean>` overwrite existing file or directory. The copy operation will ignore errors if you set this to `false` and the destination exists. Use the `errorOnExist` option to change this behavior. **Default:** `true`.
 - `preserveTimestamps` `<boolean>` When `true` timestamps from `src` will be preserved. **Default:** `false`.
 - `recursive` `<boolean>` copy directories recursively **Default:** `false`

Synchronously copies the entire directory structure from `src` to `dest`, including subdirectories and files.

When copying a directory to another directory, globs are not supported and behavior is similar to `cp dir1/ dir2/`.

fs.existsSync(path)

- `path` `<string> | <Buffer> | <URL>`
- Returns: `<boolean>`

Returns `true` if the path exists, `false` otherwise.

For detailed information, see the documentation of the asynchronous version of this API: `fs.exists()`.

`fs.exists()` is deprecated, but `fs.existsSync()` is not. The `callback` parameter to `fs.exists()` accepts parameters that are inconsistent with other Node.js callbacks. `fs.existsSync()` does not use a callback.

```
import { existsSync } from 'fs';

if (existsSync('/etc/passwd'))
  console.log('The path exists.');
```

fs.fchmodSync(fd, mode)

- `fd` `<integer>`
- `mode` `<string> | <integer>`

Sets the permissions on the file. Returns `undefined`.

See the POSIX `fchmod(2)` documentation for more detail.

fs.fchownSync(fd, uid, gid)

- `fd` `<integer>`
- `uid` `<integer>` The file's new owner's user id.
- `gid` `<integer>` The file's new group's group id.

Sets the owner of the file. Returns `undefined`.

See the POSIX `fchown(2)` documentation for more detail.

fs.fdatasyncSync(fd)

- `fd` `<integer>`

Forces all currently queued I/O operations associated with the file to the operating system's synchronized I/O completion state. Refer to the POSIX `fdatasync(2)` documentation for details. Returns `undefined`.

fs.fstatSync(fd[, options])

- `fd` `<integer>`
- `options` `<Object>`
 - `bigint` `<boolean>` Whether the numeric values in the returned `<fs.Stats>` object should be `bigint`. **Default:** `false`.
- Returns: `<fs.Stats>`

Retrieves the `<fs.Stats>` for the file descriptor.

See the POSIX `fstat(2)` documentation for more detail.

fs.fsyncSync(fd)

- `fd` `<integer>`

Request that all data for the open file descriptor is flushed to the storage device. The specific implementation is operating system and device specific. Refer to the POSIX `fsync(2)` documentation for more detail. Returns `undefined`.

fs.ftruncateSync(fd[, len])

- `fd` `<integer>`
- `len` `<integer>` **Default:** `0`

Truncates the file descriptor. Returns `undefined`.

For detailed information, see the documentation of the asynchronous version of this API: `fs.ftruncate()`.

`fs.futimesSync(fd, atime, mtime)`

- `fd` `<integer>`
- `atime` `<number>` | `<string>` | `<Date>`
- `mtime` `<number>` | `<string>` | `<Date>`

Synchronous version of `fs.futimes()`. Returns `undefined`.

`fs.lchmodSync(path, mode)`

- `path` `<string>` | `<Buffer>` | `<URL>`
- `mode` `<integer>`

Changes the permissions on a symbolic link. Returns `undefined`.

This method is only implemented on macOS.

See the POSIX `lchmod(2)` documentation for more detail.

`fs.lchownSync(path, uid, gid)`

- `path` `<string>` | `<Buffer>` | `<URL>`
- `uid` `<integer>` The file's new owner's user id.
- `gid` `<integer>` The file's new group's group id.

Set the owner for the path. Returns `undefined`.

See the POSIX `lchown(2)` documentation for more details.

`fs.lutimesSync(path, atime, mtime)`

- `path` `<string>` | `<Buffer>` | `<URL>`
- `atime` `<number>` | `<string>` | `<Date>`
- `mtime` `<number>` | `<string>` | `<Date>`

Change the file system timestamps of the symbolic link referenced by `path`. Returns `undefined`, or throws an exception when parameters are incorrect or the operation fails. This is the synchronous version of `fs.lutimes()`.

`fs.linkSync(existingPath, newPath)`

- `existingPath` `<string>` | `<Buffer>` | `<URL>`
- `newPath` `<string>` | `<Buffer>` | `<URL>`

Creates a new link from the `existingPath` to the `newPath`. See the POSIX `link(2)` documentation for more detail. Returns `undefined`.

`fs.lstatSync(path[, options])`

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<Object>`
 - `bigint` `<boolean>` Whether the numeric values in the returned `<fs.Stats>` object should be `bigint`. **Default:** `false`.
 - `throwIfNoEntry` `<boolean>` Whether an exception will be thrown if no file system entry exists, rather than returning `undefined`. **Default:** `true`.
- Returns: `<fs.Stats>`

Retrieves the `<fs.Stats>` for the symbolic link referred to by `path`.

See the POSIX `lstat(2)` documentation for more details.

`fs.mkdirSync(path[, options])`

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<Object>` | `<integer>`
 - `recursive` `<boolean>` **Default:** `false`
 - `mode` `<string>` | `<integer>` Not supported on Windows. **Default:** `0o777`.
- Returns: `<string>` | `<undefined>`

Synchronously creates a directory. Returns `undefined`, or if `recursive` is `true`, the first directory path created. This is the synchronous version of `fs.mkdir()`.

See the POSIX `mkdir(2)` documentation for more details.

`fs.mkdtempSync(prefix[, options])`

- `prefix` `<string>`
- `options` `<string>` | `<Object>`
 - `encoding` `<string>` **Default:** `'utf8'`
- Returns: `<string>`

Returns the created directory path.

For detailed information, see the documentation of the asynchronous version of this API: `fs.mkdtemp()`.

The optional `options` argument can be a string specifying an encoding, or an object with an `encoding` property specifying the character encoding to use.

`fs.opendirSync(path[, options])`

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<Object>`
 - `encoding` `<string>` | `<null>` **Default:** `'utf8'`
 - `bufferSize` `<number>` Number of directory entries that are buffered internally when reading from the directory. Higher values lead to better performance but higher memory usage. **Default:** `32`
- Returns: `<fs.Dir>`

Synchronously open a directory. See `opendir(3)`.

Creates an `<fs.Dir>`, which contains all further functions for reading from and cleaning up the directory.

The `encoding` option sets the encoding for the `path` while opening the directory and subsequent read operations.

`fs.openSync(path[, flags[, mode]])`

- `path` `<string>` | `<Buffer>` | `<URL>`
- `flags` `<string>` | `<number>` **Default:** `'r'`. See [support of file system flags](#).
- `mode` `<string>` | `<integer>` **Default:** `0o666`
- Returns: `<number>`

Returns an integer representing the file descriptor.

For detailed information, see the documentation of the asynchronous version of this API: `fs.open()`.

`fs.readdirSync(path[, options])`

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<string>` | `<Object>`
 - `encoding` `<string>` **Default:** `'utf8'`
 - `withFileTypes` `<boolean>` **Default:** `false`
- Returns: `<string[]>` | `<Buffer[]>` | `<fs.Dirent[]>`

Reads the contents of the directory.

See the POSIX `readdir(3)` documentation for more details.

The optional `options` argument can be a string specifying an encoding, or an object with an `encoding` property specifying the character encoding to use for the filenames returned. If the `encoding` is set to `'buffer'`, the filenames returned will be passed as `<Buffer>` objects.

If `options.withFileTypes` is set to `true`, the result will contain `<fs.Dirent>` objects.

`fs.readFileSync(path[, options])`

- `path` `<string>` | `<Buffer>` | `<URL>` | `<integer>` filename or file descriptor
- `options` `<Object>` | `<string>`
 - `encoding` `<string>` | `<null>` **Default:** `null`
 - `flag` `<string>` See [support of file system flags](#). **Default:** `'r'`.
- Returns: `<string>` | `<Buffer>`

Returns the contents of the `path`.

For detailed information, see the documentation of the asynchronous version of this API: `fs.readFile()`.

If the `encoding` option is specified then this function returns a string. Otherwise it returns a buffer.

Similar to `fs.readFile()`, when the path is a directory, the behavior of `fs.readFileSync()` is platform-specific.

```
import { readFileSync } from 'fs';

// macOS, Linux, and Windows
readFileSync('<directory>');
// => [Error: EISDIR: illegal operation on a directory, read <directory>]

// FreeBSD
readFileSync('<directory>'); // => <data>
```

`fs.readlinkSync(path[, options])`

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<string>` | `<Object>`
 - `encoding` `<string>` **Default:** `'utf8'`
- Returns: `<string>` | `<Buffer>`

Returns the symbolic link's string value.

See the POSIX `readlink(2)` documentation for more details.

The optional `options` argument can be a string specifying an encoding, or an object with an `encoding` property specifying the character encoding to use for the link path returned. If the `encoding` is set to `'buffer'`, the link path returned will be passed as a `<Buffer>` object.

`fs.readSync(fd, buffer, offset, length, position)`

- `fd` `<integer>`
- `buffer` `<Buffer>` | `<TypedArray>` | `<DataView>`
- `offset` `<integer>`
- `length` `<integer>`
- `position` `<integer>` | `<bigint>`
- Returns: `<number>`

Returns the number of `bytesRead`.

For detailed information, see the documentation of the asynchronous version of this API: `fs.read()`.

`fs.readSync(fd, buffer, [options])`

- `fd` `<integer>`
- `buffer` `<Buffer>` | `<TypedArray>` | `<DataView>`
- `options` `<Object>`
 - `offset` `<integer>` **Default:** `0`
 - `length` `<integer>` **Default:** `buffer.byteLength`
 - `position` `<integer>` | `<bigint>` **Default:** `null`
- Returns: `<number>`

Returns the number of `bytesRead`.

Similar to the above `fs.readSync` function, this version takes an optional `options` object. If no `options` object is specified, it will default with the above values.

For detailed information, see the documentation of the asynchronous version of this API: `fs.read()`.

`fs.readvSync(fd, buffers[, position])`

- `fd` `<integer>`
- `buffers` `<ArrayBufferView[]>`
- `position` `<integer>`
- Returns: `<number>` The number of bytes read.

For detailed information, see the documentation of the asynchronous version of this API: `fs.readv()`.

`fs.realpathSync(path[, options])`

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<string>` | `<Object>`
 - `encoding` `<string>` **Default:** `'utf8'`
- Returns: `<string>` | `<Buffer>`

Returns the resolved pathname.

For detailed information, see the documentation of the asynchronous version of this API: `fs.realpath()`.

fs.realpathSync.native(path[, options])

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<string>` | `<Object>`
 - `encoding` `<string>` **Default:** `'utf8'`
- Returns: `<string>` | `<Buffer>`

Synchronous `realpath(3)`.

Only paths that can be converted to UTF8 strings are supported.

The optional `options` argument can be a string specifying an encoding, or an object with an `encoding` property specifying the character encoding to use for the path returned. If the `encoding` is set to `'buffer'`, the path returned will be passed as a `<Buffer>` object.

On Linux, when Node.js is linked against musl libc, the procfs file system must be mounted on `/proc` in order for this function to work. Glibc does not have this restriction.

fs.renameSync(oldPath, newPath)

- `oldPath` `<string>` | `<Buffer>` | `<URL>`
- `newPath` `<string>` | `<Buffer>` | `<URL>`

Renames the file from `oldPath` to `newPath`. Returns `undefined`.

See the POSIX `rename(2)` documentation for more details.

fs.rmdirSync(path[, options])

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<Object>`
 - `maxRetries` `<integer>` If an `EBUSY`, `EMFILE`, `ENFILE`, `ENOTEMPTY`, or `EPERM` error is encountered, Node.js retries the operation with a linear backoff wait of `retryDelay` milliseconds longer on each try. This option represents the number of retries. This option is ignored if the `recursive` option is not `true`. **Default:** `0`.
 - `recursive` `<boolean>` If `true`, perform a recursive directory removal. In recursive mode, operations are retried on failure. **Default:** `false`. **Deprecated.**
 - `retryDelay` `<integer>` The amount of time in milliseconds to wait between retries. This option is ignored if the `recursive` option is not `true`. **Default:** `100`.

Synchronous `rmdir(2)`. Returns `undefined`.

Using `fs.rmdirSync()` on a file (not a directory) results in an `ENOENT` error on Windows and an `ENOTDIR` error on POSIX.

To get a behavior similar to the `rm -rf` Unix command, use `fs.rmSync()` with options `{ recursive: true, force: true }`.

fs.rmSync(path[, options])

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<Object>`
 - `force` `<boolean>` When `true`, exceptions will be ignored if `path` does not exist. **Default:** `false`.
 - `maxRetries` `<integer>` If an `EBUSY`, `EMFILE`, `ENFILE`, `ENOTEMPTY`, or `EPERM` error is encountered, Node.js will retry the operation with a linear backoff wait of `retryDelay` milliseconds longer on each try. This option represents the number of retries. This option is ignored if the `recursive` option is not `true`. **Default:** `0`.
 - `recursive` `<boolean>` If `true`, perform a recursive directory removal. In recursive mode operations are retried on failure. **Default:** `false`.

- `retryDelay` `<integer>` The amount of time in milliseconds to wait between retries. This option is ignored if the `recursive` option is not `true`. **Default:** `100`.

Synchronously removes files and directories (modeled on the standard POSIX `rm` utility). Returns `undefined`.

`fs.statSync(path[, options])`

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<Object>`
 - `bigint` `<boolean>` Whether the numeric values in the returned `<fs.Stats>` object should be `bigint`. **Default:** `false`.
 - `throwIfNoEntry` `<boolean>` Whether an exception will be thrown if no file system entry exists, rather than returning `undefined`. **Default:** `true`.
- Returns: `<fs.Stats>`

Retrieves the `<fs.Stats>` for the path.

`fs.symlinkSync(target, path[, type])`

- `target` `<string>` | `<Buffer>` | `<URL>`
- `path` `<string>` | `<Buffer>` | `<URL>`
- `type` `<string>`

Returns `undefined`.

For detailed information, see the documentation of the asynchronous version of this API: `fs.symlink()`.

`fs.truncateSync(path[, len])`

- `path` `<string>` | `<Buffer>` | `<URL>`
- `len` `<integer>` **Default:** `0`

Truncates the file. Returns `undefined`. A file descriptor can also be passed as the first argument. In this case, `fs.ftruncateSync()` is called.

Passing a file descriptor is deprecated and may result in an error being thrown in the future.

`fs.unlinkSync(path)`

- `path` `<string>` | `<Buffer>` | `<URL>`

Synchronous `unlink(2)`. Returns `undefined`.

`fs.utimesSync(path, atime, mtime)`

- `path` `<string>` | `<Buffer>` | `<URL>`
- `atime` `<number>` | `<string>` | `<Date>`
- `mtime` `<number>` | `<string>` | `<Date>`

Returns `undefined`.

For detailed information, see the documentation of the asynchronous version of this API: `fs.utimes()`.

`fs.writeFileSync(file, data[, options])`

- `file` `<string>` | `<Buffer>` | `<URL>` | `<integer>` filename or file descriptor
- `data` `<string>` | `<Buffer>` | `<TypedArray>` | `<DataView>` | `<Object>`
- `options` `<Object>` | `<string>`

- `encoding` `<string>` | `<null>` **Default:** `'utf8'`
- `mode` `<integer>` **Default:** `0o666`
- `flag` `<string>` See [support of file system flags](#) . **Default:** `'w'` .

Returns `undefined` .

If `data` is a plain object, it must have an own (not inherited) `toString` function property.

For detailed information, see the documentation of the asynchronous version of this API: `fs.writeFile()` .

`fs.writeFileSync(fd, buffer[, offset[, length[, position]]])`

- `fd` `<integer>`
- `buffer` `<Buffer>` | `<TypedArray>` | `<DataView>` | `<string>` | `<Object>`
- `offset` `<integer>`
- `length` `<integer>`
- `position` `<integer>`
- Returns: `<number>` The number of bytes written.

If `buffer` is a plain object, it must have an own (not inherited) `toString` function property.

For detailed information, see the documentation of the asynchronous version of this API: `fs.write(fd, buffer...)` .

`fs.writeFileSync(fd, string[, position[, encoding]])`

- `fd` `<integer>`
- `string` `<string>` | `<Object>`
- `position` `<integer>`
- `encoding` `<string>`
- Returns: `<number>` The number of bytes written.

If `string` is a plain object, it must have an own (not inherited) `toString` function property.

For detailed information, see the documentation of the asynchronous version of this API: `fs.write(fd, string...)` .

`fs.writevSync(fd, buffers[, position])`

- `fd` `<integer>`
- `buffers` `<ArrayBufferView[]>`
- `position` `<integer>`
- Returns: `<number>` The number of bytes written.

For detailed information, see the documentation of the asynchronous version of this API: `fs.writev()` .

Common Objects

The common objects are shared by all of the file system API variants (promise, callback, and synchronous).

Class: `fs.Dir`

A class representing a directory stream.

Created by `fs.opendir()` , `fs.opendirSync()` , or `fsPromises.opendir()` .

```
import { opendir } from 'fs/promises';

try {
  const dir = await opendir('.');
  for await (const dirent of dir)
    console.log(dirent.name);
} catch (err) {
  console.error(err);
}
```

When using the async iterator, the `<fs.Dir>` object will be automatically closed after the iterator exits.

`dir.close()`

- Returns: `<Promise>`

Asynchronously close the directory's underlying resource handle. Subsequent reads will result in errors.

A promise is returned that will be resolved after the resource has been closed.

`dir.close(callback)`

- `callback` `<Function>`
 - `err` `<Error>`

Asynchronously close the directory's underlying resource handle. Subsequent reads will result in errors.

The `callback` will be called after the resource handle has been closed.

`dir.closeSync()`

Synchronously close the directory's underlying resource handle. Subsequent reads will result in errors.

`dir.path`

- `<string>`

The read-only path of this directory as was provided to `fs.opendir()`, `fs.opendirSync()`, or `fsPromises.opendir()`.

`dir.read()`

- Returns: `<Promise>` containing `<fs.Dirent>` | `<null>`

Asynchronously read the next directory entry via `readdir(3)` as an `<fs.Dirent>`.

A promise is returned that will be resolved with an `<fs.Dirent>`, or `null` if there are no more directory entries to read.

Directory entries returned by this function are in no particular order as provided by the operating system's underlying directory mechanisms. Entries added or removed while iterating over the directory might not be included in the iteration results.

`dir.read(callback)`

- `callback` `<Function>`
 - `err` `<Error>`
 - `dirent` `<fs.Dirent>` | `<null>`

Asynchronously read the next directory entry via `readdir(3)` as an `<fs.Dirent>`.

After the read is completed, the `callback` will be called with an `<fs.Dirent>`, or `null` if there are no more directory entries to read.

Directory entries returned by this function are in no particular order as provided by the operating system's underlying directory mechanisms. Entries added or removed while iterating over the directory might not be included in the iteration results.

`dir.readSync()`

- Returns: `<fs.Dirent>` | `<null>`

Synchronously read the next directory entry as an `<fs.Dirent>`. See the POSIX `readdir(3)` documentation for more detail.

If there are no more directory entries to read, `null` will be returned.

Directory entries returned by this function are in no particular order as provided by the operating system's underlying directory mechanisms. Entries added or removed while iterating over the directory might not be included in the iteration results.

`dir[Symbol.asyncIterator]()`

- Returns: `<AsyncIterator>` of `<fs.Dirent>`

Asynchronously iterates over the directory until all entries have been read. Refer to the POSIX `readdir(3)` documentation for more detail.

Entries returned by the async iterator are always an `<fs.Dirent>`. The `null` case from `dir.read()` is handled internally.

See `<fs.Dir>` for an example.

Directory entries returned by this iterator are in no particular order as provided by the operating system's underlying directory mechanisms. Entries added or removed while iterating over the directory might not be included in the iteration results.

Class: `fs.Dirent`

A representation of a directory entry, which can be a file or a subdirectory within the directory, as returned by reading from an `<fs.Dir>`. The directory entry is a combination of the file name and file type pairs.

Additionally, when `fs.readdir()` or `fs.readdirSync()` is called with the `withFileTypes` option set to `true`, the resulting array is filled with `<fs.Dirent>` objects, rather than strings or `<Buffer>`s.

`dirent.isBlockDevice()`

- Returns: `<boolean>`

Returns `true` if the `<fs.Dirent>` object describes a block device.

`dirent.isCharacterDevice()`

- Returns: `<boolean>`

Returns `true` if the `<fs.Dirent>` object describes a character device.

`dirent.isDirectory()`

- Returns: `<boolean>`

Returns `true` if the `<fs.Dirent>` object describes a file system directory.

`dirent.isFIFO()`

- Returns: `<boolean>`

Returns `true` if the `<fs.Dirent>` object describes a first-in-first-out (FIFO) pipe.

dirent.isFile()

- Returns: `<boolean>`

Returns `true` if the `<fs.Dirent>` object describes a regular file.

dirent.isSocket()

- Returns: `<boolean>`

Returns `true` if the `<fs.Dirent>` object describes a socket.

dirent.isSymbolicLink()

- Returns: `<boolean>`

Returns `true` if the `<fs.Dirent>` object describes a symbolic link.

dirent.name

- `<string> | <Buffer>`

The file name that this `<fs.Dirent>` object refers to. The type of this value is determined by the `options.encoding` passed to `fs.readdir()` or `fs.readdirSync()`.

Class: fs.FSWatcher

- Extends `<EventEmitter>`

A successful call to `fs.watch()` method will return a new `<fs.FSWatcher>` object.

All `<fs.FSWatcher>` objects emit a `'change'` event whenever a specific watched file is modified.

Event: 'change'

- `eventType` `<string>` The type of change event that has occurred
- `filename` `<string> | <Buffer>` The filename that changed (if relevant/available)

Emitted when something changes in a watched directory or file. See more details in `fs.watch()`.

The `filename` argument may not be provided depending on operating system support. If `filename` is provided, it will be provided as a `<Buffer>` if `fs.watch()` is called with its `encoding` option set to `'buffer'`, otherwise `filename` will be a UTF-8 string.

```
import { watch } from 'fs';
// Example when handled through fs.watch() listener
watch('./tmp', { encoding: 'buffer' }, (eventType, filename) => {
  if (filename) {
    console.log(filename);
    // Prints: <Buffer ...>
  }
});
```

Event: 'close'

Emitted when the watcher stops watching for changes. The closed `<fs.FSWatcher>` object is no longer usable in the event handler.

Event: 'error'

- `error` `<Error>`

Emitted when an error occurs while watching the file. The errored `<fs.FSWatcher>` object is no longer usable in the event handler.

`watcher.close()`

Stop watching for changes on the given `<fs.FSWatcher>`. Once stopped, the `<fs.FSWatcher>` object is no longer usable.

`watcher.ref()`

- Returns: `<fs.FSWatcher>`

When called, requests that the Node.js event loop *not* exit so long as the `<fs.FSWatcher>` is active. Calling `watcher.ref()` multiple times will have no effect.

By default, all `<fs.FSWatcher>` objects are "ref'ed", making it normally unnecessary to call `watcher.ref()` unless `watcher.unref()` had been called previously.

`watcher.unref()`

- Returns: `<fs.FSWatcher>`

When called, the active `<fs.FSWatcher>` object will not require the Node.js event loop to remain active. If there is no other activity keeping the event loop running, the process may exit before the `<fs.FSWatcher>` object's callback is invoked. Calling `watcher.unref()` multiple times will have no effect.

Class: `fs.StatWatcher`

- Extends `<EventEmitter>`

A successful call to `fs.watchFile()` method will return a new `<fs.StatWatcher>` object.

`watcher.ref()`

- Returns: `<fs.StatWatcher>`

When called, requests that the Node.js event loop *not* exit so long as the `<fs.StatWatcher>` is active. Calling `watcher.ref()` multiple times will have no effect.

By default, all `<fs.StatWatcher>` objects are "ref'ed", making it normally unnecessary to call `watcher.ref()` unless `watcher.unref()` had been called previously.

`watcher.unref()`

- Returns: `<fs.StatWatcher>`

When called, the active `<fs.StatWatcher>` object will not require the Node.js event loop to remain active. If there is no other activity keeping the event loop running, the process may exit before the `<fs.StatWatcher>` object's callback is invoked. Calling `watcher.unref()` multiple times will have no effect.

Class: `fs.ReadStream`

- Extends: `<stream.Readable>`

Instances of `<fs.ReadStream>` are created and returned using the `fs.createReadStream()` function.

Event: `'close'`

Emitted when the `<fs.ReadStream>`'s underlying file descriptor has been closed.

Event: 'open'

- `fd` `<integer>` Integer file descriptor used by the `<fs.ReadStream>`.

Emitted when the `<fs.ReadStream>`'s file descriptor has been opened.

Event: 'ready'

Emitted when the `<fs.ReadStream>` is ready to be used.

Fires immediately after `'open'`.

`readStream.bytesRead`

- `<number>`

The number of bytes that have been read so far.

`readStream.path`

- `<string>` | `<Buffer>`

The path to the file the stream is reading from as specified in the first argument to `fs.createReadStream()`. If `path` is passed as a string, then `readStream.path` will be a string. If `path` is passed as a `<Buffer>`, then `readStream.path` will be a `<Buffer>`.

`readStream.pending`

- `<boolean>`

This property is `true` if the underlying file has not been opened yet, i.e. before the `'ready'` event is emitted.

Class: `fs.Stats`

A `<fs.Stats>` object provides information about a file.

Objects returned from `fs.stat()`, `fs.lstat()` and `fs.fstat()` and their synchronous counterparts are of this type. If `bigint` in the `options` passed to those methods is true, the numeric values will be `bigint` instead of `number`, and the object will contain additional nanosecond-precision properties suffixed with `Ns`.

```
Stats {
  dev: 2114,
  ino: 48064969,
  mode: 33188,
  nlink: 1,
  uid: 85,
  gid: 100,
  rdev: 0,
  size: 527,
  blksize: 4096,
  blocks: 8,
  atimeMs: 1318289051000.1,
  mtimeMs: 1318289051000.1,
  ctimeMs: 1318289051000.1,
  birthtimeMs: 1318289051000.1,
  atime: Mon, 10 Oct 2011 23:24:11 GMT,
  mtime: Mon, 10 Oct 2011 23:24:11 GMT,
```

```
ctime: Mon, 10 Oct 2011 23:24:11 GMT,  
birthtime: Mon, 10 Oct 2011 23:24:11 GMT }
```

bigint version:

```
BigIntStats {  
  dev: 2114n,  
  ino: 48064969n,  
  mode: 33188n,  
  nlink: 1n,  
  uid: 85n,  
  gid: 100n,  
  rdev: 0n,  
  size: 527n,  
  blksize: 4096n,  
  blocks: 8n,  
  atimeMs: 1318289051000n,  
  mtimeMs: 1318289051000n,  
  ctimeMs: 1318289051000n,  
  birthtimeMs: 1318289051000n,  
  atimeNs: 1318289051000000000n,  
  mtimeNs: 1318289051000000000n,  
  ctimeNs: 1318289051000000000n,  
  birthtimeNs: 1318289051000000000n,  
  atime: Mon, 10 Oct 2011 23:24:11 GMT,  
  mtime: Mon, 10 Oct 2011 23:24:11 GMT,  
  ctime: Mon, 10 Oct 2011 23:24:11 GMT,  
  birthtime: Mon, 10 Oct 2011 23:24:11 GMT }
```

stats.isBlockDevice()

- Returns: **<boolean>**

Returns **true** if the **<fs.Stats>** object describes a block device.

stats.isCharacterDevice()

- Returns: **<boolean>**

Returns **true** if the **<fs.Stats>** object describes a character device.

stats.isDirectory()

- Returns: **<boolean>**

Returns **true** if the **<fs.Stats>** object describes a file system directory.

If the **<fs.Stats>** object was obtained from **fs.lstat()**, this method will always return **false**. This is because **fs.lstat()** returns information about a symbolic link itself and not the path it resolves to.

stats.isFIFO()

- Returns: **<boolean>**

Returns **true** if the **<fs.Stats>** object describes a first-in-first-out (FIFO) pipe.

stats.isFile()

- Returns: `<boolean>`

Returns `true` if the `<fs.Stats>` object describes a regular file.

stats.isSocket()

- Returns: `<boolean>`

Returns `true` if the `<fs.Stats>` object describes a socket.

stats.isSymbolicLink()

- Returns: `<boolean>`

Returns `true` if the `<fs.Stats>` object describes a symbolic link.

This method is only valid when using `fs.lstat()` .

stats.dev

- `<number> | <bigint>`

The numeric identifier of the device containing the file.

stats.ino

- `<number> | <bigint>`

The file system specific "Inode" number for the file.

stats.mode

- `<number> | <bigint>`

A bit-field describing the file type and mode.

stats.nlink

- `<number> | <bigint>`

The number of hard-links that exist for the file.

stats.uid

- `<number> | <bigint>`

The numeric user identifier of the user that owns the file (POSIX).

stats.gid

- `<number> | <bigint>`

The numeric group identifier of the group that owns the file (POSIX).

stats.rdev

- `<number> | <bigint>`

A numeric device identifier if the file represents a device.

stats.size

- `<number> | <bigint>`

The size of the file in bytes.

stats.blksize

- `<number> | <bigint>`

The file system block size for i/o operations.

stats.blocks

- `<number> | <bigint>`

The number of blocks allocated for this file.

stats.atimeMs

- `<number> | <bigint>`

The timestamp indicating the last time this file was accessed expressed in milliseconds since the POSIX Epoch.

stats.mtimeMs

- `<number> | <bigint>`

The timestamp indicating the last time this file was modified expressed in milliseconds since the POSIX Epoch.

stats.ctimeMs

- `<number> | <bigint>`

The timestamp indicating the last time the file status was changed expressed in milliseconds since the POSIX Epoch.

stats.birthtimeMs

- `<number> | <bigint>`

The timestamp indicating the creation time of this file expressed in milliseconds since the POSIX Epoch.

stats.atimeNs

- `<bigint>`

Only present when `bigint: true` is passed into the method that generates the object. The timestamp indicating the last time this file was accessed expressed in nanoseconds since the POSIX Epoch.

stats.mtimeNs

- `<bigint>`

Only present when `bigint: true` is passed into the method that generates the object. The timestamp indicating the last time this file was modified expressed in nanoseconds since the POSIX Epoch.

stats.ctimeNs

- `<bigint>`

Only present when `bigint: true` is passed into the method that generates the object. The timestamp indicating the last time the file status was changed expressed in nanoseconds since the POSIX Epoch.

stats.birthtimeNs

- `<bigint>`

Only present when `bigint: true` is passed into the method that generates the object. The timestamp indicating the creation time of this file expressed in nanoseconds since the POSIX Epoch.

stats.atime

- `<Date>`

The timestamp indicating the last time this file was accessed.

stats.mtime

- `<Date>`

The timestamp indicating the last time this file was modified.

stats.ctime

- `<Date>`

The timestamp indicating the last time the file status was changed.

stats.birthtime

- `<Date>`

The timestamp indicating the creation time of this file.

Stat time values

The `atimeMs`, `mtimeMs`, `ctimeMs`, `birthtimeMs` properties are numeric values that hold the corresponding times in milliseconds. Their precision is platform specific. When `bigint: true` is passed into the method that generates the object, the properties will be `bigints`, otherwise they will be `numbers`.

The `atimeNs`, `mtimeNs`, `ctimeNs`, `birthtimeNs` properties are `bigints` that hold the corresponding times in nanoseconds. They are only present when `bigint: true` is passed into the method that generates the object. Their precision is platform specific.

`atime`, `mtime`, `ctime`, and `birthtime` are `Date` object alternate representations of the various times. The `Date` and number values are not connected. Assigning a new number value, or mutating the `Date` value, will not be reflected in the corresponding alternate representation.

The times in the stat object have the following semantics:

- `atime` "Access Time": Time when file data last accessed. Changed by the `mknod(2)`, `utimes(2)`, and `read(2)` system calls.
- `mtime` "Modified Time": Time when file data last modified. Changed by the `mknod(2)`, `utimes(2)`, and `write(2)` system calls.
- `ctime` "Change Time": Time when file status was last changed (inode data modification). Changed by the `chmod(2)`, `chown(2)`, `link(2)`, `mknod(2)`, `rename(2)`, `unlink(2)`, `utimes(2)`, `read(2)`, and `write(2)` system calls.
- `birthtime` "Birth Time": Time of file creation. Set once when the file is created. On filesystems where birthtime is not available, this field may instead hold either the `ctime` or `1970-01-01T00:00Z` (ie, Unix epoch timestamp `0`). This value may be greater than `atime` or `mtime` in this case. On Darwin and other FreeBSD variants, also set if the `atime` is explicitly set to an earlier value than the current `birthtime` using the `utimes(2)` system call.

Prior to Node.js 0.12, the `ctime` held the `birthtime` on Windows systems. As of 0.12, `ctime` is not "creation time", and on Unix systems, it never was.

Class: fs.WriteStream

- Extends `<stream.Writable>`

Instances of `<fs.WriteStream>` are created and returned using the `fs.createWriteStream()` function.

Event: 'close'

Emitted when the `<fs.WriteStream>`'s underlying file descriptor has been closed.

Event: 'open'

- `fd` `<integer>` Integer file descriptor used by the `<fs.WriteStream>`.

Emitted when the `<fs.WriteStream>`'s file is opened.

Event: 'ready'

Emitted when the `<fs.WriteStream>` is ready to be used.

Fires immediately after `'open'`.

writeStream.bytesWritten

The number of bytes written so far. Does not include data that is still queued for writing.

writeStream.close([callback])

- `callback` `<Function>`
 - `err` `<Error>`

Closes `writeStream`. Optionally accepts a callback that will be executed once the `writeStream` is closed.

writeStream.path

The path to the file the stream is writing to as specified in the first argument to `fs.createWriteStream()`. If `path` is passed as a string, then `writeStream.path` will be a string. If `path` is passed as a `<Buffer>`, then `writeStream.path` will be a `<Buffer>`.

writeStream.pending

- `<boolean>`

This property is `true` if the underlying file has not been opened yet, i.e. before the `'ready'` event is emitted.

fs.constants

- `<Object>`

Returns an object containing commonly used constants for file system operations.

FS constants

The following constants are exported by `fs.constants`.

Not every constant will be available on every operating system.

To use more than one constant, use the bitwise OR `|` operator.

Example:

```
import { open, constants } from 'fs';

const {
```

```
O_RDWR,
O_CREAT,
O_EXCL
} = constants;

open('/path/to/my/file', O_RDWR | O_CREAT | O_EXCL, (err, fd) => {
  // ...
});
```

File access constants

The following constants are meant for use with `fs.access()`.

Constant	Description
<code>F_OK</code>	Flag indicating that the file is visible to the calling process. This is useful for determining if a file exists, but says nothing about <code>rxw</code> permissions. Default if no mode is specified.
<code>R_OK</code>	Flag indicating that the file can be read by the calling process.
<code>W_OK</code>	Flag indicating that the file can be written by the calling process.
<code>X_OK</code>	Flag indicating that the file can be executed by the calling process. This has no effect on Windows (will behave like <code>fs.constants.F_OK</code>).

File copy constants

The following constants are meant for use with `fs.copyFile()`.

Constant	Description
<code>COPYFILE_EXCL</code>	If present, the copy operation will fail with an error if the destination path already exists.
<code>COPYFILE_FICLONE</code>	If present, the copy operation will attempt to create a copy-on-write reflink. If the underlying platform does not support copy-on-write, then a fallback copy mechanism is used.
<code>COPYFILE_FICLONE_FORCE</code>	If present, the copy operation will attempt to create a copy-on-write reflink. If the underlying platform does not support copy-on-write, then the operation will fail with an error.

File open constants

The following constants are meant for use with `fs.open()`.

Constant	Description
<code>O_RDONLY</code>	Flag indicating to open a file for read-only access.
<code>O_WRONLY</code>	Flag indicating to open a file for write-only access.
<code>O_RDWR</code>	Flag indicating to open a file for read-write access.
<code>O_CREAT</code>	Flag indicating to create the file if it does not already exist.
<code>O_EXCL</code>	Flag indicating that opening a file should fail if the <code>O_CREAT</code> flag is set and the file already exists.
<code>O_NOCTTY</code>	Flag indicating that if path identifies a terminal device, opening the path shall not cause that terminal to become the controlling terminal for the process (if the process does not already have one).

<code>O_TRUNC</code>	Flag indicating that if the file exists and is a regular file, and the file is opened successfully for write access, its length shall be truncated to zero.
<code>O_APPEND</code>	Flag indicating that data will be appended to the end of the file.
<code>O_DIRECTORY</code>	Flag indicating that the open should fail if the path is not a directory.
<code>O_NOATIME</code>	Flag indicating reading accesses to the file system will no longer result in an update to the <code>atime</code> information associated with the file. This flag is available on Linux operating systems only.
<code>O_NOFOLLOW</code>	Flag indicating that the open should fail if the path is a symbolic link.
<code>O_SYNC</code>	Flag indicating that the file is opened for synchronized I/O with write operations waiting for file integrity.
<code>O_DSYNC</code>	Flag indicating that the file is opened for synchronized I/O with write operations waiting for data integrity.
<code>O_SYMLINK</code>	Flag indicating to open the symbolic link itself rather than the resource it is pointing to.
<code>O_DIRECT</code>	When set, an attempt will be made to minimize caching effects of file I/O.
<code>O_NONBLOCK</code>	Flag indicating to open the file in nonblocking mode when possible.
<code>UV_FS_O_FILEMAP</code>	When set, a memory file mapping is used to access the file. This flag is available on Windows operating systems only. On other operating systems, this flag is ignored.

File type constants

The following constants are meant for use with the `<fs.Stats>` object's `mode` property for determining a file's type.

Constant	Description
<code>S_IFMT</code>	Bit mask used to extract the file type code.
<code>S_IFREG</code>	File type constant for a regular file.
<code>S_IFDIR</code>	File type constant for a directory.
<code>S_IFCHR</code>	File type constant for a character-oriented device file.
<code>S_IFBLK</code>	File type constant for a block-oriented device file.
<code>S_IFIFO</code>	File type constant for a FIFO/pipe.
<code>S_IFLNK</code>	File type constant for a symbolic link.
<code>S_IFSOCK</code>	File type constant for a socket.

File mode constants

The following constants are meant for use with the `<fs.Stats>` object's `mode` property for determining the access permissions for a file.

Constant	Description
<code>S_IRWXU</code>	File mode indicating readable, writable, and executable by owner.
<code>S_IRUSR</code>	File mode indicating readable by owner.

S_IWUSR	File mode indicating writable by owner.
S_IXUSR	File mode indicating executable by owner.
S_IRWXG	File mode indicating readable, writable, and executable by group.
S_IRGRP	File mode indicating readable by group.
S_IWGRP	File mode indicating writable by group.
S_IXGRP	File mode indicating executable by group.
S_IRWXO	File mode indicating readable, writable, and executable by others.
S_IROTH	File mode indicating readable by others.
S_IWOTH	File mode indicating writable by others.
S_IXOTH	File mode indicating executable by others.

Notes

Ordering of callback and promise-based operations

Because they are executed asynchronously by the underlying thread pool, there is no guaranteed ordering when using either the callback or promise-based methods.

For example, the following is prone to error because the `fs.stat()` operation might complete before the `fs.rename()` operation:

```
fs.rename('/tmp/hello', '/tmp/world', (err) => {
  if (err) throw err;
  console.log('renamed complete');
});
fs.stat('/tmp/world', (err, stats) => {
  if (err) throw err;
  console.log(`stats: ${JSON.stringify(stats)}`);
});
```

It is important to correctly order the operations by awaiting the results of one before invoking the other:

```
import { rename, stat } from 'fs/promises';

const from = '/tmp/hello';
const to = '/tmp/world';

try {
  await rename(from, to);
  const stats = await stat(to);
  console.log(`stats: ${JSON.stringify(stats)}`);
} catch (error) {
  console.error('there was an error:', error.message);
}const { rename, stat } = require('fs/promises');
```

```
(async function(from, to) {
  try {
    await rename(from, to);
    const stats = await stat(to);
    console.log(`stats: ${JSON.stringify(stats)}`);
  } catch (error) {
    console.error('there was an error:', error.message);
  }
})('/tmp/hello', '/tmp/world');
```

Or, when using the callback APIs, move the `fs.stat()` call into the callback of the `fs.rename()` operation:

```
import { rename, stat } from 'fs';

rename('/tmp/hello', '/tmp/world', (err) => {
  if (err) throw err;
  stat('/tmp/world', (err, stats) => {
    if (err) throw err;
    console.log(`stats: ${JSON.stringify(stats)}`);
  });
});const { rename, stat } = require('fs/promises');

rename('/tmp/hello', '/tmp/world', (err) => {
  if (err) throw err;
  stat('/tmp/world', (err, stats) => {
    if (err) throw err;
    console.log(`stats: ${JSON.stringify(stats)}`);
  });
});
```

File paths

Most `fs` operations accept file paths that may be specified in the form of a string, a `<Buffer>`, or a `<URL>` object using the `file:` protocol.

String paths

String form paths are interpreted as UTF-8 character sequences identifying the absolute or relative filename. Relative paths will be resolved relative to the current working directory as determined by calling `process.cwd()`.

Example using an absolute path on POSIX:

```
import { open } from 'fs/promises';

let fd;
try {
  fd = await open('/open/some/file.txt', 'r');
  // Do something with the file
} finally {
  await fd.close();
}
```

Example using a relative path on POSIX (relative to `process.cwd()`):

```
import { open } from 'fs/promises';

let fd;
try {
  fd = await open('file.txt', 'r');
  // Do something with the file
} finally {
  await fd.close();
}
```

File URL paths

For most `fs` module functions, the `path` or `filename` argument may be passed as a `<URL>` object using the `file:` protocol.

```
import { readFileSync } from 'fs';

readFileSync(new URL('file:///tmp/hello'));
```

`file:` URLs are always absolute paths.

Platform-specific considerations

On Windows, `file:` `<URL>`s with a host name convert to UNC paths, while `file:` `<URL>`s with drive letters convert to local absolute paths. `file:` `<URL>`s without a host name nor a drive letter will result in an error:

```
import { readFileSync } from 'fs';
// On Windows :

// - WHATWG file URLs with hostname convert to UNC path
// file://hostname/p/a/t/h/file => \\hostname\p\a\t\h\file
readFileSync(new URL('file://hostname/p/a/t/h/file'));

// - WHATWG file URLs with drive letters convert to absolute path
// file:///C:/tmp/hello => C:\tmp\hello
readFileSync(new URL('file:///C:/tmp/hello'));

// - WHATWG file URLs without hostname must have a drive letters
readFileSync(new URL('file:///notdriveletter/p/a/t/h/file'));
readFileSync(new URL('file:///c/p/a/t/h/file'));
// TypeError [ERR_INVALID_FILE_URL_PATH]: File URL path must be absolute
```

`file:` `<URL>`s with drive letters must use `:` as a separator just after the drive letter. Using another separator will result in an error.

On all other platforms, `file:` `<URL>`s with a host name are unsupported and will result in an error:

```
import { readFileSync } from 'fs';
// On other platforms:
```

```
// - WHATWG file URLs with hostname are unsupported
// file://hostname/p/a/t/h/file => throw!
readFileSync(new URL('file://hostname/p/a/t/h/file'));
// TypeError [ERR_INVALID_FILE_URL_PATH]: must be absolute

// - WHATWG file URLs convert to absolute path
// file:///tmp/hello => /tmp/hello
readFileSync(new URL('file:///tmp/hello'));
```

A file: <URL> having encoded slash characters will result in an error on all platforms:

```
import { readFileSync } from 'fs';

// On Windows
readFileSync(new URL('file:///C:/p/a/t/h/%2F'));
readFileSync(new URL('file:///C:/p/a/t/h/%2f'));
/* TypeError [ERR_INVALID_FILE_URL_PATH]: File URL path must not include encoded
\ or / characters */

// On POSIX
readFileSync(new URL('file:///p/a/t/h/%2F'));
readFileSync(new URL('file:///p/a/t/h/%2f'));
/* TypeError [ERR_INVALID_FILE_URL_PATH]: File URL path must not include encoded
/ characters */
```

On Windows, file: <URL> s having encoded backslash will result in an error:

```
import { readFileSync } from 'fs';

// On Windows
readFileSync(new URL('file:///C:/path/%5C'));
readFileSync(new URL('file:///C:/path/%5c'));
/* TypeError [ERR_INVALID_FILE_URL_PATH]: File URL path must not include encoded
\ or / characters */
```

Buffer paths

Paths specified using a <Buffer> are useful primarily on certain POSIX operating systems that treat file paths as opaque byte sequences. On such systems, it is possible for a single file path to contain sub-sequences that use multiple character encodings. As with string paths, <Buffer> paths may be relative or absolute:

Example using an absolute path on POSIX:

```
import { open } from 'fs/promises';
import { Buffer } from 'buffer';

let fd;
try {
  fd = await open(Buffer.from('/open/some/file.txt'), 'r');
  // Do something with the file
```

```
} finally {  
  await fd.close();  
}
```

Per-drive working directories on Windows

On Windows, Node.js follows the concept of per-drive working directory. This behavior can be observed when using a drive path without a backslash. For example `fs.readdirSync('C:\\')` can potentially return a different result than `fs.readdirSync('C:')`. For more information, see [this MSDN page](#).

File descriptors

On POSIX systems, for every process, the kernel maintains a table of currently open files and resources. Each open file is assigned a simple numeric identifier called a *file descriptor*. At the system-level, all file system operations use these file descriptors to identify and track each specific file. Windows systems use a different but conceptually similar mechanism for tracking resources. To simplify things for users, Node.js abstracts away the differences between operating systems and assigns all open files a numeric file descriptor.

The callback-based `fs.open()`, and synchronous `fs.openSync()` methods open a file and allocate a new file descriptor. Once allocated, the file descriptor may be used to read data from, write data to, or request information about the file.

Operating systems limit the number of file descriptors that may be open at any given time so it is critical to close the descriptor when operations are completed. Failure to do so will result in a memory leak that will eventually cause an application to crash.

```
import { open, close, fstat } from 'fs';  
  
function closeFd(fd) {  
  close(fd, (err) => {  
    if (err) throw err;  
  });  
}  
  
open('/open/some/file.txt', 'r', (err, fd) => {  
  if (err) throw err;  
  try {  
    fstat(fd, (err, stat) => {  
      if (err) {  
        closeFd(fd);  
        throw err;  
      }  
  
      // use stat  
  
      closeFd(fd);  
    });  
  } catch (err) {  
    closeFd(fd);  
    throw err;  
  }  
});
```

The promise-based APIs use a `<FileHandle>` object in place of the numeric file descriptor. These objects are better managed by the system to ensure that resources are not leaked. However, it is still required that they are closed when operations are completed:


```
import { open } from 'fs/promises';

let file;
try {
  file = await open('/open/some/file.txt', 'r');
  const stat = await file.stat();
  // use stat
} finally {
  await file.close();
}
```

Threadpool usage

All callback and promise-based file system APIs (with the exception of `fs.FSWatcher()`) use libuv's threadpool. This can have surprising and negative performance implications for some applications. See the [UV_THREADPOOL_SIZE](#) documentation for more information.

File system flags

The following flags are available wherever the `flag` option takes a string.

- `'a'`: Open file for appending. The file is created if it does not exist.
- `'ax'`: Like `'a'` but fails if the path exists.
- `'a+'`: Open file for reading and appending. The file is created if it does not exist.
- `'ax+'`: Like `'a+'` but fails if the path exists.
- `'as'`: Open file for appending in synchronous mode. The file is created if it does not exist.
- `'as+'`: Open file for reading and appending in synchronous mode. The file is created if it does not exist.
- `'r'`: Open file for reading. An exception occurs if the file does not exist.
- `'r+'`: Open file for reading and writing. An exception occurs if the file does not exist.
- `'rs+'`: Open file for reading and writing in synchronous mode. Instructs the operating system to bypass the local file system cache.

This is primarily useful for opening files on NFS mounts as it allows skipping the potentially stale local cache. It has a very real impact on I/O performance so using this flag is not recommended unless it is needed.

This doesn't turn `fs.open()` or `fsPromises.open()` into a synchronous blocking call. If synchronous operation is desired, something like `fs.openSync()` should be used.

- `'w'`: Open file for writing. The file is created (if it does not exist) or truncated (if it exists).
- `'wx'`: Like `'w'` but fails if the path exists.
- `'w+'`: Open file for reading and writing. The file is created (if it does not exist) or truncated (if it exists).
- `'wx+'`: Like `'w+'` but fails if the path exists.

`flag` can also be a number as documented by `open(2)`; commonly used constants are available from `fs.constants`. On Windows, flags are translated to their equivalent ones where applicable, e.g. `O_WRONLY` to `FILE_GENERIC_WRITE`, or `O_EXCL|O_CREAT` to `CREATE_NEW`, as accepted by `CreateFileW`.

The exclusive flag `'x'` (`O_EXCL` flag in `open(2)`) causes the operation to return an error if the path already exists. On POSIX, if the path is a symbolic link, using `O_EXCL` returns an error even if the link is to a path that does not exist. The exclusive flag might not work with network file systems.

On Linux, positional writes don't work when the file is opened in append mode. The kernel ignores the position argument and always appends the data to the end of the file.

Modifying a file rather than replacing it may require the `flag` option to be set to `'r+'` rather than the default `'w'`.

The behavior of some flags are platform-specific. As such, opening a directory on macOS and Linux with the `'a+'` flag, as in the example below, will return an error. In contrast, on Windows and FreeBSD, a file descriptor or a `FileHandle` will be returned.

```
// macOS and Linux
fs.open('<directory>', 'a+', (err, fd) => {
  // => [Error: EISDIR: illegal operation on a directory, open <directory>]
});

// Windows and FreeBSD
fs.open('<directory>', 'a+', (err, fd) => {
  // => null, <fd>
});
```

On Windows, opening an existing hidden file using the `'w'` flag (either through `fs.open()` or `fs.writeFile()` or `fsPromises.open()`) will fail with `EPERM`. Existing hidden files can be opened for writing with the `'r+'` flag.

A call to `fs.ftruncate()` or `filehandle.truncate()` can be used to reset the file contents.

Global objects

These objects are available in all modules. The following variables may appear to be global but are not. They exist only in the scope of modules, see the [module system documentation](#) :

- `__dirname`
- `__filename`
- `exports`
- `module`
- `require()`

The objects listed here are specific to Node.js. There are [built-in objects](#) that are part of the JavaScript language itself, which are also globally accessible.

Class: `AbortController`

A utility class used to signal cancelation in selected `Promise`-based APIs. The API is based on the Web API [AbortController](#) .

```
const ac = new AbortController();

ac.signal.addEventListener('abort', () => console.log('Aborted!'),
  { once: true });

ac.abort();
```

```
console.log(ac.signal.aborted); // Prints True
```

abortController.abort()

Triggers the abort signal, causing the `abortController.signal` to emit the `'abort'` event.

abortController.signal

- Type: `<AbortSignal>`

Class: AbortSignal

- Extends: `<EventTarget>`

The `AbortSignal` is used to notify observers when the `abortController.abort()` method is called.

Static method: AbortSignal.abort()

- Returns: `<AbortSignal>`

Returns a new already aborted `AbortSignal`.

Event: 'abort'

The `'abort'` event is emitted when the `abortController.abort()` method is called. The callback is invoked with a single object argument with a single `type` property set to `'abort'`:

```
const ac = new AbortController();

// Use either the onabort property...
ac.signal.onabort = () => console.log('aborted!');

// Or the EventTarget API...
ac.signal.addEventListener('abort', (event) => {
  console.log(event.type); // Prints 'abort'
}, { once: true });

ac.abort();
```

The `AbortController` with which the `AbortSignal` is associated will only ever trigger the `'abort'` event once. We recommended that code check that the `abortSignal.aborted` attribute is `false` before adding an `'abort'` event listener.

Any event listeners attached to the `AbortSignal` should use the `{ once: true }` option (or, if using the `EventEmitter` APIs to attach a listener, use the `once()` method) to ensure that the event listener is removed as soon as the `'abort'` event is handled. Failure to do so may result in memory leaks.

abortSignal.aborted

- Type: `<boolean>` True after the `AbortController` has been aborted.

abortSignal.onabort

- Type: `<Function>`

An optional callback function that may be set by user code to be notified when the `abortController.abort()` function has been called.

Class: Buffer

- `<Function>`

Used to handle binary data. See the `buffer` section .

`__dirname`

This variable may appear to be global but is not. See `__dirname` .

`__filename`

This variable may appear to be global but is not. See `__filename` .

`atob(data)`

Stability: 3 - Legacy. Use `Buffer.from(data, 'base64')` instead.

Global alias for `buffer.atob()` .

`btoa(data)`

Stability: 3 - Legacy. Use `buf.toString('base64')` instead.

Global alias for `buffer.btoa()` .

`clearImmediate(immediateObject)`

`clearImmediate` is described in the `timers` section.

`clearInterval(intervalObject)`

`clearInterval` is described in the `timers` section.

`clearTimeout(timeoutObject)`

`clearTimeout` is described in the `timers` section.

`console`

- `<Object>`

Used to print to stdout and stderr. See the `console` section.

Event

A browser-compatible implementation of the `Event` class. See [EventTarget and Event API](#) for more details.

EventTarget

A browser-compatible implementation of the `EventTarget` class. See [EventTarget and Event API](#) for more details.

exports

This variable may appear to be global but is not. See [exports](#).

global

- `<Object>` The global namespace object.

In browsers, the top-level scope is the global scope. This means that within the browser `var something` will define a new global variable. In Node.js this is different. The top-level scope is not the global scope; `var something` inside a Node.js module will be local to that module.

MessageChannel

The `MessageChannel` class. See [MessageChannel](#) for more details.

MessageEvent

The `MessageEvent` class. See [MessageEvent](#) for more details.

MessagePort

The `MessagePort` class. See [MessagePort](#) for more details.

module

This variable may appear to be global but is not. See [module](#).

performance

The `perf_hooks.performance` object.

process

- `<Object>`

The process object. See the [process object](#) section.

queueMicrotask(callback)

- `callback` `<Function>` Function to be queued.

The `queueMicrotask()` method queues a microtask to invoke `callback`. If `callback` throws an exception, the `process` object `'uncaughtException'` event will be emitted.

The microtask queue is managed by V8 and may be used in a similar manner to the `process.nextTick()` queue, which is managed by Node.js. The `process.nextTick()` queue is always processed before the microtask queue within each turn of the Node.js event loop.

```
// Here, `queueMicrotask()` is used to ensure the 'load' event is always
// emitted asynchronously, and therefore consistently. Using
// `process.nextTick()` here would result in the 'load' event always emitting
// before any other promise jobs.
```

```
DataHandler.prototype.load = async function load(key) {
  const hit = this._cache.get(key);
  if (hit !== undefined) {
    queueMicrotask(() => {
      this.emit('load', hit);
    });
    return;
  }

  const data = await fetchData(key);
  this._cache.set(key, data);
  this.emit('load', data);
};
```

require()

This variable may appear to be global but is not. See `require()`.

setImmediate(callback[, ...args])

`setImmediate` is described in the `timers` section.

setInterval(callback, delay[, ...args])

`setInterval` is described in the `timers` section.

setTimeout(callback, delay[, ...args])

`setTimeout` is described in the `timers` section.

TextDecoder

The WHATWG `TextDecoder` class. See the `TextDecoder` section.

TextEncoder

The WHATWG `TextEncoder` class. See the [TextEncoder](#) section.

URL

The WHATWG `URL` class. See the [URL](#) section.

URLSearchParams

The WHATWG `URLSearchParams` class. See the [URLSearchParams](#) section.

WebAssembly

- `<Object>`

The object that acts as the namespace for all W3C [WebAssembly](#) related functionality. See the [Mozilla Developer Network](#) for usage and compatibility.

HTTP

Stability: 2 - Stable

Source Code: [lib/http.js](#)

To use the HTTP server and client one must `require('http')`.

The HTTP interfaces in Node.js are designed to support many features of the protocol which have been traditionally difficult to use. In particular, large, possibly chunk-encoded, messages. The interface is careful to never buffer entire requests or responses, so the user is able to stream data.

HTTP message headers are represented by an object like this:

```
{ 'content-length': '123',
  'content-type': 'text/plain',
  'connection': 'keep-alive',
  'host': 'mysite.com',
  'accept': '*/.*' }
```

Keys are lowercased. Values are not modified.

In order to support the full spectrum of possible HTTP applications, the Node.js HTTP API is very low-level. It deals with stream handling and message parsing only. It parses a message into headers and body but it does not parse the actual headers or the body.

See `message.headers` for details on how duplicate headers are handled.

The raw headers as they were received are retained in the `rawHeaders` property, which is an array of `[key, value, key2, value2, ...]`. For example, the previous message header object might have a `rawHeaders` list like the following:

```
[ 'Content-Length', '123456',
  'content-LENGTH', '123',
```

```
'content-type', 'text/plain',
'CONNECTION', 'keep-alive',
'Host', 'mysite.com',
'accept', ' */*' ]
```

Class: http.Agent

An `Agent` is responsible for managing connection persistence and reuse for HTTP clients. It maintains a queue of pending requests for a given host and port, reusing a single socket connection for each until the queue is empty, at which time the socket is either destroyed or put into a pool where it is kept to be used again for requests to the same host and port. Whether it is destroyed or pooled depends on the `keepAlive` option.

Pooled connections have TCP Keep-Alive enabled for them, but servers may still close idle connections, in which case they will be removed from the pool and a new connection will be made when a new HTTP request is made for that host and port. Servers may also refuse to allow multiple requests over the same connection, in which case the connection will have to be remade for every request and cannot be pooled. The `Agent` will still make the requests to that server, but each one will occur over a new connection.

When a connection is closed by the client or the server, it is removed from the pool. Any unused sockets in the pool will be unrefed so as not to keep the Node.js process running when there are no outstanding requests. (see `socket.unref()`).

It is good practice, to `destroy()` an `Agent` instance when it is no longer in use, because unused sockets consume OS resources.

Sockets are removed from an agent when the socket emits either a `'close'` event or an `'agentRemove'` event. When intending to keep one HTTP request open for a long time without keeping it in the agent, something like the following may be done:

```
http.get(options, (res) => {
  // Do stuff
}).on('socket', (socket) => {
  socket.emit('agentRemove');
});
```

An agent may also be used for an individual request. By providing `{agent: false}` as an option to the `http.get()` or `http.request()` functions, a one-time use `Agent` with default options will be used for the client connection.

`agent: false:`

```
http.get({
  hostname: 'localhost',
  port: 80,
  path: '/',
  agent: false // Create a new agent just for this one request
}, (res) => {
  // Do stuff with response
});
```

new Agent([options])

- `options` `<Object>` Set of configurable options to set on the agent. Can have the following fields:
 - `keepAlive` `<boolean>` Keep sockets around even when there are no outstanding requests, so they can be used for future requests without having to reestablish a TCP connection. Not to be confused with the `keep-alive` value of the `Connection` header. The `Connection: keep-alive` header is always sent when using an agent except when the `Connection` header is explicitly specified

or when the `keepAlive` and `maxSockets` options are respectively set to `false` and `Infinity`, in which case `Connection: close` will be used. **Default:** `false`.

- `keepAliveMsecs` `<number>` When using the `keepAlive` option, specifies the `initial delay` for TCP Keep-Alive packets. Ignored when the `keepAlive` option is `false` or `undefined`. **Default:** `1000`.
- `maxSockets` `<number>` Maximum number of sockets to allow per host. If the same host opens multiple concurrent connections, each request will use new socket until the `maxSockets` value is reached. If the host attempts to open more connections than `maxSockets`, the additional requests will enter into a pending request queue, and will enter active connection state when an existing connection terminates. This makes sure there are at most `maxSockets` active connections at any point in time, from a given host. **Default:** `Infinity`.
- `maxTotalSockets` `<number>` Maximum number of sockets allowed for all hosts in total. Each request will use a new socket until the maximum is reached. **Default:** `Infinity`.
- `maxFreeSockets` `<number>` Maximum number of sockets to leave open in a free state. Only relevant if `keepAlive` is set to `true`. **Default:** `256`.
- `scheduling` `<string>` Scheduling strategy to apply when picking the next free socket to use. It can be `'fifo'` or `'lifo'`. The main difference between the two scheduling strategies is that `'lifo'` selects the most recently used socket, while `'fifo'` selects the least recently used socket. In case of a low rate of request per second, the `'lifo'` scheduling will lower the risk of picking a socket that might have been closed by the server due to inactivity. In case of a high rate of request per second, the `'fifo'` scheduling will maximize the number of open sockets, while the `'lifo'` scheduling will keep it as low as possible. **Default:** `'lifo'`.
- `timeout` `<number>` Socket timeout in milliseconds. This will set the timeout when the socket is created.

`options` in `socket.connect()` are also supported.

The default `http.globalAgent` that is used by `http.request()` has all of these values set to their respective defaults.

To configure any of them, a custom `http.Agent` instance must be created.

```
const http = require('http');
const keepAliveAgent = new http.Agent({ keepAlive: true });
options.agent = keepAliveAgent;
http.request(options, onResponseCallback);
```

agent.createConnection(options[, callback])

- `options` `<Object>` Options containing connection details. Check `net.createConnection()` for the format of the options
- `callback` `<Function>` Callback function that receives the created socket
- Returns: `<stream.Duplex>`

Produces a socket/stream to be used for HTTP requests.

By default, this function is the same as `net.createConnection()`. However, custom agents may override this method in case greater flexibility is desired.

A socket/stream can be supplied in one of two ways: by returning the socket/stream from this function, or by passing the socket/stream to `callback`.

This method is guaranteed to return an instance of the `<net.Socket>` class, a subclass of `<stream.Duplex>`, unless the user specifies a socket type other than `<net.Socket>`.

`callback` has a signature of `(err, stream)`.

agent.keepSocketAlive(socket)

- `socket` `<stream.Duplex>`

Called when `socket` is detached from a request and could be persisted by the `Agent`. Default behavior is to:

```
socket.setKeepAlive(true, this.keepAliveMsecs);
socket.unref();
return true;
```

This method can be overridden by a particular `Agent` subclass. If this method returns a falsy value, the socket will be destroyed instead of persisting it for use with the next request.

The `socket` argument can be an instance of `<net.Socket>`, a subclass of `<stream.Duplex>`.

agent.reuseSocket(socket, request)

- `socket` `<stream.Duplex>`
- `request` `<http.ClientRequest>`

Called when `socket` is attached to `request` after being persisted because of the keep-alive options. Default behavior is to:

```
socket.ref();
```

This method can be overridden by a particular `Agent` subclass.

The `socket` argument can be an instance of `<net.Socket>`, a subclass of `<stream.Duplex>`.

agent.destroy()

Destroy any sockets that are currently in use by the agent.

It is usually not necessary to do this. However, if using an agent with `keepAlive` enabled, then it is best to explicitly shut down the agent when it is no longer needed. Otherwise, sockets might stay open for quite a long time before the server terminates them.

agent.freeSockets

- `<Object>`

An object which contains arrays of sockets currently awaiting use by the agent when `keepAlive` is enabled. Do not modify.

Sockets in the `freeSockets` list will be automatically destroyed and removed from the array on `'timeout'`.

agent.getName(options)

- `options` `<Object>` A set of options providing information for name generation
 - `host` `<string>` A domain name or IP address of the server to issue the request to
 - `port` `<number>` Port of remote server
 - `localAddress` `<string>` Local interface to bind for network connections when issuing the request
 - `family` `<integer>` Must be 4 or 6 if this doesn't equal `undefined`.
- Returns: `<string>`

Get a unique name for a set of request options, to determine whether a connection can be reused. For an HTTP agent, this returns `host:port:localAddress` or `host:port:localAddress:family`. For an HTTPS agent, the name includes the CA, cert, ciphers, and other HTTPS/TLS-specific options that determine socket reusability.

agent.maxFreeSockets

- `<number>`

By default set to 256. For agents with `keepAlive` enabled, this sets the maximum number of sockets that will be left open in the free state.

agent.maxSockets

- `<number>`

By default set to `Infinity`. Determines how many concurrent sockets the agent can have open per origin. Origin is the returned value of `agent.getName()`.

agent.maxTotalSockets

- `<number>`

By default set to `Infinity`. Determines how many concurrent sockets the agent can have open. Unlike `maxSockets`, this parameter applies across all origins.

agent.requests

- `<Object>`

An object which contains queues of requests that have not yet been assigned to sockets. Do not modify.

agent.sockets

- `<Object>`

An object which contains arrays of sockets currently in use by the agent. Do not modify.

Class: http.ClientRequest

- Extends: `<Stream>`

This object is created internally and returned from `http.request()`. It represents an *in-progress* request whose header has already been queued. The header is still mutable using the `setHeader(name, value)`, `getHeader(name)`, `removeHeader(name)` API. The actual header will be sent along with the first data chunk or when calling `request.end()`.

To get the response, add a listener for `'response'` to the request object. `'response'` will be emitted from the request object when the response headers have been received. The `'response'` event is executed with one argument which is an instance of `http.IncomingMessage`.

During the `'response'` event, one can add listeners to the response object; particularly to listen for the `'data'` event.

If no `'response'` handler is added, then the response will be entirely discarded. However, if a `'response'` event handler is added, then the data from the response object **must** be consumed, either by calling `response.read()` whenever there is a `'readable'` event, or by adding a `'data'` handler, or by calling the `.resume()` method. Until the data is consumed, the `'end'` event will not fire. Also, until the data is read it will consume memory that can eventually lead to a 'process out of memory' error.

For backward compatibility, `res` will only emit `'error'` if there is an `'error'` listener registered.

Node.js does not check whether Content-Length and the length of the body which has been transmitted are equal or not.

Event: 'abort'

Emitted when the request has been aborted by the client. This event is only emitted on the first call to `abort()`.

Event: 'connect'

- `response` `<http.IncomingMessage>`
- `socket` `<stream.Duplex>`
- `head` `<Buffer>`

Emitted each time a server responds to a request with a `CONNECT` method. If this event is not being listened for, clients receiving a `CONNECT` method will have their connections closed.

This event is guaranteed to be passed an instance of the `<net.Socket>` class, a subclass of `<stream.Duplex>`, unless the user specifies a socket type other than `<net.Socket>`.

A client and server pair demonstrating how to listen for the `'connect'` event:

```
const http = require('http');
const net = require('net');
const { URL } = require('url');

// Create an HTTP tunneling proxy
const proxy = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('okay');
});
proxy.on('connect', (req, clientSocket, head) => {
  // Connect to an origin server
  const { port, hostname } = new URL(`http://${req.url}`);
  const serverSocket = net.connect(port || 80, hostname, () => {
    clientSocket.write('HTTP/1.1 200 Connection Established\r\n' +
      'Proxy-agent: Node.js-Proxy\r\n' +
      '\r\n');
    serverSocket.write(head);
    serverSocket.pipe(clientSocket);
    clientSocket.pipe(serverSocket);
  });
});

// Now that proxy is running
proxy.listen(1337, '127.0.0.1', () => {

  // Make a request to a tunneling proxy
  const options = {
    port: 1337,
    host: '127.0.0.1',
    method: 'CONNECT',
    path: 'www.google.com:80'
  };

  const req = http.request(options);
  req.end();

  req.on('connect', (res, socket, head) => {
    console.log('got connected!');

    // Make a request over an HTTP tunnel
```

```

socket.write('GET / HTTP/1.1\r\n' +
             'Host: www.google.com:80\r\n' +
             'Connection: close\r\n' +
             '\r\n');
socket.on('data', (chunk) => {
  console.log(chunk.toString());
});
socket.on('end', () => {
  proxy.close();
});
});

```

Event: 'continue'

Emitted when the server sends a '100 Continue' HTTP response, usually because the request contained 'Expect: 100-continue'. This is an instruction that the client should send the request body.

Event: 'information'

- `info` `<Object>`
 - `httpVersion` `<string>`
 - `httpVersionMajor` `<integer>`
 - `httpVersionMinor` `<integer>`
 - `statusCode` `<integer>`
 - `statusMessage` `<string>`
 - `headers` `<Object>`
 - `rawHeaders` `<string[]>`

Emitted when the server sends a 1xx intermediate response (excluding 101 Upgrade). The listeners of this event will receive an object containing the HTTP version, status code, status message, key-value headers object, and array with the raw header names followed by their respective values.

```

const http = require('http');

const options = {
  host: '127.0.0.1',
  port: 8080,
  path: '/length_request'
};

// Make a request
const req = http.request(options);
req.end();

req.on('information', (info) => {
  console.log(`Got information prior to main response: ${info.statusCode}`);
});

```

101 Upgrade statuses do not fire this event due to their break from the traditional HTTP request/response chain, such as web sockets, in-place TLS upgrades, or HTTP 2.0. To be notified of 101 Upgrade notices, listen for the `'upgrade'` event instead.

Event: `'response'`

- `response` `<http.IncomingMessage>`

Emitted when a response is received to this request. This event is emitted only once.

Event: `'socket'`

- `socket` `<stream.Duplex>`

This event is guaranteed to be passed an instance of the `<net.Socket>` class, a subclass of `<stream.Duplex>`, unless the user specifies a socket type other than `<net.Socket>`.

Event: `'timeout'`

Emitted when the underlying socket times out from inactivity. This only notifies that the socket has been idle. The request must be aborted manually.

See also: `request.setTimeout()`.

Event: `'upgrade'`

- `response` `<http.IncomingMessage>`
- `socket` `<stream.Duplex>`
- `head` `<Buffer>`

Emitted each time a server responds to a request with an upgrade. If this event is not being listened for and the response status code is 101 Switching Protocols, clients receiving an upgrade header will have their connections closed.

This event is guaranteed to be passed an instance of the `<net.Socket>` class, a subclass of `<stream.Duplex>`, unless the user specifies a socket type other than `<net.Socket>`.

A client server pair demonstrating how to listen for the `'upgrade'` event.

```
const http = require('http');

// Create an HTTP server
const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('okay');
});

server.on('upgrade', (req, socket, head) => {
  socket.write('HTTP/1.1 101 Web Socket Protocol Handshake\r\n' +
    'Upgrade: WebSocket\r\n' +
    'Connection: Upgrade\r\n' +
    '\r\n');

  socket.pipe(socket); // echo back
});

// Now that server is running
server.listen(1337, '127.0.0.1', () => {
```

```
// make a request
const options = {
  port: 1337,
  host: '127.0.0.1',
  headers: {
    'Connection': 'Upgrade',
    'Upgrade': 'websocket'
  }
};

const req = http.request(options);
req.end();

req.on('upgrade', (res, socket, upgradeHead) => {
  console.log('got upgraded!');
  socket.end();
  process.exit(0);
});
});
```

request.abort()

Stability: 0 - Deprecated: Use `request.destroy()` instead.

Marks the request as aborting. Calling this will cause remaining data in the response to be dropped and the socket to be destroyed.

request.aborted

- `<boolean>`

The `request.aborted` property will be `true` if the request has been aborted.

request.connection

Stability: 0 - Deprecated. Use `request.socket`.

- `<stream.Duplex>`

See `request.socket`.

request.end([data[, encoding]][, callback])

- `data` `<string> | <Buffer>`
- `encoding` `<string>`
- `callback` `<Function>`
- Returns: `<this>`

Finishes sending the request. If any parts of the body are unsent, it will flush them to the stream. If the request is chunked, this will send the terminating `'0\r\n\r\n'`.

If `data` is specified, it is equivalent to calling `request.write(data, encoding)` followed by `request.end(callback)`.

If `callback` is specified, it will be called when the request stream is finished.

`request.destroy([error])`

- `error` `<Error>` Optional, an error to emit with `'error'` event.
- Returns: `<this>`

Destroy the request. Optionally emit an `'error'` event, and emit a `'close'` event. Calling this will cause remaining data in the response to be dropped and the socket to be destroyed.

See `writable.destroy()` for further details.

`request.destroyed`

- `<boolean>`

Is `true` after `request.destroy()` has been called.

See `writable.destroyed` for further details.

`request.finished`

Stability: 0 - Deprecated. Use `request.writableEnded`.

- `<boolean>`

The `request.finished` property will be `true` if `request.end()` has been called. `request.end()` will automatically be called if the request was initiated via `http.get()`.

`request.flushHeaders()`

Flushes the request headers.

For efficiency reasons, Node.js normally buffers the request headers until `request.end()` is called or the first chunk of request data is written. It then tries to pack the request headers and data into a single TCP packet.

That's usually desired (it saves a TCP round-trip), but not when the first data is not sent until possibly much later. `request.flushHeaders()` bypasses the optimization and kickstarts the request.

`request.getHeader(name)`

- `name` `<string>`
- Returns: `<any>`

Reads out a header on the request. The name is case-insensitive. The type of the return value depends on the arguments provided to `request.setHeader()`.

```
request.setHeader('content-type', 'text/html');
request.setHeader('Content-Length', Buffer.byteLength(body));
request.setHeader('Cookie', ['type=ninja', 'language=javascript']);
const contentType = request.getHeader('Content-Type');
// 'contentType' is 'text/html'
const contentLength = request.getHeader('Content-Length');
```



```
// 'contentLength' is of type number
const cookie = request.getHeader('Cookie');
// 'cookie' is of type string[]
```

request.getRawHeaderNames()

- Returns: `<string[]>`

Returns an array containing the unique names of the current outgoing raw headers. Header names are returned with their exact casing being set.

```
request.setHeader('Foo', 'bar');
request.setHeader('Set-Cookie', ['foo=bar', 'bar=baz']);

const headerNames = request.getRawHeaderNames();
// headerNames === ['Foo', 'Set-Cookie']
```

request.maxHeadersCount

- `<number>` Default: 2000

Limits maximum response headers count. If set to 0, no limit will be applied.

request.path

- `<string>` The request path.

request.method

- `<string>` The request method.

request.host

- `<string>` The request host.

request.protocol

- `<string>` The request protocol.

request.removeHeader(name)

- `name` `<string>`

Removes a header that's already defined into headers object.

```
request.removeHeader('Content-Type');
```

request.reusedSocket

- `<boolean>` Whether the request is send through a reused socket.

When sending request through a keep-alive enabled agent, the underlying socket might be reused. But if server closes connection at unfortunate time, client may run into a 'ECONNRESET' error.

```

const http = require('http');

// Server has a 5 seconds keep-alive timeout by default
http
  .createServer((req, res) => {
    res.write('hello\n');
    res.end();
  })
  .listen(3000);

setInterval(() => {
  // Adapting a keep-alive agent
  http.get('http://localhost:3000', { agent }, (res) => {
    res.on('data', (data) => {
      // Do nothing
    });
  });
}, 5000); // Sending request on 5s interval so it's easy to hit idle timeout

```

By marking a request whether it reused socket or not, we can do automatic error retry base on it.

```

const http = require('http');
const agent = new http.Agent({ keepAlive: true });

function retrieableRequest() {
  const req = http
    .get('http://localhost:3000', { agent }, (res) => {
      // ...
    })
    .on('error', (err) => {
      // Check if retry is needed
      if (req.reusedSocket && err.code === 'ECONNRESET') {
        retrieableRequest();
      }
    });
}

retrieableRequest();

```

request.setHeader(name, value)

- `name` `<string>`
- `value` `<any>`

Sets a single header value for headers object. If this header already exists in the to-be-sent headers, its value will be replaced. Use an array of strings here to send multiple headers with the same name. Non-string values will be stored without modification. Therefore, `request.getHeader()` may return non-string values. However, the non-string values will be converted to strings for network transmission.

```

request.setHeader('Content-Type', 'application/json');

```

or

```
request.setHeader('Cookie', ['type=ninja', 'language=javascript']);
```

request.setNoDelay([noDelay])

- `noDelay` `<boolean>`

Once a socket is assigned to this request and is connected `socket.setNoDelay()` will be called.

request.setSocketKeepAlive([enable][, initialDelay])

- `enable` `<boolean>`
- `initialDelay` `<number>`

Once a socket is assigned to this request and is connected `socket.setKeepAlive()` will be called.

request.setTimeout(timeout[, callback])

- `timeout` `<number>` Milliseconds before a request times out.
- `callback` `<Function>` Optional function to be called when a timeout occurs. Same as binding to the `'timeout'` event.
- Returns: `<http.ClientRequest>`

Once a socket is assigned to this request and is connected `socket.setTimeout()` will be called.

request.socket

- `<stream.Duplex>`

Reference to the underlying socket. Usually users will not want to access this property. In particular, the socket will not emit `'readable'` events because of how the protocol parser attaches to the socket.

```
const http = require('http');
const options = {
  host: 'www.google.com',
};
const req = http.get(options);
req.end();
req.once('response', (res) => {
  const ip = req.socket.localAddress;
  const port = req.socket.localPort;
  console.log(`Your IP address is ${ip} and your source port is ${port}.`);
  // Consume response object
});
```

This property is guaranteed to be an instance of the `<net.Socket>` class, a subclass of `<stream.Duplex>`, unless the user specified a socket type other than `<net.Socket>`.

request.writableEnded

- `<boolean>`

Is `true` after `request.end()` has been called. This property does not indicate whether the data has been flushed, for this use `request.writableFinished` instead.

request.writableFinished

- `<boolean>`

Is `true` if all data has been flushed to the underlying system, immediately before the `'finish'` event is emitted.

request.write(chunk[, encoding][, callback])

- `chunk` `<string>` | `<Buffer>`
- `encoding` `<string>`
- `callback` `<Function>`
- Returns: `<boolean>`

Sends a chunk of the body. This method can be called multiple times. If no `Content-Length` is set, data will automatically be encoded in HTTP Chunked transfer encoding, so that server knows when the data ends. The `Transfer-Encoding: chunked` header is added. Calling `request.end()` is necessary to finish sending the request.

The `encoding` argument is optional and only applies when `chunk` is a string. Defaults to `'utf8'`.

The `callback` argument is optional and will be called when this chunk of data is flushed, but only if the chunk is non-empty.

Returns `true` if the entire data was flushed successfully to the kernel buffer. Returns `false` if all or part of the data was queued in user memory. `'drain'` will be emitted when the buffer is free again.

When `write` function is called with empty string or buffer, it does nothing and waits for more input.

Class: http.Server

- Extends: `<net.Server>`

Event: 'checkContinue'

- `request` `<http.IncomingMessage>`
- `response` `<http.ServerResponse>`

Emitted each time a request with an HTTP `Expect: 100-continue` is received. If this event is not listened for, the server will automatically respond with a `100 Continue` as appropriate.

Handling this event involves calling `response.writeContinue()` if the client should continue to send the request body, or generating an appropriate HTTP response (e.g. 400 Bad Request) if the client should not continue to send the request body.

When this event is emitted and handled, the `'request'` event will not be emitted.

Event: 'checkExpectation'

- `request` `<http.IncomingMessage>`
- `response` `<http.ServerResponse>`

Emitted each time a request with an HTTP `Expect` header is received, where the value is not `100-continue`. If this event is not listened for, the server will automatically respond with a `417 Expectation Failed` as appropriate.

When this event is emitted and handled, the `'request'` event will not be emitted.

Event: 'clientError'

- `exception` `<Error>`

- `socket` `<stream.Duplex>`

If a client connection emits an `'error'` event, it will be forwarded here. Listener of this event is responsible for closing/destroying the underlying socket. For example, one may wish to more gracefully close the socket with a custom HTTP response instead of abruptly severing the connection.

This event is guaranteed to be passed an instance of the `<net.Socket>` class, a subclass of `<stream.Duplex>`, unless the user specifies a socket type other than `<net.Socket>`.

Default behavior is to try close the socket with a HTTP '400 Bad Request', or a HTTP '431 Request Header Fields Too Large' in the case of a `HPE_HEADER_OVERFLOW` error. If the socket is not writable or has already written data it is immediately destroyed.

`socket` is the `net.Socket` object that the error originated from.

```
const http = require('http');

const server = http.createServer((req, res) => {
  res.end();
});
server.on('clientError', (err, socket) => {
  socket.end('HTTP/1.1 400 Bad Request\r\n\r\n');
});
server.listen(8000);
```

When the `'clientError'` event occurs, there is no `request` or `response` object, so any HTTP response sent, including response headers and payload, *must* be written directly to the `socket` object. Care must be taken to ensure the response is a properly formatted HTTP response message.

`err` is an instance of `Error` with two extra columns:

- `bytesParsed` : the bytes count of request packet that Node.js may have parsed correctly;
- `rawPacket` : the raw packet of current request.

In some cases, the client has already received the response and/or the socket has already been destroyed, like in case of `ECONNRESET` errors. Before trying to send data to the socket, it is better to check that it is still writable.

```
server.on('clientError', (err, socket) => {
  if (err.code === 'ECONNRESET' || !socket.writable) {
    return;
  }

  socket.end('HTTP/1.1 400 Bad Request\r\n\r\n');
});
```

Event: 'close'

Emitted when the server closes.

Event: 'connect'

- `request` `<http.IncomingMessage>` Arguments for the HTTP request, as it is in the `'request'` event
- `socket` `<stream.Duplex>` Network socket between the server and client
- `head` `<Buffer>` The first packet of the tunneling stream (may be empty)

Emitted each time a client requests an HTTP `CONNECT` method. If this event is not listened for, then clients requesting a `CONNECT` method will have their connections closed.

This event is guaranteed to be passed an instance of the `<net.Socket>` class, a subclass of `<stream.Duplex>`, unless the user specifies a socket type other than `<net.Socket>`.

After this event is emitted, the request's socket will not have a `'data'` event listener, meaning it will need to be bound in order to handle data sent to the server on that socket.

Event: 'connection'

- `socket` `<stream.Duplex>`

This event is emitted when a new TCP stream is established. `socket` is typically an object of type `net.Socket`. Usually users will not want to access this event. In particular, the socket will not emit `'readable'` events because of how the protocol parser attaches to the socket. The `socket` can also be accessed at `request.socket`.

This event can also be explicitly emitted by users to inject connections into the HTTP server. In that case, any `Duplex` stream can be passed.

If `socket.setTimeout()` is called here, the timeout will be replaced with `server.keepAliveTimeout` when the socket has served a request (if `server.keepAliveTimeout` is non-zero).

This event is guaranteed to be passed an instance of the `<net.Socket>` class, a subclass of `<stream.Duplex>`, unless the user specifies a socket type other than `<net.Socket>`.

Event: 'request'

- `request` `<http.IncomingMessage>`
- `response` `<http.ServerResponse>`

Emitted each time there is a request. There may be multiple requests per connection (in the case of HTTP Keep-Alive connections).

Event: 'upgrade'

- `request` `<http.IncomingMessage>` Arguments for the HTTP request, as it is in the `'request'` event
- `socket` `<stream.Duplex>` Network socket between the server and client
- `head` `<Buffer>` The first packet of the upgraded stream (may be empty)

Emitted each time a client requests an HTTP upgrade. Listening to this event is optional and clients cannot insist on a protocol change.

After this event is emitted, the request's socket will not have a `'data'` event listener, meaning it will need to be bound in order to handle data sent to the server on that socket.

This event is guaranteed to be passed an instance of the `<net.Socket>` class, a subclass of `<stream.Duplex>`, unless the user specifies a socket type other than `<net.Socket>`.

`server.close([callback])`

- `callback` `<Function>`

Stops the server from accepting new connections. See `net.Server.close()`.

`server.headersTimeout`

- `<number>` **Default:** `60000`

Limit the amount of time the parser will wait to receive the complete HTTP headers.

In case of inactivity, the rules defined in `server.timeout` apply. However, that inactivity based timeout would still allow the connection to be kept open if the headers are being sent very slowly (by default, up to a byte per 2 minutes). In order to prevent this, whenever header data arrives an additional check is made that more than `server.headersTimeout` milliseconds has not passed since the connection was established. If the check fails, a `'timeout'` event is emitted on the server object, and (by default) the socket is destroyed. See `server.timeout` for more information on how timeout behavior can be customized.

`server.listen()`

Starts the HTTP server listening for connections. This method is identical to `server.listen()` from `net.Server`.

`server.listening`

- `<boolean>` Indicates whether or not the server is listening for connections.

`server.maxHeadersCount`

- `<number>` **Default:** 2000

Limits maximum incoming headers count. If set to 0, no limit will be applied.

`server.requestTimeout`

- `<number>` **Default:** 0

Sets the timeout value in milliseconds for receiving the entire request from the client.

If the timeout expires, the server responds with status 408 without forwarding the request to the request listener and then closes the connection.

It must be set to a non-zero value (e.g. 120 seconds) to protect against potential Denial-of-Service attacks in case the server is deployed without a reverse proxy in front.

`server.setTimeout([msecs][, callback])`

- `msecs` `<number>` **Default:** 0 (no timeout)
- `callback` `<Function>`
- Returns: `<http.Server>`

Sets the timeout value for sockets, and emits a `'timeout'` event on the Server object, passing the socket as an argument, if a timeout occurs.

If there is a `'timeout'` event listener on the Server object, then it will be called with the timed-out socket as an argument.

By default, the Server does not timeout sockets. However, if a callback is assigned to the Server's `'timeout'` event, timeouts must be handled explicitly.

`server.timeout`

- `<number>` Timeout in milliseconds. **Default:** 0 (no timeout)

The number of milliseconds of inactivity before a socket is presumed to have timed out.

A value of 0 will disable the timeout behavior on incoming connections.

The socket timeout logic is set up on connection, so changing this value only affects new connections to the server, not any existing connections.

`server.keepAliveTimeout`

- `<number>` Timeout in milliseconds. **Default:** 5000 (5 seconds).

The number of milliseconds of inactivity a server needs to wait for additional incoming data, after it has finished writing the last response, before a socket will be destroyed. If the server receives new data before the keep-alive timeout has fired, it will reset the regular inactivity timeout, i.e., `server.timeout`.

A value of `0` will disable the keep-alive timeout behavior on incoming connections. A value of `0` makes the http server behave similarly to Node.js versions prior to 8.0.0, which did not have a keep-alive timeout.

The socket timeout logic is set up on connection, so changing this value only affects new connections to the server, not any existing connections.

Class: `http.ServerResponse`

- Extends: `<Stream>`

This object is created internally by an HTTP server, not by the user. It is passed as the second parameter to the `'request'` event.

Event: `'close'`

Indicates that the response is completed, or its underlying connection was terminated prematurely (before the response completion).

Event: `'finish'`

Emitted when the response has been sent. More specifically, this event is emitted when the last segment of the response headers and body have been handed off to the operating system for transmission over the network. It does not imply that the client has received anything yet.

`response.addTrailers(headers)`

- `headers` `<Object>`

This method adds HTTP trailing headers (a header but at the end of the message) to the response.

Trailers will **only** be emitted if chunked encoding is used for the response; if it is not (e.g. if the request was HTTP/1.0), they will be silently discarded.

HTTP requires the `Trailer` header to be sent in order to emit trailers, with a list of the header fields in its value. E.g.,

```
response.writeHead(200, { 'Content-Type': 'text/plain',
                          'Trailer': 'Content-MD5' });
response.write(fileData);
response.addTrailers({ 'Content-MD5': '7895bf4b8828b55ceaf47747b4bca667' });
response.end();
```

Attempting to set a header field name or value that contains invalid characters will result in a `TypeError` being thrown.

`response.connection`

Stability: 0 - Deprecated. Use `response.socket`.

- `<stream.Duplex>`

See `response.socket`.

`response.cork()`

See `writable.cork()` .

`response.end([data[, encoding]][, callback])`

- `data` `<string> | <Buffer>`
- `encoding` `<string>`
- `callback` `<Function>`
- Returns: `<this>`

This method signals to the server that all of the response headers and body have been sent; that server should consider this message complete. The method, `response.end()` , MUST be called on each response.

If `data` is specified, it is similar in effect to calling `response.write(data, encoding)` followed by `response.end(callback)` .

If `callback` is specified, it will be called when the response stream is finished.

`response.finished`

Stability: 0 - Deprecated. Use `response.writableEnded` .

- `<boolean>`

The `response.finished` property will be `true` if `response.end()` has been called.

`response.flushHeaders()`

Flushes the response headers. See also: `request.flushHeaders()` .

`response.getHeader(name)`

- `name` `<string>`
- Returns: `<any>`

Reads out a header that's already been queued but not sent to the client. The name is case-insensitive. The type of the return value depends on the arguments provided to `response.setHeader()` .

```
response.setHeader('Content-Type', 'text/html');
response.setHeader('Content-Length', Buffer.byteLength(body));
response.setHeader('Set-Cookie', ['type=ninja', 'language=javascript']);
const contentType = response.getHeader('content-type');
// contentType is 'text/html'
const contentLength = response.getHeader('Content-Length');
// contentLength is of type number
const setCookie = response.getHeader('set-cookie');
// setCookie is of type string[]
```

`response.getHeaderNames()`

- Returns: `<string[]>`

Returns an array containing the unique names of the current outgoing headers. All header names are lowercase.

```
response.setHeader('Foo', 'bar');
response.setHeader('Set-Cookie', ['foo=bar', 'bar=baz']);

const headerNames = response.getHeaderNames();
// headerNames === ['foo', 'set-cookie']
```

response.getHeaders()

- Returns: `<Object>`

Returns a shallow copy of the current outgoing headers. Since a shallow copy is used, array values may be mutated without additional calls to various header-related http module methods. The keys of the returned object are the header names and the values are the respective header values. All header names are lowercase.

The object returned by the `response.getHeaders()` method *does not* prototypically inherit from the JavaScript `Object`. This means that typical `Object` methods such as `obj.toString()`, `obj.hasOwnProperty()`, and others are not defined and *will not work*.

```
response.setHeader('Foo', 'bar');
response.setHeader('Set-Cookie', ['foo=bar', 'bar=baz']);

const headers = response.getHeaders();
// headers === { foo: 'bar', 'set-cookie': ['foo=bar', 'bar=baz'] }
```

response.hasHeader(name)

- `name` `<string>`
- Returns: `<boolean>`

Returns `true` if the header identified by `name` is currently set in the outgoing headers. The header name matching is case-insensitive.

```
const hasContentType = response.hasHeader('content-type');
```

response.headersSent

- `<boolean>`

Boolean (read-only). True if headers were sent, false otherwise.

response.removeHeader(name)

- `name` `<string>`

Removes a header that's queued for implicit sending.

```
response.removeHeader('Content-Encoding');
```

response.req

- `<http.IncomingMessage>`

A reference to the original HTTP `request` object.

response.sendDate

- `<boolean>`

When true, the Date header will be automatically generated and sent in the response if it is not already present in the headers. Defaults to true.

This should only be disabled for testing; HTTP requires the Date header in responses.

`response.setHeader(name, value)`

- `name` `<string>`
- `value` `<any>`
- Returns: `<http.ServerResponse>`

Returns the response object.

Sets a single header value for implicit headers. If this header already exists in the to-be-sent headers, its value will be replaced. Use an array of strings here to send multiple headers with the same name. Non-string values will be stored without modification. Therefore, `response.getHeader()` may return non-string values. However, the non-string values will be converted to strings for network transmission. The same response object is returned to the caller, to enable call chaining.

```
response.setHeader('Content-Type', 'text/html');
```

or

```
response.setHeader('Set-Cookie', ['type=ninja', 'language=javascript']);
```

Attempting to set a header field name or value that contains invalid characters will result in a `TypeError` being thrown.

When headers have been set with `response.setHeader()`, they will be merged with any headers passed to `response.writeHead()`, with the headers passed to `response.writeHead()` given precedence.

```
// Returns content-type = text/plain
const server = http.createServer((req, res) => {
  res.setHeader('Content-Type', 'text/html');
  res.setHeader('X-Foo', 'bar');
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('ok');
});
```

If `response.writeHead()` method is called and this method has not been called, it will directly write the supplied header values onto the network channel without caching internally, and the `response.getHeader()` on the header will not yield the expected result. If progressive population of headers is desired with potential future retrieval and modification, use `response.setHeader()` instead of `response.writeHead()`.

`response.setTimeout(msecs[, callback])`

- `msecs` `<number>`
- `callback` `<Function>`
- Returns: `<http.ServerResponse>`

Sets the Socket's timeout value to `msecs`. If a callback is provided, then it is added as a listener on the `'timeout'` event on the response object.

If no `'timeout'` listener is added to the request, the response, or the server, then sockets are destroyed when they time out. If a handler is assigned to the request, the response, or the server's `'timeout'` events, timed out sockets must be handled explicitly.

response.socket

- `<stream.Duplex>`

Reference to the underlying socket. Usually users will not want to access this property. In particular, the socket will not emit `'readable'` events because of how the protocol parser attaches to the socket. After `response.end()`, the property is nulled.

```
const http = require('http');
const server = http.createServer((req, res) => {
  const ip = res.socket.remoteAddress;
  const port = res.socket.remotePort;
  res.end(`Your IP address is ${ip} and your source port is ${port}.`);
}).listen(3000);
```

This property is guaranteed to be an instance of the `<net.Socket>` class, a subclass of `<stream.Duplex>`, unless the user specified a socket type other than `<net.Socket>`.

response.statusCode

- `<number>` **Default:** 200

When using implicit headers (not calling `response.writeHead()` explicitly), this property controls the status code that will be sent to the client when the headers get flushed.

```
response.statusCode = 404;
```

After response header was sent to the client, this property indicates the status code which was sent out.

response.statusMessage

- `<string>`

When using implicit headers (not calling `response.writeHead()` explicitly), this property controls the status message that will be sent to the client when the headers get flushed. If this is left as `undefined` then the standard message for the status code will be used.

```
response.statusMessage = 'Not found';
```

After response header was sent to the client, this property indicates the status message which was sent out.

response.uncork()

See `writable.uncork()`.

response.writableEnded

- `<boolean>`

Is `true` after `response.end()` has been called. This property does not indicate whether the data has been flushed, for this use `response.writableFinished` instead.

response.writableFinished

- `<boolean>`

Is `true` if all data has been flushed to the underlying system, immediately before the `'finish'` event is emitted.

`response.write(chunk[, encoding][, callback])`

- `chunk` `<string>` | `<Buffer>`
- `encoding` `<string>` Default: `'utf8'`
- `callback` `<Function>`
- Returns: `<boolean>`

If this method is called and `response.writeHead()` has not been called, it will switch to implicit header mode and flush the implicit headers.

This sends a chunk of the response body. This method may be called multiple times to provide successive parts of the body.

In the `http` module, the response body is omitted when the request is a HEAD request. Similarly, the `204` and `304` responses *must not* include a message body.

`chunk` can be a string or a buffer. If `chunk` is a string, the second parameter specifies how to encode it into a byte stream. `callback` will be called when this chunk of data is flushed.

This is the raw HTTP body and has nothing to do with higher-level multi-part body encodings that may be used.

The first time `response.write()` is called, it will send the buffered header information and the first chunk of the body to the client. The second time `response.write()` is called, Node.js assumes data will be streamed, and sends the new data separately. That is, the response is buffered up to the first chunk of the body.

Returns `true` if the entire data was flushed successfully to the kernel buffer. Returns `false` if all or part of the data was queued in user memory. `'drain'` will be emitted when the buffer is free again.

`response.writeContinue()`

Sends a HTTP/1.1 100 Continue message to the client, indicating that the request body should be sent. See the `'checkContinue'` event on `Server`.

`response.writeHead(statusCode[, statusMessage][, headers])`

- `statusCode` `<number>`
- `statusMessage` `<string>`
- `headers` `<Object>` | `<Array>`
- Returns: `<http.ServerResponse>`

Sends a response header to the request. The status code is a 3-digit HTTP status code, like `404`. The last argument, `headers`, are the response headers. Optionally one can give a human-readable `statusMessage` as the second argument.

`headers` may be an `Array` where the keys and values are in the same list. It is *not* a list of tuples. So, the even-numbered offsets are key values, and the odd-numbered offsets are the associated values. The array is in the same format as `request.rawHeaders`.

Returns a reference to the `ServerResponse`, so that calls can be chained.

```
const body = 'hello world';
response
  .writeHead(200, {
    'Content-Length': Buffer.byteLength(body),
    'Content-Type': 'text/plain'
```

```
})  
.end(body);
```

This method must only be called once on a message and it must be called before `response.end()` is called.

If `response.write()` or `response.end()` are called before calling this, the implicit/mutable headers will be calculated and call this function.

When headers have been set with `response.setHeader()`, they will be merged with any headers passed to `response.writeHead()`, with the headers passed to `response.writeHead()` given precedence.

If this method is called and `response.setHeader()` has not been called, it will directly write the supplied header values onto the network channel without caching internally, and the `response.getHeader()` on the header will not yield the expected result. If progressive population of headers is desired with potential future retrieval and modification, use `response.setHeader()` instead.

```
// Returns content-type = text/plain  
const server = http.createServer((req, res) => {  
  res.setHeader('Content-Type', 'text/html');  
  res.setHeader('X-Foo', 'bar');  
  res.writeHead(200, { 'Content-Type': 'text/plain' });  
  res.end('ok');  
});
```

`Content-Length` is given in bytes, not characters. Use `Buffer.byteLength()` to determine the length of the body in bytes. Node.js does not check whether `Content-Length` and the length of the body which has been transmitted are equal or not.

Attempting to set a header field name or value that contains invalid characters will result in a `TypeError` being thrown.

`response.writeProcessing()`

Sends a HTTP/1.1 102 Processing message to the client, indicating that the request body should be sent.

Class: `http.IncomingMessage`

- Extends: `<stream.Readable>`

An `IncomingMessage` object is created by `http.Server` or `http.ClientRequest` and passed as the first argument to the `'request'` and `'response'` event respectively. It may be used to access response status, headers and data.

Different from its `socket` value which is a subclass of `<stream.Duplex>`, the `IncomingMessage` itself extends `<stream.Readable>` and is created separately to parse and emit the incoming HTTP headers and payload, as the underlying socket may be reused multiple times in case of keep-alive.

Event: `'aborted'`

Emitted when the request has been aborted.

Event: `'close'`

Indicates that the underlying connection was closed.

`message.aborted`

- `<boolean>`

The `message.aborted` property will be `true` if the request has been aborted.

message.complete

- `<boolean>`

The `message.complete` property will be `true` if a complete HTTP message has been received and successfully parsed.

This property is particularly useful as a means of determining if a client or server fully transmitted a message before a connection was terminated:

```
const req = http.request({
  host: '127.0.0.1',
  port: 8080,
  method: 'POST'
}, (res) => {
  res.resume();
  res.on('end', () => {
    if (!res.complete)
      console.error(
        'The connection was terminated while the message was still being sent');
  });
});
```

message.connection

Stability: 0 - Deprecated. Use `message.socket`.

Alias for `message.socket`.

message.destroy([error])

- `error` `<Error>`
- Returns: `<this>`

Calls `destroy()` on the socket that received the `IncomingMessage`. If `error` is provided, an `'error'` event is emitted on the socket and `error` is passed as an argument to any listeners on the event.

message.headers

- `<Object>`

The request/response headers object.

Key-value pairs of header names and values. Header names are lower-cased.

```
// Prints something like:
//
// { 'user-agent': 'curl/7.22.0',
//   host: '127.0.0.1:8000',
//   accept: '*/*' }
console.log(request.headers);
```

Duplicates in raw headers are handled in the following ways, depending on the header name:

- Duplicates of `age`, `authorization`, `content-length`, `content-type`, `etag`, `expires`, `from`, `host`, `if-modified-since`, `if-unmodified-since`, `last-modified`, `location`, `max-forwards`, `proxy-authorization`, `referer`, `retry-after`, `server`, or `user-agent` are discarded.
- `set-cookie` is always an array. Duplicates are added to the array.
- For duplicate `cookie` headers, the values are joined together with `;`.
- For all other headers, the values are joined together with `,`.

message.httpVersion

- `<string>`

In case of server request, the HTTP version sent by the client. In the case of client response, the HTTP version of the connected-to server. Probably either `'1.1'` or `'1.0'`.

Also `message.httpVersionMajor` is the first integer and `message.httpVersionMinor` is the second.

message.method

- `<string>`

Only valid for request obtained from `http.Server`.

The request method as a string. Read only. Examples: `'GET'`, `'DELETE'`.

message.rawHeaders

- `<string[]>`

The raw request/response headers list exactly as they were received.

The keys and values are in the same list. It is *not* a list of tuples. So, the even-numbered offsets are key values, and the odd-numbered offsets are the associated values.

Header names are not lowercased, and duplicates are not merged.

```
// Prints something like:
//
// [ 'user-agent',
//   'this is invalid because there can be only one',
//   'User-Agent',
//   'curl/7.22.0',
//   'Host',
//   '127.0.0.1:8000',
//   'ACCEPT',
//   '*/*' ]
console.log(request.rawHeaders);
```

message.rawTrailers

- `<string[]>`

The raw request/response trailer keys and values exactly as they were received. Only populated at the `'end'` event.

message.setTimeout(msecs[, callback])

- `msecs` `<number>`

- `callback` `<Function>`
- Returns: `<http.IncomingMessage>`

Calls `message.socket.setTimeout(msecs, callback)`.

`message.socket`

- `<stream.Duplex>`

The `net.Socket` object associated with the connection.

With HTTPS support, use `request.socket.getPeerCertificate()` to obtain the client's authentication details.

This property is guaranteed to be an instance of the `<net.Socket>` class, a subclass of `<stream.Duplex>`, unless the user specified a socket type other than `<net.Socket>`.

`message.statusCode`

- `<number>`

Only valid for response obtained from `http.ClientRequest`.

The 3-digit HTTP response status code. E.G. `404`.

`message.statusMessage`

- `<string>`

Only valid for response obtained from `http.ClientRequest`.

The HTTP response status message (reason phrase). E.G. `OK` or `Internal Server Error`.

`message.trailers`

- `<Object>`

The request/response trailers object. Only populated at the `'end'` event.

`message.url`

- `<string>`

Only valid for request obtained from `http.Server`.

Request URL string. This contains only the URL that is present in the actual HTTP request. Take the following request:

```
GET /status?name=ryan HTTP/1.1
Accept: text/plain
```

To parse the URL into its parts:

```
new URL(request.url, `http://${request.headers.host}`);
```

When `request.url` is `'/status?name=ryan'` and `request.headers.host` is `'localhost:3000'`:

```
$ node
> new URL(request.url, `http://${request.headers.host}`)
```

```

URL {
  href: 'http://localhost:3000/status?name=ryan',
  origin: 'http://localhost:3000',
  protocol: 'http:',
  username: '',
  password: '',
  host: 'localhost:3000',
  hostname: 'localhost',
  port: '3000',
  pathname: '/status',
  search: '?name=ryan',
  searchParams: URLSearchParams { 'name' => 'ryan' },
  hash: ''
}

```

Class: `http.OutgoingMessage`

- Extends: `<Stream>`

This class serves as the parent class of `http.ClientRequest` and `http.ServerResponse`. It is an abstract of outgoing message from the perspective of the participants of HTTP transaction.

Event: `drain`

Emitted when the buffer of the message is free again.

Event: `finish`

Emitted when the transmission is finished successfully.

Event: `prefinish`

Emitted when `outgoingMessage.end` was called. When the event is emitted, all data has been processed but not necessarily completely flushed.

`outgoingMessage.addTrailers(headers)`

- `headers` `<Object>`

Adds HTTP trailers (headers but at the end of the message) to the message.

Trailers are **only** be emitted if the message is chunked encoded. If not, the trailer will be silently discarded.

HTTP requires the `Trailer` header to be sent to emit trailers, with a list of header fields in its value, e.g.

```

message.writeHead(200, { 'Content-Type': 'text/plain',
                        'Trailer': 'Content-MD5' });

message.write(fileData);

message.addTrailers({ 'Content-MD5': '7895bf4b8828b55ceaf47747b4bca667' });

message.end();

```

Attempting to set a header field name or value that contains invalid characters will result in a `TypeError` being thrown.

outgoingMessage.connection

Stability: 0 - Deprecated: Use `outgoingMessage.socket` instead.

Aliases of `outgoingMessage.socket`

outgoingMessage.cork()

See `writable.cork()`.

outgoingMessage.destroy([error])

- `error` `<Error>` Optional, an error to emit with `error` event
- Returns: `<this>`

Destroys the message. Once a socket is associated with the message and is connected, that socket will be destroyed as well.

outgoingMessage.end(chunk[, encoding][, callback])

- `chunk` `<string>` | `<Buffer>`
- `encoding` `<string>` Optional, Default: `utf8`
- `callback` `<Function>` Optional
- Returns: `<this>`

Finishes the outgoing message. If any parts of the body are unsent, it will flush them to the underlying system. If the message is chunked, it will send the terminating chunk `0\r\n\r\n`, and send the trailer (if any).

If `chunk` is specified, it is equivalent to call `outgoingMessage.write(chunk, encoding)`, followed by `outgoingMessage.end(callback)`.

If `callback` is provided, it will be called when the message is finished. (equivalent to the callback to event `finish`)

outgoingMessage.flushHeaders()

Compulsorily flushes the message headers

For efficiency reason, Node.js normally buffers the message headers until `outgoingMessage.end()` is called or the first chunk of message data is written. It then tries to pack the headers and data into a single TCP packet.

It is usually desired (it saves a TCP round-trip), but not when the first data is not sent until possibly much later. `outgoingMessage.flushHeaders()` bypasses the optimization and kickstarts the request.

outgoingMessage.getHeader(name)

- `name` `<string>` Name of header
- Returns `<string>` | `<undefined>`

Gets the value of HTTP header with the given name. If such a name doesn't exist in message, it will be `undefined`.

outgoingMessage.getHeaderNames()

- Returns `<string[]>`

Returns an array of names of headers of the outgoing `outgoingMessage`. All names are lowercase.

outgoingMessage.getHeaders()

- Returns: `<Object>`

Returns a shallow copy of the current outgoing headers. Since a shallow copy is used, array values may be mutated without additional calls to various header-related HTTP module methods. The keys of the returned object are the header names and the values are the respective header values. All header names are lowercase.

The object returned by the `outgoingMessage.getHeaders()` method does not prototypically inherit from the JavaScript Object. This means that typical Object methods such as `obj.toString()`, `obj.hasOwnProperty()`, and others are not defined and will not work.

```
outgoingMessage.setHeader('Foo', 'bar');
outgoingMessage.setHeader('Set-Cookie', ['foo=bar', 'bar=baz']);

const headers = outgoingMessage.getHeaders();
// headers === { foo: 'bar', 'set-cookie': ['foo=bar', 'bar=baz'] }
```

outgoingMessage.hasHeader(name)

- `name` `<string>`
- Returns `<boolean>`

Returns `true` if the header identified by `name` is currently set in the outgoing headers. The header name is case-insensitive.

```
const hasContentType = outgoingMessage.hasHeader('content-type');
```

outgoingMessage.headersSent

- `<boolean>`

Read-only. `true` if the headers were sent, otherwise `false`.

outgoingMessage.pipe()

Overrides the pipe method of legacy `Stream` which is the parent class of `http.outgoingMessage`.

Since `OutgoingMessage` should be a write-only stream, call this function will throw an `Error`. Thus, it disabled the pipe method it inherits from `Stream`.

The User should not call this function directly.

outgoingMessage.removeHeader()

Removes a header that is queued for implicit sending.

```
outgoingMessage.removeHeader('Content-Encoding');
```

outgoingMessage.setHeader(name, value)

- `name` `<string>` Header name
- `value` `<string>` Header value
- Returns: `<this>`

Sets a single header value for the header object.

outgoingMessage.setTimeout(msesc[, callback])

- `msesc` `<number>`
- `callback` `<Function>` Optional function to be called when a timeout

occurs, Same as binding to the `timeout` event.

- Returns: `<this>`

Once a socket is associated with the message and is connected, `socket.setTimeout()` will be called with `msecs` as the first parameter.

outgoingMessage.socket

- `<stream.Duplex>`

Reference to the underlying socket. Usually, users will not want to access this property.

After calling `outgoingMessage.end()`, this property will be nulled.

outgoingMessage.uncork()

See `writable.uncork()`

outgoingMessage.writableCorked

- `<number>`

This `outgoingMessage.writableCorked` will return the time how many `outgoingMessage.cork()` have been called.

outgoingMessage.writableEnded

- `<boolean>`

Readonly, `true` if `outgoingMessage.end()` has been called. Noted that this property does not reflect whether the data has been flush. For that purpose, use `message.writableFinished` instead.

outgoingMessage.writableFinished

- `<boolean>`

Readonly, `true` if all data has been flushed to the underlying system.

outgoingMessage.writableHighWaterMark

- `<number>`

This `outgoingMessage.writableHighWaterMark` will be the `highWaterMark` of underlying socket if socket exists. Else, it would be the default `highWaterMark`.

`highWaterMark` is the maximum amount of data that can be potentially buffered by the socket.

outgoingMessage.writableLength

- `<number>`

Readonly, This `outgoingMessage.writableLength` contains the number of bytes (or objects) in the buffer ready to send.

outgoingMessage.writableObjectMode

- `<boolean>`

Readonly, always returns `false`.

outgoingMessage.write(chunk[, encoding][, callback])

- `chunk` `<string> | <Buffer>`
- `encoding` `<string>` **Default:** `utf8`
- `callback` `<Function>`
- Returns `<boolean>`

If this method is called and the header is not sent, it will call `this._implicitHeader` to flush implicit header. If the message should not have a body (indicated by `this._hasBody`), the call is ignored and `chunk` will not be sent. It could be useful when handling a particular message which must not include a body. e.g. response to `HEAD` request, `204` and `304` response.

`chunk` can be a string or a buffer. When `chunk` is a string, the `encoding` parameter specifies how to encode `chunk` into a byte stream. `callback` will be called when the `chunk` is flushed.

If the message is transferred in chunked encoding (indicated by `this.chunkedEncoding`), `chunk` will be flushed as one chunk among a stream of chunks. Otherwise, it will be flushed as the body of message.

This method handles the raw body of the HTTP message and has nothing to do with higher-level multi-part body encodings that may be used.

If it is the first call to this method of a message, it will send the buffered header first, then flush the `chunk` as described above.

The second and successive calls to this method will assume the data will be streamed and send the new data separately. It means that the response is buffered up to the first chunk of the body.

Returns `true` if the entire data was flushed successfully to the kernel buffer. Returns `false` if all or part of the data was queued in the user memory. Event `drain` will be emitted when the buffer is free again.

http.METHODS

- `<string[]>`

A list of the HTTP methods that are supported by the parser.

http.STATUS_CODES

- `<Object>`

A collection of all the standard HTTP response status codes, and the short description of each. For example, `http.STATUS_CODES[404] === 'Not Found'`.

http.createServer([options][, requestListener])

- `options` `<Object>`
 - `IncomingMessage` `<http.IncomingMessage>` Specifies the `IncomingMessage` class to be used. Useful for extending the original `IncomingMessage`. **Default:** `IncomingMessage`.
 - `ServerResponse` `<http.ServerResponse>` Specifies the `ServerResponse` class to be used. Useful for extending the original `ServerResponse`. **Default:** `ServerResponse`.
 - `insecureHTTPParser` `<boolean>` Use an insecure HTTP parser that accepts invalid HTTP headers when `true`. Using the insecure parser should be avoided. See `--insecure-http-parser` for more information. **Default:** `false`
 - `maxHeaderSize` `<number>` Optionally overrides the value of `--max-http-header-size` for requests received by this server, i.e. the maximum length of request headers in bytes. **Default:** `16384` (16 KB).
- `requestListener` `<Function>`

- Returns: `<http.Server>`

Returns a new instance of `http.Server` .

The `requestListener` is a function which is automatically added to the `'request'` event.

```
const http = require('http');

// Create a local server to receive data from
const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'application/json' });
  res.end(JSON.stringify({
    data: 'Hello World!'
  }));
});

server.listen(8000);

const http = require('http');

// Create a local server to receive data from
const server = http.createServer();

// Listen to the request event
server.on('request', (request, res) => {
  res.writeHead(200, { 'Content-Type': 'application/json' });
  res.end(JSON.stringify({
    data: 'Hello World!'
  }));
});

server.listen(8000);
```

http.get(options[, callback])

http.get(url[, options][, callback])

- `url` `<string>` | `<URL>`
- `options` `<Object>` Accepts the same `options` as `http.request()` , with the `method` always set to `GET` . Properties that are inherited from the prototype are ignored.
- `callback` `<Function>`
- Returns: `<http.ClientRequest>`

Since most requests are GET requests without bodies, Node.js provides this convenience method. The only difference between this method and `http.request()` is that it sets the method to GET and calls `req.end()` automatically. The callback must take care to consume the response data for reasons stated in `http.ClientRequest` section.

The `callback` is invoked with a single argument that is an instance of `http.IncomingMessage` .

JSON fetching example:

```
http.get('http://localhost:8000/', (res) => {
  const { statusCode } = res;
  const contentType = res.headers['content-type'];

  let error;
  // Any 2xx status code signals a successful response but
  // here we're only checking for 200.
  if (statusCode !== 200) {
    error = new Error('Request Failed.\n' +
      `Status Code: ${statusCode}`);
  } else if (!/^application\/json/.test(contentType)) {
    error = new Error('Invalid content-type.\n' +
      `Expected application/json but received ${contentType}`);
  }
  if (error) {
    console.error(error.message);
    // Consume response data to free up memory
    res.resume();
    return;
  }

  res.setEncoding('utf8');
  let rawData = '';
  res.on('data', (chunk) => { rawData += chunk; });
  res.on('end', () => {
    try {
      const parsedData = JSON.parse(rawData);
      console.log(parsedData);
    } catch (e) {
      console.error(e.message);
    }
  });
}).on('error', (e) => {
  console.error(`Got error: ${e.message}`);
});

// Create a local server to receive data from
const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'application/json' });
  res.end(JSON.stringify({
    data: 'Hello World!'
  }));
});

server.listen(8000);
```

http.globalAgent

- `<http.Agent>`

Global instance of `Agent` which is used as the default for all HTTP client requests.

http.maxHeaderSize

- `<number>`

Read-only property specifying the maximum allowed size of HTTP headers in bytes. Defaults to 8 KB. Configurable using the `--max-http-header-size` CLI option.

This can be overridden for servers and client requests by passing the `maxHeaderSize` option.

http.request(options[, callback])

http.request(url[, options][, callback])

- `url` `<string>` | `<URL>`
- `options` `<Object>`
 - `agent` `<http.Agent>` | `<boolean>` Controls `Agent` behavior. Possible values:
 - `undefined` (default): use `http.globalAgent` for this host and port.
 - `Agent` object: explicitly use the passed in `Agent`.
 - `false`: causes a new `Agent` with default values to be used.
 - `auth` `<string>` Basic authentication i.e. `'user:password'` to compute an Authorization header.
 - `createConnection` `<Function>` A function that produces a socket/stream to use for the request when the `agent` option is not used. This can be used to avoid creating a custom `Agent` class just to override the default `createConnection` function. See `agent.createConnection()` for more details. Any `Duplex` stream is a valid return value.
 - `defaultPort` `<number>` Default port for the protocol. **Default:** `agent.defaultPort` if an `Agent` is used, else `undefined`.
 - `family` `<number>` IP address family to use when resolving `host` or `hostname`. Valid values are `4` or `6`. When unspecified, both IP v4 and v6 will be used.
 - `headers` `<Object>` An object containing request headers.
 - `hints` `<number>` Optional `dns.lookup()` hints.
 - `host` `<string>` A domain name or IP address of the server to issue the request to. **Default:** `'localhost'`.
 - `hostname` `<string>` Alias for `host`. To support `url.parse()`, `hostname` will be used if both `host` and `hostname` are specified.
 - `insecureHTTPParser` `<boolean>` Use an insecure HTTP parser that accepts invalid HTTP headers when `true`. Using the insecure parser should be avoided. See `--insecure-http-parser` for more information. **Default:** `false`
 - `localAddress` `<string>` Local interface to bind for network connections.
 - `localPort` `<number>` Local port to connect from.
 - `lookup` `<Function>` Custom lookup function. **Default:** `dns.lookup()`.
 - `maxHeaderSize` `<number>` Optionally overrides the value of `--max-http-header-size` for requests received from the server, i.e. the maximum length of response headers in bytes. **Default:** 16384 (16 KB).
 - `method` `<string>` A string specifying the HTTP request method. **Default:** `'GET'`.
 - `path` `<string>` Request path. Should include query string if any. E.G. `'/index.html?page=12'`. An exception is thrown when the request path contains illegal characters. Currently, only spaces are rejected but that may change in the future. **Default:** `'/'`.
 - `port` `<number>` Port of remote server. **Default:** `defaultPort` if set, else `80`.
 - `protocol` `<string>` Protocol to use. **Default:** `'http:'`.

- `setHost` `<boolean>` : Specifies whether or not to automatically add the `Host` header. Defaults to `true`.
- `socketPath` `<string>` Unix Domain Socket (cannot be used if one of `host` or `port` is specified, those specify a TCP Socket).
- `timeout` `<number>` : A number specifying the socket timeout in milliseconds. This will set the timeout before the socket is connected.
- `signal` `<AbortSignal>` : An `AbortSignal` that may be used to abort an ongoing request.
- `callback` `<Function>`
- Returns: `<http.ClientRequest>`

Node.js maintains several connections per server to make HTTP requests. This function allows one to transparently issue requests.

`url` can be a string or a `URL` object. If `url` is a string, it is automatically parsed with `new URL()`. If it is a `URL` object, it will be automatically converted to an ordinary `options` object.

If both `url` and `options` are specified, the objects are merged, with the `options` properties taking precedence.

The optional `callback` parameter will be added as a one-time listener for the `'response'` event.

`http.request()` returns an instance of the `http.ClientRequest` class. The `ClientRequest` instance is a writable stream. If one needs to upload a file with a POST request, then write to the `ClientRequest` object.

```
const http = require('http');

const postData = JSON.stringify({
  'msg': 'Hello World!'
});

const options = {
  hostname: 'www.google.com',
  port: 80,
  path: '/upload',
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
    'Content-Length': Buffer.byteLength(postData)
  }
};

const req = http.request(options, (res) => {
  console.log(`STATUS: ${res.statusCode}`);
  console.log(`HEADERS: ${JSON.stringify(res.headers)}`);
  res.setEncoding('utf8');
  res.on('data', (chunk) => {
    console.log(`BODY: ${chunk}`);
  });
  res.on('end', () => {
    console.log('No more data in response.');
```

```
// Write data to request body
req.write(postData);
req.end();
```

In the example `req.end()` was called. With `http.request()` one must always call `req.end()` to signify the end of the request - even if there is no data being written to the request body.

If any error is encountered during the request (be that with DNS resolution, TCP level errors, or actual HTTP parse errors) an `'error'` event is emitted on the returned request object. As with all `'error'` events, if no listeners are registered the error will be thrown.

There are a few special headers that should be noted.

- Sending a `'Connection: keep-alive'` will notify Node.js that the connection to the server should be persisted until the next request.
- Sending a `'Content-Length'` header will disable the default chunked encoding.
- Sending an `'Expect'` header will immediately send the request headers. Usually, when sending `'Expect: 100-continue'`, both a timeout and a listener for the `'continue'` event should be set. See RFC 2616 Section 8.2.3 for more information.
- Sending an Authorization header will override using the `auth` option to compute basic authentication.

Example using a `URL` as `options`:

```
const options = new URL('http://abc:xyz@example.com');

const req = http.request(options, (res) => {
  // ...
});
```

In a successful request, the following events will be emitted in the following order:

- `'socket'`
- `'response'`
 - `'data'` any number of times, on the `res` object (`'data'` will not be emitted at all if the response body is empty, for instance, in most redirects)
 - `'end'` on the `res` object
- `'close'`

In the case of a connection error, the following events will be emitted:

- `'socket'`
- `'error'`
- `'close'`

In the case of a premature connection close before the response is received, the following events will be emitted in the following order:

- `'socket'`
- `'error'` with an error with message `'Error: socket hang up'` and code `'ECONNRESET'`
- `'close'`

In the case of a premature connection close after the response is received, the following events will be emitted in the following order:

- `'socket'`

- 'response'
 - 'data' any number of times, on the `res` object
- (connection closed here)
- 'aborted' on the `res` object
- 'error' on the `res` object with an error with message 'Error: aborted' and code 'ECONNRESET'.
- 'close'
- 'close' on the `res` object

If `req.destroy()` is called before a socket is assigned, the following events will be emitted in the following order:

- (`req.destroy()` called here)
- 'error' with an error with message 'Error: socket hang up' and code 'ECONNRESET'
- 'close'

If `req.destroy()` is called before the connection succeeds, the following events will be emitted in the following order:

- 'socket'
- (`req.destroy()` called here)
- 'error' with an error with message 'Error: socket hang up' and code 'ECONNRESET'
- 'close'

If `req.destroy()` is called after the response is received, the following events will be emitted in the following order:

- 'socket'
- 'response'
 - 'data' any number of times, on the `res` object
- (`req.destroy()` called here)
- 'aborted' on the `res` object
- 'error' on the `res` object with an error with message 'Error: aborted' and code 'ECONNRESET'.
- 'close'
- 'close' on the `res` object

If `req.abort()` is called before a socket is assigned, the following events will be emitted in the following order:

- (`req.abort()` called here)
- 'abort'
- 'close'

If `req.abort()` is called before the connection succeeds, the following events will be emitted in the following order:

- 'socket'
- (`req.abort()` called here)
- 'abort'
- 'error' with an error with message 'Error: socket hang up' and code 'ECONNRESET'
- 'close'

If `req.abort()` is called after the response is received, the following events will be emitted in the following order:

- 'socket'
- 'response'
 - 'data' any number of times, on the `res` object

- `(req.abort())` called here)
- `'abort'`
- `'aborted'` on the `res` object
- `'error'` on the `res` object with an error with message `'Error: aborted'` and code `'ECONNRESET'`.
- `'close'`
- `'close'` on the `res` object

Setting the `timeout` option or using the `setTimeout()` function will not abort the request or do anything besides add a `'timeout'` event.

Passing an `AbortSignal` and then calling `abort` on the corresponding `AbortController` will behave the same way as calling `.destroy()` on the request itself.

http.validateHeaderName(name)

- `name` `<string>`

Performs the low-level validations on the provided `name` that are done when `res.setHeader(name, value)` is called.

Passing illegal value as `name` will result in a `TypeError` being thrown, identified by `code: 'ERR_INVALID_HTTP_TOKEN'`.

It is not necessary to use this method before passing headers to an HTTP request or response. The HTTP module will automatically validate such headers. Examples:

Example:

```
const { validateHeaderName } = require('http');

try {
  validateHeaderName('');
} catch (err) {
  err instanceof TypeError; // --> true
  err.code; // --> 'ERR_INVALID_HTTP_TOKEN'
  err.message; // --> 'Header name must be a valid HTTP token ["]'
```

http.validateHeaderValue(name, value)

- `name` `<string>`
- `value` `<any>`

Performs the low-level validations on the provided `value` that are done when `res.setHeader(name, value)` is called.

Passing illegal value as `value` will result in a `TypeError` being thrown.

- Undefined value error is identified by `code: 'ERR_HTTP_INVALID_HEADER_VALUE'`.
- Invalid value character error is identified by `code: 'ERR_INVALID_CHAR'`.

It is not necessary to use this method before passing headers to an HTTP request or response. The HTTP module will automatically validate such headers.

Examples:

```
const { validateHeaderValue } = require('http');

try {
  validateHeaderValue('x-my-header', undefined);
} catch (err) {
  err instanceof TypeError; // --> true
  err.code === 'ERR_HTTP_INVALID_HEADER_VALUE'; // --> true
  err.message; // --> 'Invalid value "undefined" for header "x-my-header"'
}

try {
  validateHeaderValue('x-my-header', 'oumīgə');
} catch (err) {
  err instanceof TypeError; // --> true
  err.code === 'ERR_INVALID_CHAR'; // --> true
  err.message; // --> 'Invalid character in header content ["x-my-header"]'
}
```

HTTP/2

Stability: 2 - Stable

Source Code: [lib/http2.js](#)

The `http2` module provides an implementation of the [HTTP/2](#) protocol. It can be accessed using:

```
const http2 = require('http2');
```

Core API

The Core API provides a low-level interface designed specifically around support for HTTP/2 protocol features. It is specifically *not* designed for compatibility with the existing [HTTP/1](#) module API. However, the [Compatibility API](#) is.

The `http2` Core API is much more symmetric between client and server than the `http` API. For instance, most events, like `'error'`, `'connect'` and `'stream'`, can be emitted either by client-side code or server-side code.

Server-side example

The following illustrates a simple HTTP/2 server using the Core API. Since there are no browsers known that support [unencrypted HTTP/2](#), the use of `http2.createSecureServer()` is necessary when communicating with browser clients.

```
const http2 = require('http2');
const fs = require('fs');

const server = http2.createSecureServer({
  key: fs.readFileSync('localhost-privkey.pem'),
  cert: fs.readFileSync('localhost-cert.pem')
```

```

});
server.on('error', (err) => console.error(err));

server.on('stream', (stream, headers) => {
  // stream is a Duplex
  stream.respond({
    'content-type': 'text/html; charset=utf-8',
    ':status': 200
  });
  stream.end('<h1>Hello World</h1>');
});

server.listen(8443);

```

To generate the certificate and key for this example, run:

```

openssl req -x509 -newkey rsa:2048 -nodes -sha256 -subj '/CN=localhost' \
  -keyout localhost-privkey.pem -out localhost-cert.pem

```

Client-side example

The following illustrates an HTTP/2 client:

```

const http2 = require('http2');
const fs = require('fs');
const client = http2.connect('https://localhost:8443', {
  ca: fs.readFileSync('localhost-cert.pem')
});
client.on('error', (err) => console.error(err));

const req = client.request({ ':path': '/' });

req.on('response', (headers, flags) => {
  for (const name in headers) {
    console.log(`${name}: ${headers[name]}`);
  }
});

req.setEncoding('utf8');
let data = '';
req.on('data', (chunk) => { data += chunk; });
req.on('end', () => {
  console.log(`\n${data}`);
  client.close();
});
req.end();

```

Class: Http2Session

- Extends: `<EventEmitter>`

Instances of the `http2.Http2Session` class represent an active communications session between an HTTP/2 client and server. Instances of this class are *not* intended to be constructed directly by user code.

Each `Http2Session` instance will exhibit slightly different behaviors depending on whether it is operating as a server or a client. The `http2session.type` property can be used to determine the mode in which an `Http2Session` is operating. On the server side, user code should rarely have occasion to work with the `Http2Session` object directly, with most actions typically taken through interactions with either the `Http2Server` or `Http2Stream` objects.

User code will not create `Http2Session` instances directly. Server-side `Http2Session` instances are created by the `Http2Server` instance when a new HTTP/2 connection is received. Client-side `Http2Session` instances are created using the `http2.connect()` method.

Http2Session and sockets

Every `Http2Session` instance is associated with exactly one `net.Socket` or `tls.TLSSocket` when it is created. When either the `Socket` or the `Http2Session` are destroyed, both will be destroyed.

Because of the specific serialization and processing requirements imposed by the HTTP/2 protocol, it is not recommended for user code to read data from or write data to a `Socket` instance bound to a `Http2Session`. Doing so can put the HTTP/2 session into an indeterminate state causing the session and the socket to become unusable.

Once a `Socket` has been bound to an `Http2Session`, user code should rely solely on the API of the `Http2Session`.

Event: 'close'

The `'close'` event is emitted once the `Http2Session` has been destroyed. Its listener does not expect any arguments.

Event: 'connect'

- `session` `<Http2Session>`
- `socket` `<net.Socket>`

The `'connect'` event is emitted once the `Http2Session` has been successfully connected to the remote peer and communication may begin.

User code will typically not listen for this event directly.

Event: 'error'

- `error` `<Error>`

The `'error'` event is emitted when an error occurs during the processing of an `Http2Session`.

Event: 'frameError'

- `type` `<integer>` The frame type.
- `code` `<integer>` The error code.
- `id` `<integer>` The stream id (or `0` if the frame isn't associated with a stream).

The `'frameError'` event is emitted when an error occurs while attempting to send a frame on the session. If the frame that could not be sent is associated with a specific `Http2Stream`, an attempt to emit a `'frameError'` event on the `Http2Stream` is made.

If the `'frameError'` event is associated with a stream, the stream will be closed and destroyed immediately following the `'frameError'` event. If the event is not associated with a stream, the `Http2Session` will be shut down immediately following the `'frameError'` event.

Event: 'goaway'

- `errorCode` `<number>` The HTTP/2 error code specified in the `GOAWAY` frame.

- `lastStreamID` `<number>` The ID of the last stream the remote peer successfully processed (or `0` if no ID is specified).
- `opaqueData` `<Buffer>` If additional opaque data was included in the `GOAWAY` frame, a `Buffer` instance will be passed containing that data.

The `'goaway'` event is emitted when a `GOAWAY` frame is received.

The `Http2Session` instance will be shut down automatically when the `'goaway'` event is emitted.

Event: 'localSettings'

- `settings` `<HTTP/2 Settings Object>` A copy of the `SETTINGS` frame received.

The `'localSettings'` event is emitted when an acknowledgment `SETTINGS` frame has been received.

When using `http2session.settings()` to submit new settings, the modified settings do not take effect until the `'localSettings'` event is emitted.

```
session.settings({ enablePush: false });

session.on('localSettings', (settings) => {
  /* Use the new settings */
});
```

Event: 'ping'

- `payload` `<Buffer>` The `PING` frame 8-byte payload

The `'ping'` event is emitted whenever a `PING` frame is received from the connected peer.

Event: 'remoteSettings'

- `settings` `<HTTP/2 Settings Object>` A copy of the `SETTINGS` frame received.

The `'remoteSettings'` event is emitted when a new `SETTINGS` frame is received from the connected peer.

```
session.on('remoteSettings', (settings) => {
  /* Use the new settings */
});
```

Event: 'stream'

- `stream` `<Http2Stream>` A reference to the stream
- `headers` `<HTTP/2 Headers Object>` An object describing the headers
- `flags` `<number>` The associated numeric flags
- `rawHeaders` `<Array>` An array containing the raw header names followed by their respective values.

The `'stream'` event is emitted when a new `Http2Stream` is created.

```
const http2 = require('http2');
session.on('stream', (stream, headers, flags) => {
  const method = headers[':method'];
  const path = headers[':path'];
  // ...
  stream.respond({
```

```

    ':status': 200,
    'content-type': 'text/plain; charset=utf-8'
  });
  stream.write('hello ');
  stream.end('world');
});

```

On the server side, user code will typically not listen for this event directly, and would instead register a handler for the `'stream'` event emitted by the `net.Server` or `tls.Server` instances returned by `http2.createServer()` and `http2.createSecureServer()`, respectively, as in the example below:

```

const http2 = require('http2');

// Create an unencrypted HTTP/2 server
const server = http2.createServer();

server.on('stream', (stream, headers) => {
  stream.respond({
    'content-type': 'text/html; charset=utf-8',
    ':status': 200
  });
  stream.on('error', (error) => console.error(error));
  stream.end('<h1>Hello World</h1>');
});

server.listen(80);

```

Even though HTTP/2 streams and network sockets are not in a 1:1 correspondence, a network error will destroy each individual stream and must be handled on the stream level, as shown above.

Event: 'timeout'

After the `http2session.setTimeout()` method is used to set the timeout period for this `Http2Session`, the `'timeout'` event is emitted if there is no activity on the `Http2Session` after the configured number of milliseconds. Its listener does not expect any arguments.

```

session.setTimeout(2000);
session.on('timeout', () => { /* .. */ });

```

http2session.alpnProtocol

- `<string> | <undefined>`

Value will be `undefined` if the `Http2Session` is not yet connected to a socket, `h2c` if the `Http2Session` is not connected to a `TLSocket`, or will return the value of the connected `TLSocket`'s own `alpnProtocol` property.

http2session.close([callback])

- `callback` `<Function>`

Gracefully closes the `Http2Session`, allowing any existing streams to complete on their own and preventing new `Http2Stream` instances from being created. Once closed, `http2session.destroy()` *might* be called if there are no open `Http2Stream` instances.

If specified, the `callback` function is registered as a handler for the `'close'` event.

`http2session.closed`

- `<boolean>`

Will be `true` if this `Http2Session` instance has been closed, otherwise `false`.

`http2session.connecting`

- `<boolean>`

Will be `true` if this `Http2Session` instance is still connecting, will be set to `false` before emitting `connect` event and/or calling the `http2.connect` callback.

`http2session.destroy([error][, code])`

- `error` `<Error>` An `Error` object if the `Http2Session` is being destroyed due to an error.
- `code` `<number>` The HTTP/2 error code to send in the final `GOAWAY` frame. If unspecified, and `error` is not undefined, the default is `INTERNAL_ERROR`, otherwise defaults to `NO_ERROR`.

Immediately terminates the `Http2Session` and the associated `net.Socket` or `tls.TLSSocket`.

Once destroyed, the `Http2Session` will emit the `'close'` event. If `error` is not undefined, an `'error'` event will be emitted immediately before the `'close'` event.

If there are any remaining open `Http2Streams` associated with the `Http2Session`, those will also be destroyed.

`http2session.destroyed`

- `<boolean>`

Will be `true` if this `Http2Session` instance has been destroyed and must no longer be used, otherwise `false`.

`http2session.encrypted`

- `<boolean> | <undefined>`

Value is `undefined` if the `Http2Session` session socket has not yet been connected, `true` if the `Http2Session` is connected with a `TLSSocket`, and `false` if the `Http2Session` is connected to any other kind of socket or stream.

`http2session.goaway([code[, lastStreamID[, opaqueData]]])`

- `code` `<number>` An HTTP/2 error code
- `lastStreamID` `<number>` The numeric ID of the last processed `Http2Stream`
- `opaqueData` `<Buffer> | <TypedArray> | <DataView>` A `TypedArray` or `DataView` instance containing additional data to be carried within the `GOAWAY` frame.

Transmits a `GOAWAY` frame to the connected peer *without* shutting down the `Http2Session`.

`http2session.localSettings`

- `<HTTP/2 Settings Object>`

A prototype-less object describing the current local settings of this `Http2Session`. The local settings are local to *this* `Http2Session` instance.

`http2session.originSet`

- `<string[]> | <undefined>`

If the `Http2Session` is connected to a `TLSocket`, the `originSet` property will return an `Array` of origins for which the `Http2Session` may be considered authoritative.

The `originSet` property is only available when using a secure TLS connection.

`http2session.pendingSettingsAck`

- `<boolean>`

Indicates whether the `Http2Session` is currently waiting for acknowledgment of a sent `SETTINGS` frame. Will be `true` after calling the `http2session.settings()` method. Will be `false` once all sent `SETTINGS` frames have been acknowledged.

`http2session.ping([payload,]callback)`

- `payload` `<Buffer>` | `<TypedArray>` | `<DataView>` Optional ping payload.
- `callback` `<Function>`
- Returns: `<boolean>`

Sends a `PING` frame to the connected HTTP/2 peer. A `callback` function must be provided. The method will return `true` if the `PING` was sent, `false` otherwise.

The maximum number of outstanding (unacknowledged) pings is determined by the `maxOutstandingPings` configuration option. The default maximum is 10.

If provided, the `payload` must be a `Buffer`, `TypedArray`, or `DataView` containing 8 bytes of data that will be transmitted with the `PING` and returned with the ping acknowledgment.

The callback will be invoked with three arguments: an error argument that will be `null` if the `PING` was successfully acknowledged, a `duration` argument that reports the number of milliseconds elapsed since the ping was sent and the acknowledgment was received, and a `Buffer` containing the 8-byte `PING` payload.

```
session.ping(Buffer.from('abcdefgh'), (err, duration, payload) => {
  if (!err) {
    console.log(`Ping acknowledged in ${duration} milliseconds`);
    console.log(`With payload '${payload.toString()}'`);
  }
});
```

If the `payload` argument is not specified, the default payload will be the 64-bit timestamp (little endian) marking the start of the `PING` duration.

`http2session.ref()`

Calls `ref()` on this `Http2Session` instance's underlying `net.Socket`.

`http2session.remoteSettings`

- `<HTTP/2 Settings Object>`

A prototype-less object describing the current remote settings of this `Http2Session`. The remote settings are set by the *connected* HTTP/2 peer.

`http2session.setLocalWindowSize(windowSize)`

- `windowSize` `<number>`

Sets the local endpoint's window size. The `windowSize` is the total window size to set, not the delta.

```
const http2 = require('http2');

const server = http2.createServer();
const expectedWindowSize = 2 ** 20;
server.on('connect', (session) => {

  // Set local window size to be 2 ** 20
  session.setLocalWindowSize(expectedWindowSize);
});
```

`http2session.setTimeout(msecs, callback)`

- `msecs` `<number>`
- `callback` `<Function>`

Used to set a callback function that is called when there is no activity on the `Http2Session` after `msecs` milliseconds. The given `callback` is registered as a listener on the `'timeout'` event.

`http2session.socket`

- `<net.Socket>` | `<tls.TLSSocket>`

Returns a `Proxy` object that acts as a `net.Socket` (or `tls.TLSSocket`) but limits available methods to ones safe to use with HTTP/2.

`destroy`, `emit`, `end`, `pause`, `read`, `resume`, and `write` will throw an error with code `ERR_HTTP2_NO_SOCKET_MANIPULATION`. See [Http2Session and Sockets](#) for more information.

`setTimeout` method will be called on this `Http2Session`.

All other interactions will be routed directly to the socket.

`http2session.state`

Provides miscellaneous information about the current state of the `Http2Session`.

- `<Object>`
 - `effectiveLocalWindowSize` `<number>` The current local (receive) flow control window size for the `Http2Session`.
 - `effectiveRecvDataLength` `<number>` The current number of bytes that have been received since the last flow control `WINDOW_UPDATE`.
 - `nextStreamID` `<number>` The numeric identifier to be used the next time a new `Http2Stream` is created by this `Http2Session`.
 - `localWindowSize` `<number>` The number of bytes that the remote peer can send without receiving a `WINDOW_UPDATE`.
 - `lastProcStreamID` `<number>` The numeric id of the `Http2Stream` for which a `HEADERS` or `DATA` frame was most recently received.
 - `remoteWindowSize` `<number>` The number of bytes that this `Http2Session` may send without receiving a `WINDOW_UPDATE`.
 - `outboundQueueSize` `<number>` The number of frames currently within the outbound queue for this `Http2Session`.
 - `deflatedDynamicTableSize` `<number>` The current size in bytes of the outbound header compression state table.
 - `inflatedDynamicTableSize` `<number>` The current size in bytes of the inbound header compression state table.

An object describing the current status of this `Http2Session`.

`http2session.settings([settings], callback)`

- `settings` `<HTTP/2 Settings Object>`
- `callback` `<Function>` Callback that is called once the session is connected or right away if the session is already connected.
 - `err` `<Error> | <null>`
 - `settings` `<HTTP/2 Settings Object>` The updated `settings` object.
 - `duration` `<integer>`

Updates the current local settings for this `Http2Session` and sends a new `SETTINGS` frame to the connected HTTP/2 peer.

Once called, the `http2session.pendingSettingsAck` property will be `true` while the session is waiting for the remote peer to acknowledge the new settings.

The new settings will not become effective until the `SETTINGS` acknowledgment is received and the `'localSettings'` event is emitted. It is possible to send multiple `SETTINGS` frames while acknowledgment is still pending.

`http2session.type`

- `<number>`

The `http2session.type` will be equal to `http2.constants.NGHTTP2_SESSION_SERVER` if this `Http2Session` instance is a server, and `http2.constants.NGHTTP2_SESSION_CLIENT` if the instance is a client.

`http2session.unref()`

Calls `unref()` on this `Http2Session` instance's underlying `net.Socket`.

Class: `ServerHttp2Session`

- Extends: `<Http2Session>`

`serverhttp2session.altsvc(alt, originOrStream)`

- `alt` `<string>` A description of the alternative service configuration as defined by [RFC 7838](#).
- `originOrStream` `<number> | <string> | <URL> | <Object>` Either a URL string specifying the origin (or an `Object` with an `origin` property) or the numeric identifier of an active `Http2Stream` as given by the `http2stream.id` property.

Submits an `ALTSVC` frame (as defined by [RFC 7838](#)) to the connected client.

```
const http2 = require('http2');

const server = http2.createServer();
server.on('session', (session) => {
  // Set altsvc for origin https://example.org:80
  session.altsvc('h2=:8000', 'https://example.org:80');
});

server.on('stream', (stream) => {
  // Set altsvc for a specific stream
  stream.session.altsvc('h2=:8000', stream.id);
});
```

Sending an `ALTSVC` frame with a specific stream ID indicates that the alternate service is associated with the origin of the given `Http2Stream`.

The `alt` and origin string *must* contain only ASCII bytes and are strictly interpreted as a sequence of ASCII bytes. The special value `'clear'` may be passed to clear any previously set alternative service for a given domain.

When a string is passed for the `originOrStream` argument, it will be parsed as a URL and the origin will be derived. For instance, the origin for the HTTP URL `'https://example.org/foo/bar'` is the ASCII string `'https://example.org'`. An error will be thrown if either the given string cannot be parsed as a URL or if a valid origin cannot be derived.

A `URL` object, or any object with an `origin` property, may be passed as `originOrStream`, in which case the value of the `origin` property will be used. The value of the `origin` property *must* be a properly serialized ASCII origin.

Specifying alternative services

The format of the `alt` parameter is strictly defined by [RFC 7838](#) as an ASCII string containing a comma-delimited list of "alternative" protocols associated with a specific host and port.

For example, the value `'h2="example.org:81"'` indicates that the HTTP/2 protocol is available on the host `'example.org'` on TCP/IP port 81. The host and port *must* be contained within the quote (`"`) characters.

Multiple alternatives may be specified, for instance: `'h2="example.org:81", h2=":82"'`.

The protocol identifier (`'h2'` in the examples) may be any valid [ALPN Protocol ID](#).

The syntax of these values is not validated by the Node.js implementation and are passed through as provided by the user or received from the peer.

`serverhttp2session.origin(...origins)`

- `origins` `<string>` | `<URL>` | `<Object>` One or more URL Strings passed as separate arguments.

Submits an `ORIGIN` frame (as defined by [RFC 8336](#)) to the connected client to advertise the set of origins for which the server is capable of providing authoritative responses.

```
const http2 = require('http2');
const options = getSecureOptionsSomehow();
const server = http2.createSecureServer(options);
server.on('stream', (stream) => {
  stream.respond();
  stream.end('ok');
});
server.on('session', (session) => {
  session.origin('https://example.com', 'https://example.org');
});
```

When a string is passed as an `origin`, it will be parsed as a URL and the origin will be derived. For instance, the origin for the HTTP URL `'https://example.org/foo/bar'` is the ASCII string `'https://example.org'`. An error will be thrown if either the given string cannot be parsed as a URL or if a valid origin cannot be derived.

A `URL` object, or any object with an `origin` property, may be passed as an `origin`, in which case the value of the `origin` property will be used. The value of the `origin` property *must* be a properly serialized ASCII origin.

Alternatively, the `origins` option may be used when creating a new HTTP/2 server using the `http2.createSecureServer()` method:

```
const http2 = require('http2');
const options = getSecureOptionsSomehow();
```

```
options.origins = ['https://example.com', 'https://example.org'];
const server = http2.createSecureServer(options);
server.on('stream', (stream) => {
  stream.respond();
  stream.end('ok');
});
```

Class: ClientHttp2Session

- Extends: `<Http2Session>`

Event: 'altsvc'

- `alt` `<string>`
- `origin` `<string>`
- `streamId` `<number>`

The 'altsvc' event is emitted whenever an ALTSVC frame is received by the client. The event is emitted with the ALTSVC value, origin, and stream ID. If no `origin` is provided in the ALTSVC frame, `origin` will be an empty string.

```
const http2 = require('http2');
const client = http2.connect('https://example.org');

client.on('altsvc', (alt, origin, streamId) => {
  console.log(alt);
  console.log(origin);
  console.log(streamId);
});
```

Event: 'origin'

- `origins` `<string[]>`

The 'origin' event is emitted whenever an ORIGIN frame is received by the client. The event is emitted with an array of `origin` strings. The `http2session.originSet` will be updated to include the received origins.

```
const http2 = require('http2');
const client = http2.connect('https://example.org');

client.on('origin', (origins) => {
  for (let n = 0; n < origins.length; n++)
    console.log(origins[n]);
});
```

The 'origin' event is only emitted when using a secure TLS connection.

`clienthttp2session.request(headers[, options])`

- `headers` `<HTTP/2 Headers Object>`
- `options` `<Object>`

- `endStream` `<boolean>` `true` if the `Http2Stream` *writable* side should be closed initially, such as when sending a `GET` request that should not expect a payload body.
- `exclusive` `<boolean>` When `true` and `parent` identifies a parent Stream, the created stream is made the sole direct dependency of the parent, with all other existing dependents made a dependent of the newly created stream. **Default:** `false`.
- `parent` `<number>` Specifies the numeric identifier of a stream the newly created stream is dependent on.
- `weight` `<number>` Specifies the relative dependency of a stream in relation to other streams with the same `parent`. The value is a number between `1` and `256` (inclusive).
- `waitForTrailers` `<boolean>` When `true`, the `Http2Stream` will emit the `'wantTrailers'` event after the final `DATA` frame has been sent.
- `signal` `<AbortSignal>` An `AbortSignal` that may be used to abort an ongoing request.
- Returns: `<ClientHttp2Stream>`

For HTTP/2 Client `Http2Session` instances only, the `http2session.request()` creates and returns an `Http2Stream` instance that can be used to send an HTTP/2 request to the connected server.

This method is only available if `http2session.type` is equal to `http2.constants.NGHTTP2_SESSION_CLIENT`.

```
const http2 = require('http2');
const clientSession = http2.connect('https://localhost:1234');
const {
  HTTP2_HEADER_PATH,
  HTTP2_HEADER_STATUS
} = http2.constants;

const req = clientSession.request({ [HTTP2_HEADER_PATH]: '/' });
req.on('response', (headers) => {
  console.log(headers[HTTP2_HEADER_STATUS]);
  req.on('data', (chunk) => { /* .. */ });
  req.on('end', () => { /* .. */ });
});
```

When the `options.waitForTrailers` option is set, the `'wantTrailers'` event is emitted immediately after queuing the last chunk of payload data to be sent. The `http2stream.sendTrailers()` method can then be called to send trailing headers to the peer.

When `options.waitForTrailers` is set, the `Http2Stream` will not automatically close when the final `DATA` frame is transmitted. User code must call either `http2stream.sendTrailers()` or `http2stream.close()` to close the `Http2Stream`.

When `options.signal` is set with an `AbortSignal` and then `abort` on the corresponding `AbortController` is called, the request will emit an `'error'` event with an `AbortError` error.

The `:method` and `:path` pseudo-headers are not specified within `headers`, they respectively default to:

- `:method` = `'GET'`
- `:path` = `/`

Class: `Http2Stream`

- Extends: `<stream.Duplex>`

Each instance of the `Http2Stream` class represents a bidirectional HTTP/2 communications stream over an `Http2Session` instance. Any single `Http2Session` may have up to $2^{31}-1$ `Http2Stream` instances over its lifetime.

User code will not construct `Http2Stream` instances directly. Rather, these are created, managed, and provided to user code through the `Http2Session` instance. On the server, `Http2Stream` instances are created either in response to an incoming HTTP request (and handed off to user code via the `'stream'` event), or in response to a call to the `http2stream.pushStream()` method. On the client, `Http2Stream` instances are created and returned when either the `http2session.request()` method is called, or in response to an incoming `'push'` event.

The `Http2Stream` class is a base for the `ServerHttp2Stream` and `ClientHttp2Stream` classes, each of which is used specifically by either the Server or Client side, respectively.

All `Http2Stream` instances are `Duplex` streams. The `Writable` side of the `Duplex` is used to send data to the connected peer, while the `Readable` side is used to receive data sent by the connected peer.

The default text character encoding for all `Http2Stream`s is UTF-8. As a best practice, it is recommended that when using an `Http2Stream` to send text, the `'content-type'` header should be set and should identify the character encoding used.

```
stream.respond({
  'content-type': 'text/html; charset=utf-8',
  ':status': 200
});
```

Http2Stream Lifecycle

Creation

On the server side, instances of `ServerHttp2Stream` are created either when:

- A new HTTP/2 `HEADERS` frame with a previously unused stream ID is received;
- The `http2stream.pushStream()` method is called.

On the client side, instances of `ClientHttp2Stream` are created when the `http2session.request()` method is called.

On the client, the `Http2Stream` instance returned by `http2session.request()` may not be immediately ready for use if the parent `Http2Session` has not yet been fully established. In such cases, operations called on the `Http2Stream` will be buffered until the `'ready'` event is emitted. User code should rarely, if ever, need to handle the `'ready'` event directly. The ready status of an `Http2Stream` can be determined by checking the value of `http2stream.id`. If the value is `undefined`, the stream is not yet ready for use.

Destruction

All `Http2Stream` instances are destroyed either when:

- An `RST_STREAM` frame for the stream is received by the connected peer, and (for client streams only) pending data has been read.
- The `http2stream.close()` method is called, and (for client streams only) pending data has been read.
- The `http2stream.destroy()` or `http2session.destroy()` methods are called.

When an `Http2Stream` instance is destroyed, an attempt will be made to send an `RST_STREAM` frame to the connected peer.

When the `Http2Stream` instance is destroyed, the `'close'` event will be emitted. Because `Http2Stream` is an instance of `stream.Duplex`, the `'end'` event will also be emitted if the stream data is currently flowing. The `'error'` event may also be emitted if `http2stream.destroy()` was called with an `Error` passed as the first argument.

After the `Http2Stream` has been destroyed, the `http2stream.destroyed` property will be `true` and the `http2stream.rstCode` property will specify the `RST_STREAM` error code. The `Http2Stream` instance is no longer usable once destroyed.

Event: 'aborted'

The `'aborted'` event is emitted whenever a `Http2Stream` instance is abnormally aborted in mid-communication. Its listener does not expect any arguments.

The `'aborted'` event will only be emitted if the `Http2Stream` writable side has not been ended.

Event: `'close'`

The `'close'` event is emitted when the `Http2Stream` is destroyed. Once this event is emitted, the `Http2Stream` instance is no longer usable.

The HTTP/2 error code used when closing the stream can be retrieved using the `http2stream.rstCode` property. If the code is any value other than `NGHTTP2_NO_ERROR (0)`, an `'error'` event will have also been emitted.

Event: `'error'`

- `error` `<Error>`

The `'error'` event is emitted when an error occurs during the processing of an `Http2Stream`.

Event: `'frameError'`

- `type` `<integer>` The frame type.
- `code` `<integer>` The error code.
- `id` `<integer>` The stream id (or `0` if the frame isn't associated with a stream).

The `'frameError'` event is emitted when an error occurs while attempting to send a frame. When invoked, the handler function will receive an integer argument identifying the frame type, and an integer argument identifying the error code. The `Http2Stream` instance will be destroyed immediately after the `'frameError'` event is emitted.

Event: `'ready'`

The `'ready'` event is emitted when the `Http2Stream` has been opened, has been assigned an `id`, and can be used. The listener does not expect any arguments.

Event: `'timeout'`

The `'timeout'` event is emitted after no activity is received for this `Http2Stream` within the number of milliseconds set using `http2stream.setTimeout()`. Its listener does not expect any arguments.

Event: `'trailers'`

- `headers` `<HTTP/2 Headers Object>` An object describing the headers
- `flags` `<number>` The associated numeric flags

The `'trailers'` event is emitted when a block of headers associated with trailing header fields is received. The listener callback is passed the `HTTP/2 Headers Object` and flags associated with the headers.

This event might not be emitted if `http2stream.end()` is called before trailers are received and the incoming data is not being read or listened for.

```
stream.on('trailers', (headers, flags) => {
  console.log(headers);
});
```

Event: `'wantTrailers'`

The `'wantTrailers'` event is emitted when the `Http2Stream` has queued the final `DATA` frame to be sent on a frame and the `Http2Stream` is ready to send trailing headers. When initiating a request or response, the `waitForTrailers` option must be set for this event to be emitted.

`http2stream.aborted`

- `<boolean>`

Set to `true` if the `Http2Stream` instance was aborted abnormally. When set, the `'aborted'` event will have been emitted.

`http2stream.bufferSize`

- `<number>`

This property shows the number of characters currently buffered to be written. See `net.Socket.bufferSize` for details.

`http2stream.close([code[, callback]])`

- `code` `<number>` Unsigned 32-bit integer identifying the error code. **Default:** `http2.constants.NGHTTP2_NO_ERROR` (0x00).
- `callback` `<Function>` An optional function registered to listen for the `'close'` event.

Closes the `Http2Stream` instance by sending an `RST_STREAM` frame to the connected HTTP/2 peer.

`http2stream.closed`

- `<boolean>`

Set to `true` if the `Http2Stream` instance has been closed.

`http2stream.destroyed`

- `<boolean>`

Set to `true` if the `Http2Stream` instance has been destroyed and is no longer usable.

`http2stream.endAfterHeaders`

- `<boolean>`

Set the `true` if the `END_STREAM` flag was set in the request or response HEADERS frame received, indicating that no additional data should be received and the readable side of the `Http2Stream` will be closed.

`http2stream.id`

- `<number>` | `<undefined>`

The numeric stream identifier of this `Http2Stream` instance. Set to `undefined` if the stream identifier has not yet been assigned.

`http2stream.pending`

- `<boolean>`

Set to `true` if the `Http2Stream` instance has not yet been assigned a numeric stream identifier.

`http2stream.priority(options)`

- `options` `<Object>`
 - `exclusive` `<boolean>` When `true` and `parent` identifies a parent Stream, this stream is made the sole direct dependency of the parent, with all other existing dependents made a dependent of this stream. **Default:** `false`.
 - `parent` `<number>` Specifies the numeric identifier of a stream this stream is dependent on.

- `weight` `<number>` Specifies the relative dependency of a stream in relation to other streams with the same `parent`. The value is a number between `1` and `256` (inclusive).
- `silent` `<boolean>` When `true`, changes the priority locally without sending a `PRIORITY` frame to the connected peer.

Updates the priority for this `Http2Stream` instance.

`http2stream.rstCode`

- `<number>`

Set to the `RST_STREAM` `error code` reported when the `Http2Stream` is destroyed after either receiving an `RST_STREAM` frame from the connected peer, calling `http2stream.close()`, or `http2stream.destroy()`. Will be `undefined` if the `Http2Stream` has not been closed.

`http2stream.sentHeaders`

- `<HTTP/2 Headers Object>`

An object containing the outbound headers sent for this `Http2Stream`.

`http2stream.sentInfoHeaders`

- `<HTTP/2 Headers Object[]>`

An array of objects containing the outbound informational (additional) headers sent for this `Http2Stream`.

`http2stream.sentTrailers`

- `<HTTP/2 Headers Object>`

An object containing the outbound trailers sent for this `HttpStream`.

`http2stream.session`

- `<Http2Session>`

A reference to the `Http2Session` instance that owns this `Http2Stream`. The value will be `undefined` after the `Http2Stream` instance is destroyed.

`http2stream.setTimeout(msecs, callback)`

- `msecs` `<number>`
- `callback` `<Function>`

```
const http2 = require('http2');
const client = http2.connect('http://example.org:8000');
const { NGHTTP2_CANCEL } = http2.constants;
const req = client.request({ ':path': '/' });

// Cancel the stream if there's no activity after 5 seconds
req.setTimeout(5000, () => req.close(NGHTTP2_CANCEL));
```

`http2stream.state`

Provides miscellaneous information about the current state of the `Http2Stream`.

- `<Object>`
 - `localWindowSize` `<number>` The number of bytes the connected peer may send for this `Http2Stream` without receiving a `WINDOW_UPDATE`.

- `state` `<number>` A flag indicating the low-level current state of the `Http2Stream` as determined by `nghttp2`.
- `localClose` `<number>` `1` if this `Http2Stream` has been closed locally.
- `remoteClose` `<number>` `1` if this `Http2Stream` has been closed remotely.
- `sumDependencyWeight` `<number>` The sum weight of all `Http2Stream` instances that depend on this `Http2Stream` as specified using `PRIORITY` frames.
- `weight` `<number>` The priority weight of this `Http2Stream`.

A current state of this `Http2Stream`.

`http2stream.sendTrailers(headers)`

- `headers` `<HTTP/2 Headers Object>`

Sends a trailing `HEADERS` frame to the connected HTTP/2 peer. This method will cause the `Http2Stream` to be immediately closed and must only be called after the `'wantTrailers'` event has been emitted. When sending a request or sending a response, the `options.waitForTrailers` option must be set in order to keep the `Http2Stream` open after the final `DATA` frame so that trailers can be sent.

```
const http2 = require('http2');
const server = http2.createServer();
server.on('stream', (stream) => {
  stream.respond(undefined, { waitForTrailers: true });
  stream.on('wantTrailers', () => {
    stream.sendTrailers({ xyz: 'abc' });
  });
  stream.end('Hello World');
});
```

The HTTP/1 specification forbids trailers from containing HTTP/2 pseudo-header fields (e.g. `':method'`, `':path'`, etc).

Class: `ClientHttp2Stream`

- Extends `<Http2Stream>`

The `ClientHttp2Stream` class is an extension of `Http2Stream` that is used exclusively on HTTP/2 Clients. `Http2Stream` instances on the client provide events such as `'response'` and `'push'` that are only relevant on the client.

Event: `'continue'`

Emitted when the server sends a `100 Continue` status, usually because the request contained `Expect: 100-continue`. This is an instruction that the client should send the request body.

Event: `'headers'`

The `'headers'` event is emitted when an additional block of headers is received for a stream, such as when a block of `1xx` informational headers is received. The listener callback is passed the `HTTP/2 Headers Object` and flags associated with the headers.

```
stream.on('headers', (headers, flags) => {
  console.log(headers);
});
```

Event: `'push'`

The `'push'` event is emitted when response headers for a Server Push stream are received. The listener callback is passed the `HTTP/2 Headers Object` and flags associated with the headers.

```
stream.on('push', (headers, flags) => {
  console.log(headers);
});
```

Event: 'response'

The `'response'` event is emitted when a response `HEADERS` frame has been received for this stream from the connected HTTP/2 server. The listener is invoked with two arguments: an `Object` containing the received `HTTP/2 Headers Object`, and flags associated with the headers.

```
const http2 = require('http2');
const client = http2.connect('https://localhost');
const req = client.request({ ':path': '/' });
req.on('response', (headers, flags) => {
  console.log(headers[':status']);
});
```

Class: ServerHttp2Stream

- Extends: `<Http2Stream>`

The `ServerHttp2Stream` class is an extension of `Http2Stream` that is used exclusively on HTTP/2 Servers. `Http2Stream` instances on the server provide additional methods such as `http2stream.pushStream()` and `http2stream.respond()` that are only relevant on the server.

http2stream.additionalHeaders(headers)

- `headers` `<HTTP/2 Headers Object>`

Sends an additional informational `HEADERS` frame to the connected HTTP/2 peer.

http2stream.headersSent

- `<boolean>`

True if headers were sent, false otherwise (read-only).

http2stream.pushAllowed

- `<boolean>`

Read-only property mapped to the `SETTINGS_ENABLE_PUSH` flag of the remote client's most recent `SETTINGS` frame. Will be `true` if the remote peer accepts push streams, `false` otherwise. Settings are the same for every `Http2Stream` in the same `Http2Session`.

http2stream.pushStream(headers[, options], callback)

- `headers` `<HTTP/2 Headers Object>`
- `options` `<Object>`
 - `exclusive` `<boolean>` When `true` and `parent` identifies a parent Stream, the created stream is made the sole direct dependency of the parent, with all other existing dependents made a dependent of the newly created stream. **Default:** `false`.
 - `parent` `<number>` Specifies the numeric identifier of a stream the newly created stream is dependent on.
- `callback` `<Function>` Callback that is called once the push stream has been initiated.
 - `err` `<Error>`

- `pushStream` `<ServerHttp2Stream>` The returned `pushStream` object.
- `headers` `<HTTP/2 Headers Object>` Headers object the `pushStream` was initiated with.

Initiates a push stream. The callback is invoked with the new `Http2Stream` instance created for the push stream passed as the second argument, or an `Error` passed as the first argument.

```
const http2 = require('http2');
const server = http2.createServer();
server.on('stream', (stream) => {
  stream.respond({ ':status': 200 });
  stream.pushStream({ ':path': '/' }, (err, pushStream, headers) => {
    if (err) throw err;
    pushStream.respond({ ':status': 200 });
    pushStream.end('some pushed data');
  });
  stream.end('some data');
});
```

Setting the weight of a push stream is not allowed in the `HEADERS` frame. Pass a `weight` value to `http2stream.priority` with the `silent` option set to `true` to enable server-side bandwidth balancing between concurrent streams.

Calling `http2stream.pushStream()` from within a pushed stream is not permitted and will throw an error.

`http2stream.respond([headers[, options]])`

- `headers` `<HTTP/2 Headers Object>`
- `options` `<Object>`
 - `endStream` `<boolean>` Set to `true` to indicate that the response will not include payload data.
 - `waitForTrailers` `<boolean>` When `true`, the `Http2Stream` will emit the `'wantTrailers'` event after the final `DATA` frame has been sent.

```
const http2 = require('http2');
const server = http2.createServer();
server.on('stream', (stream) => {
  stream.respond({ ':status': 200 });
  stream.end('some data');
});
```

When the `options.waitForTrailers` option is set, the `'wantTrailers'` event will be emitted immediately after queuing the last chunk of payload data to be sent. The `http2stream.sendTrailers()` method can then be used to send trailing header fields to the peer.

When `options.waitForTrailers` is set, the `Http2Stream` will not automatically close when the final `DATA` frame is transmitted. User code must call either `http2stream.sendTrailers()` or `http2stream.close()` to close the `Http2Stream`.

```
const http2 = require('http2');
const server = http2.createServer();
server.on('stream', (stream) => {
  stream.respond({ ':status': 200 }, { waitForTrailers: true });
  stream.on('wantTrailers', () => {
    stream.sendTrailers({ ABC: 'some value to send' });
  });
});
```



```
stream.end('some data');
});
```

`http2stream.respondWithFD(fd[, headers[, options]])`

- `fd` `<number>` | `<FileHandle>` A readable file descriptor.
- `headers` `<HTTP/2 Headers Object>`
- `options` `<Object>`
 - `statCheck` `<Function>`
 - `waitForTrailers` `<boolean>` When `true`, the `Http2Stream` will emit the `'wantTrailers'` event after the final `DATA` frame has been sent.
 - `offset` `<number>` The offset position at which to begin reading.
 - `length` `<number>` The amount of data from the `fd` to send.

Initiates a response whose data is read from the given file descriptor. No validation is performed on the given file descriptor. If an error occurs while attempting to read data using the file descriptor, the `Http2Stream` will be closed using an `RST_STREAM` frame using the standard `INTERNAL_ERROR` code.

When used, the `Http2Stream` object's `Duplex` interface will be closed automatically.

```
const http2 = require('http2');
const fs = require('fs');

const server = http2.createServer();
server.on('stream', (stream) => {
  const fd = fs.openSync('/some/file', 'r');

  const stat = fs.fstatSync(fd);
  const headers = {
    'content-length': stat.size,
    'last-modified': stat.mtime.toUTCString(),
    'content-type': 'text/plain; charset=utf-8'
  };
  stream.respondWithFD(fd, headers);
  stream.on('close', () => fs.closeSync(fd));
});
```

The optional `options.statCheck` function may be specified to give user code an opportunity to set additional content headers based on the `fs.Stat` details of the given `fd`. If the `statCheck` function is provided, the `http2stream.respondWithFD()` method will perform an `fs.fstat()` call to collect details on the provided file descriptor.

The `offset` and `length` options may be used to limit the response to a specific range subset. This can be used, for instance, to support HTTP Range requests.

The file descriptor or `FileHandle` is not closed when the stream is closed, so it will need to be closed manually once it is no longer needed. Using the same file descriptor concurrently for multiple streams is not supported and may result in data loss. Re-using a file descriptor after a stream has finished is supported.

When the `options.waitForTrailers` option is set, the `'wantTrailers'` event will be emitted immediately after queuing the last chunk of payload data to be sent. The `http2stream.sendTrailers()` method can then be used to sent trailing header fields to the peer.

When `options.waitForTrailers` is set, the `Http2Stream` will not automatically close when the final `DATA` frame is transmitted. User code must call either `http2stream.sendTrailers()` or `http2stream.close()` to close the `Http2Stream`.

```
const http2 = require('http2');
const fs = require('fs');

const server = http2.createServer();
server.on('stream', (stream) => {
  const fd = fs.openSync('/some/file', 'r');

  const stat = fs.fstatSync(fd);
  const headers = {
    'content-length': stat.size,
    'last-modified': stat.mtime.toUTCString(),
    'content-type': 'text/plain; charset=utf-8'
  };
  stream.respondWithFD(fd, headers, { waitForTrailers: true });
  stream.on('wantTrailers', () => {
    stream.sendTrailers({ ABC: 'some value to send' });
  });

  stream.on('close', () => fs.closeSync(fd));
});
```

`http2stream.respondWithFile(path[, headers[, options]])`

- `path` `<string> | <Buffer> | <URL>`
- `headers` `<HTTP/2 Headers Object>`
- `options` `<Object>`
 - `statCheck` `<Function>`
 - `onError` `<Function>` Callback function invoked in the case of an error before send.
 - `waitForTrailers` `<boolean>` When `true`, the `Http2Stream` will emit the `'wantTrailers'` event after the final `DATA` frame has been sent.
 - `offset` `<number>` The offset position at which to begin reading.
 - `length` `<number>` The amount of data from the `fd` to send.

Sends a regular file as the response. The `path` must specify a regular file or an `'error'` event will be emitted on the `Http2Stream` object.

When used, the `Http2Stream` object's `Duplex` interface will be closed automatically.

The optional `options.statCheck` function may be specified to give user code an opportunity to set additional content headers based on the `fs.Stat` details of the given file:

If an error occurs while attempting to read the file data, the `Http2Stream` will be closed using an `RST_STREAM` frame using the standard `INTERNAL_ERROR` code. If the `onError` callback is defined, then it will be called. Otherwise the stream will be destroyed.

Example using a file path:

```
const http2 = require('http2');
const server = http2.createServer();
server.on('stream', (stream) => {
```

```
function statCheck(stat, headers) {
  headers['last-modified'] = stat.mtime.toUTCString();
}

function onError(err) {
  // stream.respond() can throw if the stream has been destroyed by
  // the other side.
  try {
    if (err.code === 'ENOENT') {
      stream.respond({ ':status': 404 });
    } else {
      stream.respond({ ':status': 500 });
    }
  } catch (err) {
    // Perform actual error handling.
    console.log(err);
  }
  stream.end();
}

stream.respondWithFile('/some/file',
  { 'content-type': 'text/plain; charset=utf-8' },
  { statCheck, onError });
});
```

The `options.statCheck` function may also be used to cancel the send operation by returning `false`. For instance, a conditional request may check the stat results to determine if the file has been modified to return an appropriate `304` response:

```
const http2 = require('http2');
const server = http2.createServer();
server.on('stream', (stream) => {
  function statCheck(stat, headers) {
    // Check the stat here...
    stream.respond({ ':status': 304 });
    return false; // Cancel the send operation
  }
  stream.respondWithFile('/some/file',
    { 'content-type': 'text/plain; charset=utf-8' },
    { statCheck });
});
```

The `content-length` header field will be automatically set.

The `offset` and `length` options may be used to limit the response to a specific range subset. This can be used, for instance, to support HTTP Range requests.

The `options.onError` function may also be used to handle all the errors that could happen before the delivery of the file is initiated. The default behavior is to destroy the stream.

When the `options.waitForTrailers` option is set, the `'wantTrailers'` event will be emitted immediately after queuing the last chunk of payload data to be sent. The `http2stream.sendTrailers()` method can then be used to send trailing header fields to the peer.

When `options.waitForTrailers` is set, the `Http2Stream` will not automatically close when the final `DATA` frame is transmitted. User code must call either `http2stream.sendTrailers()` or `http2stream.close()` to close the `Http2Stream`.

```
const http2 = require('http2');
const server = http2.createServer();
server.on('stream', (stream) => {
  stream.respondWithFile('/some/file',
    { 'content-type': 'text/plain; charset=utf-8' },
    { waitForTrailers: true });

  stream.on('wantTrailers', () => {
    stream.sendTrailers({ ABC: 'some value to send' });
  });
});
```

Class: `Http2Server`

- Extends: `<net.Server>`

Instances of `Http2Server` are created using the `http2.createServer()` function. The `Http2Server` class is not exported directly by the `http2` module.

Event: `'checkContinue'`

- `request` `<http2.Http2ServerRequest>`
- `response` `<http2.Http2ServerResponse>`

If a `'request'` listener is registered or `http2.createServer()` is supplied a callback function, the `'checkContinue'` event is emitted each time a request with an `HTTP Expect: 100-continue` is received. If this event is not listened for, the server will automatically respond with a status `100 Continue` as appropriate.

Handling this event involves calling `response.writeContinue()` if the client should continue to send the request body, or generating an appropriate HTTP response (e.g. 400 Bad Request) if the client should not continue to send the request body.

When this event is emitted and handled, the `'request'` event will not be emitted.

Event: `'connection'`

- `socket` `<stream.Duplex>`

This event is emitted when a new TCP stream is established. `socket` is typically an object of type `net.Socket`. Usually users will not want to access this event.

This event can also be explicitly emitted by users to inject connections into the HTTP server. In that case, any `Duplex` stream can be passed.

Event: `'request'`

- `request` `<http2.Http2ServerRequest>`
- `response` `<http2.Http2ServerResponse>`

Emitted each time there is a request. There may be multiple requests per session. See the [Compatibility API](#).

Event: `'session'`

The `'session'` event is emitted when a new `Http2Session` is created by the `Http2Server`.

Event: `'sessionError'`

The `'sessionError'` event is emitted when an `'error'` event is emitted by an `Http2Session` object associated with the `Http2Server`.

Event: `'stream'`

- `stream` `<Http2Stream>` A reference to the stream
- `headers` `<HTTP/2 Headers Object>` An object describing the headers
- `flags` `<number>` The associated numeric flags
- `rawHeaders` `<Array>` An array containing the raw header names followed by their respective values.

The `'stream'` event is emitted when a `'stream'` event has been emitted by an `Http2Session` associated with the server.

See also [Http2Session's 'stream' event](#).

```
const http2 = require('http2');
const {
  HTTP2_HEADER_METHOD,
  HTTP2_HEADER_PATH,
  HTTP2_HEADER_STATUS,
  HTTP2_HEADER_CONTENT_TYPE
} = http2.constants;

const server = http2.createServer();
server.on('stream', (stream, headers, flags) => {
  const method = headers[HTTP2_HEADER_METHOD];
  const path = headers[HTTP2_HEADER_PATH];
  // ...
  stream.respond({
    [HTTP2_HEADER_STATUS]: 200,
    [HTTP2_HEADER_CONTENT_TYPE]: 'text/plain; charset=utf-8'
  });
  stream.write('hello ');
  stream.end('world');
});
```

Event: `'timeout'`

The `'timeout'` event is emitted when there is no activity on the Server for a given number of milliseconds set using `http2server.setTimeout()`. **Default:** 0 (no timeout)

`server.close([callback])`

- `callback` `<Function>`

Stops the server from establishing new sessions. This does not prevent new request streams from being created due to the persistent nature of HTTP/2 sessions. To gracefully shut down the server, call `http2session.close()` on all active sessions.

If `callback` is provided, it is not invoked until all active sessions have been closed, although the server has already stopped allowing new sessions. See [net.Server.close\(\)](#) for more details.

`server.setTimeout([msecs][, callback])`

- `msecs` `<number>` **Default:** 0 (no timeout)
- `callback` `<Function>`

- Returns: `<Http2Server>`

Used to set the timeout value for http2 server requests, and sets a callback function that is called when there is no activity on the `Http2Server` after `msecs` milliseconds.

The given callback is registered as a listener on the `'timeout'` event.

In case if `callback` is not a function, a new `ERR_INVALID_CALLBACK` error will be thrown.

`server.timeout`

- `<number>` Timeout in milliseconds. **Default:** 0 (no timeout)

The number of milliseconds of inactivity before a socket is presumed to have timed out.

A value of `0` will disable the timeout behavior on incoming connections.

The socket timeout logic is set up on connection, so changing this value only affects new connections to the server, not any existing connections.

`server.updateSettings([settings])`

- `settings` `<HTTP/2 Settings Object>`

Used to update the server with the provided settings.

Throws `ERR_HTTP2_INVALID_SETTING_VALUE` for invalid `settings` values.

Throws `ERR_INVALID_ARG_TYPE` for invalid `settings` argument.

Class: `Http2SecureServer`

- Extends: `<tls.Server>`

Instances of `Http2SecureServer` are created using the `http2.createSecureServer()` function. The `Http2SecureServer` class is not exported directly by the `http2` module.

Event: `'checkContinue'`

- `request` `<http2.Http2ServerRequest>`
- `response` `<http2.Http2ServerResponse>`

If a `'request'` listener is registered or `http2.createSecureServer()` is supplied a callback function, the `'checkContinue'` event is emitted each time a request with an HTTP `Expect: 100-continue` is received. If this event is not listened for, the server will automatically respond with a status `100 Continue` as appropriate.

Handling this event involves calling `response.writeContinue()` if the client should continue to send the request body, or generating an appropriate HTTP response (e.g. 400 Bad Request) if the client should not continue to send the request body.

When this event is emitted and handled, the `'request'` event will not be emitted.

Event: `'connection'`

- `socket` `<stream.Duplex>`

This event is emitted when a new TCP stream is established, before the TLS handshake begins. `socket` is typically an object of type `net.Socket`. Usually users will not want to access this event.

This event can also be explicitly emitted by users to inject connections into the HTTP server. In that case, any `Duplex` stream can be passed.

Event: 'request'

- `request` `<http2.Http2ServerRequest>`
- `response` `<http2.Http2ServerResponse>`

Emitted each time there is a request. There may be multiple requests per session. See the [Compatibility API](#).

Event: 'session'

The `'session'` event is emitted when a new `Http2Session` is created by the `Http2SecureServer`.

Event: 'sessionError'

The `'sessionError'` event is emitted when an `'error'` event is emitted by an `Http2Session` object associated with the `Http2SecureServer`.

Event: 'stream'

- `stream` `<Http2Stream>` A reference to the stream
- `headers` `<HTTP/2 Headers Object>` An object describing the headers
- `flags` `<number>` The associated numeric flags
- `rawHeaders` `<Array>` An array containing the raw header names followed by their respective values.

The `'stream'` event is emitted when a `'stream'` event has been emitted by an `Http2Session` associated with the server.

See also [Http2Session's 'stream' event](#).

```
const http2 = require('http2');
const {
  HTTP2_HEADER_METHOD,
  HTTP2_HEADER_PATH,
  HTTP2_HEADER_STATUS,
  HTTP2_HEADER_CONTENT_TYPE
} = http2.constants;

const options = getOptionsSomehow();

const server = http2.createSecureServer(options);
server.on('stream', (stream, headers, flags) => {
  const method = headers[HTTP2_HEADER_METHOD];
  const path = headers[HTTP2_HEADER_PATH];
  // ...
  stream.respond({
    [HTTP2_HEADER_STATUS]: 200,
    [HTTP2_HEADER_CONTENT_TYPE]: 'text/plain; charset=utf-8'
  });
  stream.write('hello ');
  stream.end('world');
});
```

Event: 'timeout'

The `'timeout'` event is emitted when there is no activity on the Server for a given number of milliseconds set using `http2secureServer.setTimeout()`. **Default:** 2 minutes.

Event: 'unknownProtocol'

The `'unknownProtocol'` event is emitted when a connecting client fails to negotiate an allowed protocol (i.e. HTTP/2 or HTTP/1.1). The event handler receives the socket for handling. If no listener is registered for this event, the connection is terminated. A timeout may be specified using the `'unknownProtocolTimeout'` option passed to `http2.createSecureServer()`. See the [Compatibility API](#).

server.close([callback])

- `callback` [<Function>](#)

Stops the server from establishing new sessions. This does not prevent new request streams from being created due to the persistent nature of HTTP/2 sessions. To gracefully shut down the server, call `http2session.close()` on all active sessions.

If `callback` is provided, it is not invoked until all active sessions have been closed, although the server has already stopped allowing new sessions. See `tls.Server.close()` for more details.

server.setTimeout([msecs], callback)

- `msecs` [<number>](#) **Default:** 120000 (2 minutes)
- `callback` [<Function>](#)
- Returns: [<Http2SecureServer>](#)

Used to set the timeout value for http2 secure server requests, and sets a callback function that is called when there is no activity on the `Http2SecureServer` after `msecs` milliseconds.

The given callback is registered as a listener on the `'timeout'` event.

In case if `callback` is not a function, a new `ERR_INVALID_CALLBACK` error will be thrown.

server.timeout

- [<number>](#) Timeout in milliseconds. **Default:** 0 (no timeout)

The number of milliseconds of inactivity before a socket is presumed to have timed out.

A value of 0 will disable the timeout behavior on incoming connections.

The socket timeout logic is set up on connection, so changing this value only affects new connections to the server, not any existing connections.

server.updateSettings([settings])

- `settings` [<HTTP/2 Settings Object>](#)

Used to update the server with the provided settings.

Throws `ERR_HTTP2_INVALID_SETTING_VALUE` for invalid `settings` values.

Throws `ERR_INVALID_ARG_TYPE` for invalid `settings` argument.

http2.createServer(options[, onRequestHandler])

- `options` [<Object>](#)
 - `maxDeflateDynamicTableSize` [<number>](#) Sets the maximum dynamic table size for deflating header fields. **Default:** 4Kib.

- `maxSettings <number>` Sets the maximum number of settings entries per `SETTINGS` frame. The minimum value allowed is `1`. **Default:** `32`.
- `maxSessionMemory <number>` Sets the maximum memory that the `Http2Session` is permitted to use. The value is expressed in terms of number of megabytes, e.g. `1` equal 1 megabyte. The minimum value allowed is `1`. This is a credit based limit, existing `Http2Stream`s may cause this limit to be exceeded, but new `Http2Stream` instances will be rejected while this limit is exceeded. The current number of `Http2Stream` sessions, the current memory use of the header compression tables, current data queued to be sent, and unacknowledged `PING` and `SETTINGS` frames are all counted towards the current limit. **Default:** `10`.
- `maxHeaderListPairs <number>` Sets the maximum number of header entries. This is similar to `http.Server#maxHeadersCount` or `http.ClientRequest#maxHeadersCount`. The minimum value is `4`. **Default:** `128`.
- `maxOutstandingPings <number>` Sets the maximum number of outstanding, unacknowledged pings. **Default:** `10`.
- `maxSendHeaderBlockLength <number>` Sets the maximum allowed size for a serialized, compressed block of headers. Attempts to send headers that exceed this limit will result in a `'frameError'` event being emitted and the stream being closed and destroyed.
- `paddingStrategy <number>` The strategy used for determining the amount of padding to use for `HEADERS` and `DATA` frames. **Default:** `http2.constants.PADDING_STRATEGY_NONE`. Value may be one of:
 - `http2.constants.PADDING_STRATEGY_NONE`: No padding is applied.
 - `http2.constants.PADDING_STRATEGY_MAX`: The maximum amount of padding, determined by the internal implementation, is applied.
 - `http2.constants.PADDING_STRATEGY_ALIGNED`: Attempts to apply enough padding to ensure that the total frame length, including the 9-byte header, is a multiple of 8. For each frame, there is a maximum allowed number of padding bytes that is determined by current flow control state and settings. If this maximum is less than the calculated amount needed to ensure alignment, the maximum is used and the total frame length is not necessarily aligned at 8 bytes.
- `peerMaxConcurrentStreams <number>` Sets the maximum number of concurrent streams for the remote peer as if a `SETTINGS` frame had been received. Will be overridden if the remote peer sets its own value for `maxConcurrentStreams`. **Default:** `100`.
- `maxSessionInvalidFrames <integer>` Sets the maximum number of invalid frames that will be tolerated before the session is closed. **Default:** `1000`.
- `maxSessionRejectedStreams <integer>` Sets the maximum number of rejected upon creation streams that will be tolerated before the session is closed. Each rejection is associated with an `NGHTTP2_ENHANCE_YOUR_CALM` error that should tell the peer to not open any more streams, continuing to open streams is therefore regarded as a sign of a misbehaving peer. **Default:** `100`.
- `settings <HTTP/2 Settings Object>` The initial settings to send to the remote peer upon connection.
- `Http1IncomingMessage <http.IncomingMessage>` Specifies the `IncomingMessage` class to used for HTTP/1 fallback. Useful for extending the original `http.IncomingMessage`. **Default:** `http.IncomingMessage`.
- `Http1ServerResponse <http.ServerResponse>` Specifies the `ServerResponse` class to used for HTTP/1 fallback. Useful for extending the original `http.ServerResponse`. **Default:** `http.ServerResponse`.
- `Http2ServerRequest <http2.Http2ServerRequest>` Specifies the `Http2ServerRequest` class to use. Useful for extending the original `Http2ServerRequest`. **Default:** `Http2ServerRequest`.
- `Http2ServerResponse <http2.Http2ServerResponse>` Specifies the `Http2ServerResponse` class to use. Useful for extending the original `Http2ServerResponse`. **Default:** `Http2ServerResponse`.
- `unknownProtocolTimeout <number>` Specifies a timeout in milliseconds that a server should wait when an `'unknownProtocol'` is emitted. If the socket has not been destroyed by that time the server will destroy it. **Default:** `10000`.
- ...: Any `net.createServer()` option can be provided.
- `onRequestHandler <Function>` See [Compatibility API](#)
- Returns: `<Http2Server>`

Returns a `net.Server` instance that creates and manages `Http2Session` instances.

Since there are no browsers known that support `unencrypted HTTP/2`, the use of `http2.createSecureServer()` is necessary when communicating with browser clients.

```

const http2 = require('http2');

// Create an unencrypted HTTP/2 server.
// Since there are no browsers known that support
// unencrypted HTTP/2, the use of `http2.createSecureServer()`
// is necessary when communicating with browser clients.
const server = http2.createServer();

server.on('stream', (stream, headers) => {
  stream.respond({
    'content-type': 'text/html; charset=utf-8',
    ':status': 200
  });
  stream.end('<h1>Hello World</h1>');
});

server.listen(80);

```

http2.createSecureServer(options[, onRequestHandler])

- **options** `<Object>`
 - **allowHTTP1** `<boolean>` Incoming client connections that do not support HTTP/2 will be downgraded to HTTP/1.x when set to `true`. See the `'unknownProtocol'` event. See [ALPN negotiation](#). **Default:** `false`.
 - **maxDeflateDynamicTableSize** `<number>` Sets the maximum dynamic table size for deflating header fields. **Default:** `4Kib`.
 - **maxSettings** `<number>` Sets the maximum number of settings entries per `SETTINGS` frame. The minimum value allowed is `1`. **Default:** `32`.
 - **maxSessionMemory** `<number>` Sets the maximum memory that the `Http2Session` is permitted to use. The value is expressed in terms of number of megabytes, e.g. `1` equal 1 megabyte. The minimum value allowed is `1`. This is a credit based limit, existing `Http2Stream`s may cause this limit to be exceeded, but new `Http2Stream` instances will be rejected while this limit is exceeded. The current number of `Http2Stream` sessions, the current memory use of the header compression tables, current data queued to be sent, and unacknowledged `PING` and `SETTINGS` frames are all counted towards the current limit. **Default:** `10`.
 - **maxHeaderListPairs** `<number>` Sets the maximum number of header entries. This is similar to `http.Server#maxHeadersCount` or `http.ClientRequest#maxHeadersCount`. The minimum value is `4`. **Default:** `128`.
 - **maxOutstandingPings** `<number>` Sets the maximum number of outstanding, unacknowledged pings. **Default:** `10`.
 - **maxSendHeaderBlockLength** `<number>` Sets the maximum allowed size for a serialized, compressed block of headers. Attempts to send headers that exceed this limit will result in a `'frameError'` event being emitted and the stream being closed and destroyed.
 - **paddingStrategy** `<number>` Strategy used for determining the amount of padding to use for `HEADERS` and `DATA` frames. **Default:** `http2.constants.PADDING_STRATEGY_NONE`. Value may be one of:
 - `http2.constants.PADDING_STRATEGY_NONE`: No padding is applied.
 - `http2.constants.PADDING_STRATEGY_MAX`: The maximum amount of padding, determined by the internal implementation, is applied.
 - `http2.constants.PADDING_STRATEGY_ALIGNED`: Attempts to apply enough padding to ensure that the total frame length, including the 9-byte header, is a multiple of 8. For each frame, there is a maximum allowed number of padding bytes that is determined by current flow control state and settings. If this maximum is less than the calculated amount needed to ensure alignment, the maximum is used and the total frame length is not necessarily aligned at 8 bytes.
 - **peerMaxConcurrentStreams** `<number>` Sets the maximum number of concurrent streams for the remote peer as if a `SETTINGS` frame had been received. Will be overridden if the remote peer sets its own value for `maxConcurrentStreams`. **Default:** `100`.
 - **maxSessionInvalidFrames** `<integer>` Sets the maximum number of invalid frames that will be tolerated before the session is closed. **Default:** `1000`.

- `maxSessionRejectedStreams` `<integer>` Sets the maximum number of rejected upon creation streams that will be tolerated before the session is closed. Each rejection is associated with an `NGHTTP2_ENHANCE_YOUR_CALM` error that should tell the peer to not open any more streams, continuing to open streams is therefore regarded as a sign of a misbehaving peer. **Default:** `100`.
- `settings` `<HTTP/2 Settings Object>` The initial settings to send to the remote peer upon connection.
- ...: Any `tls.createServer()` options can be provided. For servers, the identity options (`pfx` or `key` / `cert`) are usually required.
- `origins` `<string[]>` An array of origin strings to send within an `ORIGIN` frame immediately following creation of a new server `Http2Session`.
- `unknownProtocolTimeout` `<number>` Specifies a timeout in milliseconds that a server should wait when an `'unknownProtocol'` event is emitted. If the socket has not been destroyed by that time the server will destroy it. **Default:** `10000`.
- `onRequestHandler` `<Function>` See [Compatibility API](#)
- Returns: `<Http2SecureServer>`

Returns a `tls.Server` instance that creates and manages `Http2Session` instances.

```
const http2 = require('http2');
const fs = require('fs');

const options = {
  key: fs.readFileSync('server-key.pem'),
  cert: fs.readFileSync('server-cert.pem')
};

// Create a secure HTTP/2 server
const server = http2.createSecureServer(options);

server.on('stream', (stream, headers) => {
  stream.respond({
    'content-type': 'text/html; charset=utf-8',
    ':status': 200
  });
  stream.end('<h1>Hello World</h1>');
});

server.listen(80);
```

http2.connect(authority[, options][, listener])

- `authority` `<string>` | `<URL>` The remote HTTP/2 server to connect to. This must be in the form of a minimal, valid URL with the `http://` or `https://` prefix, host name, and IP port (if a non-default port is used). Userinfo (user ID and password), path, querystring, and fragment details in the URL will be ignored.
- `options` `<Object>`
 - `maxDeflateDynamicTableSize` `<number>` Sets the maximum dynamic table size for deflating header fields. **Default:** `4Kib`.
 - `maxSettings` `<number>` Sets the maximum number of settings entries per `SETTINGS` frame. The minimum value allowed is `1`. **Default:** `32`.
 - `maxSessionMemory` `<number>` Sets the maximum memory that the `Http2Session` is permitted to use. The value is expressed in terms of number of megabytes, e.g. `1` equal 1 megabyte. The minimum value allowed is `1`. This is a credit based limit, existing `Http2Stream`s may cause this limit to be exceeded, but new `Http2Stream` instances will be rejected while this limit is exceeded. The current number of `Http2Stream` sessions, the current memory use of the header compression tables, current data queued to be sent, and unacknowledged `PING` and `SETTINGS` frames are all counted towards the current limit. **Default:** `10`.

- `maxHeaderListPairs` `<number>` Sets the maximum number of header entries. This is similar to `http.Server#maxHeadersCount` or `http.ClientRequest#maxHeadersCount`. The minimum value is `1`. **Default:** `128`.
- `maxOutstandingPings` `<number>` Sets the maximum number of outstanding, unacknowledged pings. **Default:** `10`.
- `maxReservedRemoteStreams` `<number>` Sets the maximum number of reserved push streams the client will accept at any given time. Once the current number of currently reserved push streams exceeds reaches this limit, new push streams sent by the server will be automatically rejected. The minimum allowed value is `0`. The maximum allowed value is $2^{32}-1$. A negative value sets this option to the maximum allowed value. **Default:** `200`.
- `maxSendHeaderBlockLength` `<number>` Sets the maximum allowed size for a serialized, compressed block of headers. Attempts to send headers that exceed this limit will result in a `'frameError'` event being emitted and the stream being closed and destroyed.
- `paddingStrategy` `<number>` Strategy used for determining the amount of padding to use for `HEADERS` and `DATA` frames. **Default:** `http2.constants.PADDING_STRATEGY_NONE`. Value may be one of:
 - `http2.constants.PADDING_STRATEGY_NONE`: No padding is applied.
 - `http2.constants.PADDING_STRATEGY_MAX`: The maximum amount of padding, determined by the internal implementation, is applied.
 - `http2.constants.PADDING_STRATEGY_ALIGNED`: Attempts to apply enough padding to ensure that the total frame length, including the 9-byte header, is a multiple of 8. For each frame, there is a maximum allowed number of padding bytes that is determined by current flow control state and settings. If this maximum is less than the calculated amount needed to ensure alignment, the maximum is used and the total frame length is not necessarily aligned at 8 bytes.
- `peerMaxConcurrentStreams` `<number>` Sets the maximum number of concurrent streams for the remote peer as if a `SETTINGS` frame had been received. Will be overridden if the remote peer sets its own value for `maxConcurrentStreams`. **Default:** `100`.
- `protocol` `<string>` The protocol to connect with, if not set in the `authority`. Value may be either `'http:'` or `'https:'`. **Default:** `'https:'`.
- `settings` `<HTTP/2 Settings Object>` The initial settings to send to the remote peer upon connection.
- `createConnection` `<Function>` An optional callback that receives the `URL` instance passed to `connect` and the `options` object, and returns any `Duplex` stream that is to be used as the connection for this session.
- ...: Any `net.connect()` or `tls.connect()` options can be provided.
- `unknownProtocolTimeout` `<number>` Specifies a timeout in milliseconds that a server should wait when an `'unknownProtocol'` event is emitted. If the socket has not been destroyed by that time the server will destroy it. **Default:** `10000`.
- `listener` `<Function>` Will be registered as a one-time listener of the `'connect'` event.
- Returns: `<ClientHttp2Session>`

Returns a `ClientHttp2Session` instance.

```
const http2 = require('http2');
const client = http2.connect('https://localhost:1234');

/* Use the client */

client.close();
```

http2.constants

Error codes for `RST_STREAM` and `GOAWAY`

Value	Name	Constant
<code>0x00</code>	No Error	<code>http2.constants.NGHTTP2_NO_ERROR</code>
<code>0x01</code>	Protocol Error	<code>http2.constants.NGHTTP2_PROTOCOL_ERROR</code>

Value	Name	Constant
0x02	Internal Error	http2.constants.NGHTTP2_INTERNAL_ERROR
0x03	Flow Control Error	http2.constants.NGHTTP2_FLOW_CONTROL_ERROR
0x04	Settings Timeout	http2.constants.NGHTTP2_SETTINGS_TIMEOUT
0x05	Stream Closed	http2.constants.NGHTTP2_STREAM_CLOSED
0x06	Frame Size Error	http2.constants.NGHTTP2_FRAME_SIZE_ERROR
0x07	Refused Stream	http2.constants.NGHTTP2_REFUSED_STREAM
0x08	Cancel	http2.constants.NGHTTP2_CANCEL
0x09	Compression Error	http2.constants.NGHTTP2_COMPRESSION_ERROR
0x0a	Connect Error	http2.constants.NGHTTP2_CONNECT_ERROR
0x0b	Enhance Your Calm	http2.constants.NGHTTP2_ENHANCE_YOUR_CALM
0x0c	Inadequate Security	http2.constants.NGHTTP2_INADEQUATE_SECURITY
0x0d	HTTP/1.1 Required	http2.constants.NGHTTP2_HTTP_1_1_REQUIRED

The `'timeout'` event is emitted when there is no activity on the Server for a given number of milliseconds set using `http2server.setTimeout()`.

http2.getDefaultSettings()

- Returns: `<HTTP/2 Settings Object>`

Returns an object containing the default settings for an `Http2Session` instance. This method returns a new object instance every time it is called so instances returned may be safely modified for use.

http2.getPackedSettings([settings])

- `settings` `<HTTP/2 Settings Object>`
- Returns: `<Buffer>`

Returns a `Buffer` instance containing serialized representation of the given HTTP/2 settings as specified in the `HTTP/2` specification. This is intended for use with the `HTTP2-Settings` header field.

```
const http2 = require('http2');

const packed = http2.getPackedSettings({ enablePush: false });

console.log(packed.toString('base64'));
// Prints: AAIAAAAA
```

http2.getUnpackedSettings(buf)

- `buf` `<Buffer>` | `<TypedArray>` The packed settings.
- Returns: `<HTTP/2 Settings Object>`

Returns a `HTTP/2 Settings Object` containing the deserialized settings from the given `Buffer` as generated by `http2.getPackedSettings()`.

http2.sensitiveHeaders

- `<symbol>`

This symbol can be set as a property on the HTTP/2 headers object with an array value in order to provide a list of headers considered sensitive. See [Sensitive headers](#) for more details.

Headers object

Headers are represented as own-properties on JavaScript objects. The property keys will be serialized to lower-case. Property values should be strings (if they are not they will be coerced to strings) or an `Array` of strings (in order to send more than one value per header field).

```
const headers = {
  ':status': '200',
  'content-type': 'text-plain',
  'ABC': ['has', 'more', 'than', 'one', 'value']
};

stream.respond(headers);
```

Header objects passed to callback functions will have a `null` prototype. This means that normal JavaScript object methods such as `Object.prototype.toString()` and `Object.prototype.hasOwnProperty()` will not work.

For incoming headers:

- The `:status` header is converted to `number`.
- Duplicates of `:status`, `:method`, `:authority`, `:scheme`, `:path`, `:protocol`, `age`, `authorization`, `access-control-allow-credentials`, `access-control-max-age`, `access-control-request-method`, `content-encoding`, `content-language`, `content-length`, `content-location`, `content-md5`, `content-range`, `content-type`, `date`, `dnt`, `etag`, `expires`, `from`, `host`, `if-match`, `if-modified-since`, `if-none-match`, `if-range`, `if-unmodified-since`, `last-modified`, `location`, `max-forwards`, `proxy-authorization`, `range`, `referer`, `retry-after`, `tk`, `upgrade-insecure-requests`, `user-agent` or `x-content-type-options` are discarded.
- `set-cookie` is always an array. Duplicates are added to the array.
- For duplicate `cookie` headers, the values are joined together with `;`.
- For all other headers, the values are joined together with `,`.

```
const http2 = require('http2');
const server = http2.createServer();
server.on('stream', (stream, headers) => {
  console.log(headers[':path']);
  console.log(headers.ABC);
});
```

Sensitive headers

HTTP2 headers can be marked as sensitive, which means that the HTTP/2 header compression algorithm will never index them. This can make sense for header values with low entropy and that may be considered valuable to an attacker, for example `Cookie` or `Authorization`. To achieve this, add the header name to the `[http2.sensitiveHeaders]` property as an array:

```
const headers = {
  ':status': '200',
  'content-type': 'text-plain',
  'cookie': 'some-cookie',
  'other-sensitive-header': 'very secret data',
  [http2.sensitiveHeaders]: ['cookie', 'other-sensitive-header']
};

stream.respond(headers);
```

For some headers, such as `Authorization` and short `Cookie` headers, this flag is set automatically.

This property is also set for received headers. It will contain the names of all headers marked as sensitive, including ones marked that way automatically.

Settings object

The `http2.getDefaultSettings()`, `http2.getPackedSettings()`, `http2.createServer()`, `http2.createSecureServer()`, `http2session.settings()`, `http2session.localSettings`, and `http2session.remoteSettings` APIs either return or receive as input an object that defines configuration settings for an `Http2Session` object. These objects are ordinary JavaScript objects containing the following properties.

- `headerTableSize` `<number>` Specifies the maximum number of bytes used for header compression. The minimum allowed value is 0. The maximum allowed value is $2^{32}-1$. **Default:** 4096.
- `enablePush` `<boolean>` Specifies `true` if HTTP/2 Push Streams are to be permitted on the `Http2Session` instances. **Default:** `true`.
- `initialWindowSize` `<number>` Specifies the *sender's* initial window size in bytes for stream-level flow control. The minimum allowed value is 0. The maximum allowed value is $2^{32}-1$. **Default:** 65535.
- `maxFrameSize` `<number>` Specifies the size in bytes of the largest frame payload. The minimum allowed value is 16,384. The maximum allowed value is $2^{24}-1$. **Default:** 16384.
- `maxConcurrentStreams` `<number>` Specifies the maximum number of concurrent streams permitted on an `Http2Session`. There is no default value which implies, at least theoretically, $2^{32}-1$ streams may be open concurrently at any given time in an `Http2Session`. The minimum value is 0. The maximum allowed value is $2^{32}-1$. **Default:** 4294967295.
- `maxHeaderListSize` `<number>` Specifies the maximum size (uncompressed octets) of header list that will be accepted. The minimum allowed value is 0. The maximum allowed value is $2^{32}-1$. **Default:** 65535.
- `maxHeaderSize` `<number>` Alias for `maxHeaderListSize`.
- `enableConnectProtocol` `<boolean>` Specifies `true` if the "Extended Connect Protocol" defined by [RFC 8441](#) is to be enabled. This setting is only meaningful if sent by the server. Once the `enableConnectProtocol` setting has been enabled for a given `Http2Session`, it cannot be disabled. **Default:** `false`.

All additional properties on the settings object are ignored.

Error handling

There are several types of error conditions that may arise when using the `http2` module:

Validation errors occur when an incorrect argument, option, or setting value is passed in. These will always be reported by a synchronous `throw`.

State errors occur when an action is attempted at an incorrect time (for instance, attempting to send data on a stream after it has closed). These will be reported using either a synchronous `throw` or via an `'error'` event on the `Http2Stream`, `Http2Session` or HTTP/2 Server objects, depending on where and when the error occurs.

Internal errors occur when an HTTP/2 session fails unexpectedly. These will be reported via an `'error'` event on the `Http2Session` or `HTTP/2 Server` objects.

Protocol errors occur when various HTTP/2 protocol constraints are violated. These will be reported using either a synchronous `throw` or via an `'error'` event on the `Http2Stream`, `Http2Session` or `HTTP/2 Server` objects, depending on where and when the error occurs.

Invalid character handling in header names and values

The HTTP/2 implementation applies stricter handling of invalid characters in HTTP header names and values than the HTTP/1 implementation.

Header field names are *case-insensitive* and are transmitted over the wire strictly as lower-case strings. The API provided by Node.js allows header names to be set as mixed-case strings (e.g. `Content-Type`) but will convert those to lower-case (e.g. `content-type`) upon transmission.

Header field-names *must only* contain one or more of the following ASCII characters: `a-z`, `A-Z`, `0-9`, `!`, `#`, `$`, `%`, `&`, `'`, `*`, `+`, `-`, `.`, `^`, `_`, ``` (backtick), `|`, and `~`.

Using invalid characters within an HTTP header field name will cause the stream to be closed with a protocol error being reported.

Header field values are handled with more leniency but *should* not contain new-line or carriage return characters and *should* be limited to US-ASCII characters, per the requirements of the HTTP specification.

Push streams on the client

To receive pushed streams on the client, set a listener for the `'stream'` event on the `ClientHttp2Session`:

```
const http2 = require('http2');

const client = http2.connect('http://localhost');

client.on('stream', (pushedStream, requestHeaders) => {
  pushedStream.on('push', (responseHeaders) => {
    // Process response headers
  });
  pushedStream.on('data', (chunk) => { /* handle pushed data */ });
});

const req = client.request({ ':path': '/' });
```

Supporting the `CONNECT` method

The `CONNECT` method is used to allow an HTTP/2 server to be used as a proxy for TCP/IP connections.

A simple TCP Server:

```
const net = require('net');

const server = net.createServer((socket) => {
  let name = '';
  socket.setEncoding('utf8');
  socket.on('data', (chunk) => name += chunk);
  socket.on('end', () => socket.end(`hello ${name}`));
});
```



```
});  
  
server.listen(8000);
```

An HTTP/2 CONNECT proxy:

```
const http2 = require('http2');  
const { NGHTTP2_REFUSED_STREAM } = http2.constants;  
const net = require('net');  
  
const proxy = http2.createServer();  
proxy.on('stream', (stream, headers) => {  
  if (headers[':method'] !== 'CONNECT') {  
    // Only accept CONNECT requests  
    stream.close(NGHTTP2_REFUSED_STREAM);  
    return;  
  }  
  const auth = new URL(`tcp://${headers[':authority']}`);  
  // It's a very good idea to verify that hostname and port are  
  // things this proxy should be connecting to.  
  const socket = net.connect(auth.port, auth.hostname, () => {  
    stream.respond();  
    socket.pipe(stream);  
    stream.pipe(socket);  
  });  
  socket.on('error', (error) => {  
    stream.close(http2.constants.NGHTTP2_CONNECT_ERROR);  
  });  
});  
  
proxy.listen(8001);
```

An HTTP/2 CONNECT client:

```
const http2 = require('http2');  
  
const client = http2.connect('http://localhost:8001');  
  
// Must not specify the ':path' and ':scheme' headers  
// for CONNECT requests or an error will be thrown.  
const req = client.request({  
  ':method': 'CONNECT',  
  ':authority': `localhost:${port}`  
});  
  
req.on('response', (headers) => {  
  console.log(headers[http2.constants.HTTP2_HEADER_STATUS]);  
});  
  
let data = '';  
req.setEncoding('utf8');
```

```
req.on('data', (chunk) => data += chunk);
req.on('end', () => {
  console.log(`The server says: ${data}`);
  client.close();
});
req.end('Jane');
```

The extended CONNECT protocol

[RFC 8441](#) defines an "Extended CONNECT Protocol" extension to HTTP/2 that may be used to bootstrap the use of an `Http2Stream` using the `CONNECT` method as a tunnel for other communication protocols (such as WebSockets).

The use of the Extended CONNECT Protocol is enabled by HTTP/2 servers by using the `enableConnectProtocol` setting:

```
const http2 = require('http2');
const settings = { enableConnectProtocol: true };
const server = http2.createServer({ settings });
```

Once the client receives the `SETTINGS` frame from the server indicating that the extended CONNECT may be used, it may send `CONNECT` requests that use the `:protocol` HTTP/2 pseudo-header:

```
const http2 = require('http2');
const client = http2.connect('http://localhost:8080');
client.on('remoteSettings', (settings) => {
  if (settings.enableConnectProtocol) {
    const req = client.request({ ':method': 'CONNECT', ':protocol': 'foo' });
    // ...
  }
});
```

Compatibility API

The Compatibility API has the goal of providing a similar developer experience of HTTP/1 when using HTTP/2, making it possible to develop applications that support both [HTTP/1](#) and HTTP/2. This API targets only the **public API** of the [HTTP/1](#). However many modules use internal methods or state, and those *are not supported* as it is a completely different implementation.

The following example creates an HTTP/2 server using the compatibility API:

```
const http2 = require('http2');
const server = http2.createServer((req, res) => {
  res.setHeader('Content-Type', 'text/html');
  res.setHeader('X-Foo', 'bar');
  res.writeHead(200, { 'Content-Type': 'text/plain; charset=utf-8' });
  res.end('ok');
});
```

In order to create a mixed [HTTPS](#) and HTTP/2 server, refer to the [ALPN negotiation](#) section. Upgrading from non-tls HTTP/1 servers is not supported.

The HTTP/2 compatibility API is composed of `Http2ServerRequest` and `Http2ServerResponse`. They aim at API compatibility with HTTP/1, but they do not hide the differences between the protocols. As an example, the status message for HTTP codes is ignored.

ALPN negotiation

ALPN negotiation allows supporting both `HTTPS` and HTTP/2 over the same socket. The `req` and `res` objects can be either HTTP/1 or HTTP/2, and an application **must** restrict itself to the public API of `HTTP/1`, and detect if it is possible to use the more advanced features of HTTP/2.

The following example creates a server that supports both protocols:

```
const { createSecureServer } = require('http2');
const { readFileSync } = require('fs');

const cert = readFileSync('./cert.pem');
const key = readFileSync('./key.pem');

const server = createSecureServer(
  { cert, key, allowHTTP1: true },
  onRequest
).listen(4443);

function onRequest(req, res) {
  // Detects if it is a HTTPS request or HTTP/2
  const { socket: { alpnProtocol } } = req.httpVersion === '2.0' ?
    req.stream.session : req;
  res.writeHead(200, { 'content-type': 'application/json' });
  res.end(JSON.stringify({
    alpnProtocol,
    httpVersion: req.httpVersion
  }));
}
```

The `'request'` event works identically on both `HTTPS` and HTTP/2.

Class: `http2.Http2ServerRequest`

- Extends: `<stream.Readable>`

A `Http2ServerRequest` object is created by `http2.Server` or `http2.SecureServer` and passed as the first argument to the `'request'` event. It may be used to access a request status, headers, and data.

Event: `'aborted'`

The `'aborted'` event is emitted whenever a `Http2ServerRequest` instance is abnormally aborted in mid-communication.

The `'aborted'` event will only be emitted if the `Http2ServerRequest` writable side has not been ended.

Event: `'close'`

Indicates that the underlying `Http2Stream` was closed. Just like `'end'`, this event occurs only once per response.

`request.aborted`

- `<boolean>`

The `request.aborted` property will be `true` if the request has been aborted.

request.authority

- `<string>`

The request authority pseudo header field. Because HTTP/2 allows requests to set either `:authority` or `host`, this value is derived from `req.headers[':authority']` if present. Otherwise, it is derived from `req.headers['host']`.

request.complete

- `<boolean>`

The `request.complete` property will be `true` if the request has been completed, aborted, or destroyed.

request.connection

Stability: 0 - Deprecated. Use `request.socket`.

- `<net.Socket>` | `<tls.TLSSocket>`

See `request.socket`.

request.destroy([error])

- `error` `<Error>`

Calls `destroy()` on the `Http2Stream` that received the `Http2ServerRequest`. If `error` is provided, an `'error'` event is emitted and `error` is passed as an argument to any listeners on the event.

It does nothing if the stream was already destroyed.

request.headers

- `<Object>`

The request/response headers object.

Key-value pairs of header names and values. Header names are lower-cased.

```
// Prints something like:
//
// { 'user-agent': 'curl/7.22.0',
//   host: '127.0.0.1:8000',
//   accept: '*/.*' }
console.log(request.headers);
```

See [HTTP/2 Headers Object](#).

In HTTP/2, the request path, host name, protocol, and method are represented as special headers prefixed with the `:` character (e.g. `:path`). These special headers will be included in the `request.headers` object. Care must be taken not to inadvertently modify these special headers or errors may occur. For instance, removing all headers from the request will cause errors to occur:

```
removeAllHeaders(request.headers);
assert(request.url); // Fails because the :path header has been removed
```

request.httpVersion

- `<string>`

In case of server request, the HTTP version sent by the client. In the case of client response, the HTTP version of the connected-to server. Returns `'2.0'`.

Also `message.httpVersionMajor` is the first integer and `message.httpVersionMinor` is the second.

request.method

- `<string>`

The request method as a string. Read-only. Examples: `'GET'`, `'DELETE'`.

request.rawHeaders

- `<string[]>`

The raw request/response headers list exactly as they were received.

The keys and values are in the same list. It is *not* a list of tuples. So, the even-numbered offsets are key values, and the odd-numbered offsets are the associated values.

Header names are not lowercased, and duplicates are not merged.

```
// Prints something like:
//
// [ 'user-agent',
//   'this is invalid because there can be only one',
//   'User-Agent',
//   'curl/7.22.0',
//   'Host',
//   '127.0.0.1:8000',
//   'ACCEPT',
//   '/*/*' ]
console.log(request.rawHeaders);
```

request.rawTrailers

- `<string[]>`

The raw request/response trailer keys and values exactly as they were received. Only populated at the `'end'` event.

request.scheme

- `<string>`

The request scheme pseudo header field indicating the scheme portion of the target URL.

request.setTimeout(msecs, callback)

- `msecs` `<number>`

- `callback` `<Function>`
- Returns: `<http2.Http2ServerRequest>`

Sets the `Http2Stream`'s timeout value to `msecs`. If a callback is provided, then it is added as a listener on the `'timeout'` event on the response object.

If no `'timeout'` listener is added to the request, the response, or the server, then `Http2Stream`s are destroyed when they time out. If a handler is assigned to the request, the response, or the server's `'timeout'` events, timed out sockets must be handled explicitly.

request.socket

- `<net.Socket>` | `<tls.TLSSocket>`

Returns a `Proxy` object that acts as a `net.Socket` (or `tls.TLSSocket`) but applies getters, setters, and methods based on HTTP/2 logic.

`destroyed`, `readable`, and `writable` properties will be retrieved from and set on `request.stream`.

`destroy`, `emit`, `end`, `on` and `once` methods will be called on `request.stream`.

`setTimeout` method will be called on `request.stream.session`.

`pause`, `read`, `resume`, and `write` will throw an error with code `ERR_HTTP2_NO_SOCKET_MANIPULATION`. See [Http2Session and Sockets](#) for more information.

All other interactions will be routed directly to the socket. With TLS support, use `request.socket.getPeerCertificate()` to obtain the client's authentication details.

request.stream

- `<Http2Stream>`

The `Http2Stream` object backing the request.

request.trailers

- `<Object>`

The request/response trailers object. Only populated at the `'end'` event.

request.url

- `<string>`

Request URL string. This contains only the URL that is present in the actual HTTP request. If the request is:

```
GET /status?name=ryan HTTP/1.1
Accept: text/plain
```

Then `request.url` will be:

```
'/status?name=ryan'
```

To parse the url into its parts, `new URL()` can be used:

```
$ node
> new URL('/status?name=ryan', 'http://example.com')
```

```

URL {
  href: 'http://example.com/status?name=ryan',
  origin: 'http://example.com',
  protocol: 'http:',
  username: '',
  password: '',
  host: 'example.com',
  hostname: 'example.com',
  port: '',
  pathname: '/status',
  search: '?name=ryan',
  searchParams: URLSearchParams { 'name' => 'ryan' },
  hash: ''
}

```

Class: `http2.Http2ServerResponse`

- Extends: `<Stream>`

This object is created internally by an HTTP server, not by the user. It is passed as the second parameter to the `'request'` event.

Event: `'close'`

Indicates that the underlying `Http2Stream` was terminated before `response.end()` was called or able to flush.

Event: `'finish'`

Emitted when the response has been sent. More specifically, this event is emitted when the last segment of the response headers and body have been handed off to the HTTP/2 multiplexing for transmission over the network. It does not imply that the client has received anything yet.

After this event, no more events will be emitted on the response object.

`response.addTrailers(headers)`

- `headers` `<Object>`

This method adds HTTP trailing headers (a header but at the end of the message) to the response.

Attempting to set a header field name or value that contains invalid characters will result in a `TypeError` being thrown.

`response.connection`

Stability: 0 - Deprecated. Use `response.socket`.

- `<net.Socket>` | `<tls.TLSSocket>`

See `response.socket`.

`response.createPushResponse(headers, callback)`

- `headers` `<HTTP/2 Headers Object>` An object describing the headers
- `callback` `<Function>` Called once `http2stream.pushStream()` is finished, or either when the attempt to create the pushed `Http2Stream` has failed or has been rejected, or the state of `Http2ServerRequest` is closed prior to calling the `http2stream.pushStream()` method

- `err` `<Error>`
- `res` `<http2.Http2ServerResponse>` The newly-created `Http2ServerResponse` object

Call `http2stream.pushStream()` with the given headers, and wrap the given `Http2Stream` on a newly created `Http2ServerResponse` as the callback parameter if successful. When `Http2ServerRequest` is closed, the callback is called with an error `ERR_HTTP2_INVALID_STREAM`.

`response.end([data[, encoding]][, callback])`

- `data` `<string> | <Buffer> | <Uint8Array>`
- `encoding` `<string>`
- `callback` `<Function>`
- Returns: `<this>`

This method signals to the server that all of the response headers and body have been sent; that server should consider this message complete. The method, `response.end()`, MUST be called on each response.

If `data` is specified, it is equivalent to calling `response.write(data, encoding)` followed by `response.end(callback)`.

If `callback` is specified, it will be called when the response stream is finished.

`response.finished`

Stability: 0 - Deprecated. Use `response.writableEnded`.

- `<boolean>`

Boolean value that indicates whether the response has completed. Starts as `false`. After `response.end()` executes, the value will be `true`.

`response.getHeader(name)`

- `name` `<string>`
- Returns: `<string>`

Reads out a header that has already been queued but not sent to the client. The name is case-insensitive.

```
const contentType = response.getHeader('content-type');
```

`response.getHeaderNames()`

- Returns: `<string[]>`

Returns an array containing the unique names of the current outgoing headers. All header names are lowercase.

```
response.setHeader('Foo', 'bar');
response.setHeader('Set-Cookie', ['foo=bar', 'bar=baz']);

const headerNames = response.getHeaderNames();
// headerNames === ['foo', 'set-cookie']
```

`response.getHeaders()`

- Returns: `<Object>`

Returns a shallow copy of the current outgoing headers. Since a shallow copy is used, array values may be mutated without additional calls to various header-related http module methods. The keys of the returned object are the header names and the values are the respective header values. All header names are lowercase.

The object returned by the `response.getHeaders()` method *does not* prototypically inherit from the JavaScript `Object`. This means that typical `Object` methods such as `obj.toString()`, `obj.hasOwnProperty()`, and others are not defined and *will not work*.

```
response.setHeader('Foo', 'bar');
response.setHeader('Set-Cookie', ['foo=bar', 'bar=baz']);

const headers = response.getHeaders();
// headers === { foo: 'bar', 'set-cookie': ['foo=bar', 'bar=baz'] }
```

response.hasHeader(name)

- `name` `<string>`
- Returns: `<boolean>`

Returns `true` if the header identified by `name` is currently set in the outgoing headers. The header name matching is case-insensitive.

```
const hasContentType = response.hasHeader('content-type');
```

response.headersSent

- `<boolean>`

True if headers were sent, false otherwise (read-only).

response.removeHeader(name)

- `name` `<string>`

Removes a header that has been queued for implicit sending.

```
response.removeHeader('Content-Encoding');
```

response.req

- `<http2.Http2ServerRequest>`

A reference to the original HTTP2 `request` object.

response.sendDate

- `<boolean>`

When true, the Date header will be automatically generated and sent in the response if it is not already present in the headers. Defaults to true.

This should only be disabled for testing; HTTP requires the Date header in responses.

response.setHeader(name, value)

- `name` `<string>`
- `value` `<string>` | `<string[]>`

Sets a single header value for implicit headers. If this header already exists in the to-be-sent headers, its value will be replaced. Use an array of strings here to send multiple headers with the same name.

```
response.setHeader('Content-Type', 'text/html; charset=utf-8');
```

or

```
response.setHeader('Set-Cookie', ['type=ninja', 'language=javascript']);
```

Attempting to set a header field name or value that contains invalid characters will result in a `TypeError` being thrown.

When headers have been set with `response.setHeader()`, they will be merged with any headers passed to `response.writeHead()`, with the headers passed to `response.writeHead()` given precedence.

```
// Returns content-type = text/plain
const server = http2.createServer((req, res) => {
  res.setHeader('Content-Type', 'text/html; charset=utf-8');
  res.setHeader('X-Foo', 'bar');
  res.writeHead(200, { 'Content-Type': 'text/plain; charset=utf-8' });
  res.end('ok');
});
```

`response.setTimeout(msecs[, callback])`

- `msecs` `<number>`
- `callback` `<Function>`
- Returns: `<http2.Http2ServerResponse>`

Sets the `Http2Stream`'s timeout value to `msecs`. If a callback is provided, then it is added as a listener on the `'timeout'` event on the response object.

If no `'timeout'` listener is added to the request, the response, or the server, then `Http2Stream`s are destroyed when they time out. If a handler is assigned to the request, the response, or the server's `'timeout'` events, timed out sockets must be handled explicitly.

`response.socket`

- `<net.Socket>` | `<tls.TLSSocket>`

Returns a `Proxy` object that acts as a `net.Socket` (or `tls.TLSSocket`) but applies getters, setters, and methods based on HTTP/2 logic.

`destroyed`, `readable`, and `writable` properties will be retrieved from and set on `response.stream`.

`destroy`, `emit`, `end`, `on` and `once` methods will be called on `response.stream`.

`setTimeout` method will be called on `response.stream.session`.

`pause`, `read`, `resume`, and `write` will throw an error with code `ERR_HTTP2_NO_SOCKET_MANIPULATION`. See [Http2Session and Sockets](#) for more information.

All other interactions will be routed directly to the socket.

```
const http2 = require('http2');
const server = http2.createServer((req, res) => {
```

```
const ip = req.socket.remoteAddress;
const port = req.socket.remotePort;
res.end(`Your IP address is ${ip} and your source port is ${port}.`);
}).listen(3000);
```

response.statusCode

- `<number>`

When using implicit headers (not calling `response.writeHead()` explicitly), this property controls the status code that will be sent to the client when the headers get flushed.

```
response.statusCode = 404;
```

After response header was sent to the client, this property indicates the status code which was sent out.

response.statusMessage

- `<string>`

Status message is not supported by HTTP/2 (RFC 7540 8.1.2.4). It returns an empty string.

response.stream

- `<Http2Stream>`

The `Http2Stream` object backing the response.

response.writableEnded

- `<boolean>`

Is `true` after `response.end()` has been called. This property does not indicate whether the data has been flushed, for this use `writable.writableFinished` instead.

response.write(chunk[, encoding][, callback])

- `chunk` `<string>` | `<Buffer>` | `<Uint8Array>`
- `encoding` `<string>`
- `callback` `<Function>`
- Returns: `<boolean>`

If this method is called and `response.writeHead()` has not been called, it will switch to implicit header mode and flush the implicit headers.

This sends a chunk of the response body. This method may be called multiple times to provide successive parts of the body.

In the `http` module, the response body is omitted when the request is a HEAD request. Similarly, the `204` and `304` responses *must not* include a message body.

`chunk` can be a string or a buffer. If `chunk` is a string, the second parameter specifies how to encode it into a byte stream. By default the `encoding` is `'utf8'`. `callback` will be called when this chunk of data is flushed.

This is the raw HTTP body and has nothing to do with higher-level multi-part body encodings that may be used.

The first time `response.write()` is called, it will send the buffered header information and the first chunk of the body to the client. The second time `response.write()` is called, Node.js assumes data will be streamed, and sends the new data separately. That is, the response is buffered up to the first chunk of the body.

Returns `true` if the entire data was flushed successfully to the kernel buffer. Returns `false` if all or part of the data was queued in user memory. `'drain'` will be emitted when the buffer is free again.

`response.writeContinue()`

Sends a status `100 Continue` to the client, indicating that the request body should be sent. See the `'checkContinue'` event on `Http2Server` and `Http2SecureServer`.

`response.writeHead(statusCode[, statusMessage][, headers])`

- `statusCode` `<number>`
- `statusMessage` `<string>`
- `headers` `<Object>`
- Returns: `<http2.Http2ServerResponse>`

Sends a response header to the request. The status code is a 3-digit HTTP status code, like `404`. The last argument, `headers`, are the response headers.

Returns a reference to the `Http2ServerResponse`, so that calls can be chained.

For compatibility with `HTTP/1`, a human-readable `statusMessage` may be passed as the second argument. However, because the `statusMessage` has no meaning within `HTTP/2`, the argument will have no effect and a process warning will be emitted.

```
const body = 'hello world';
response.writeHead(200, {
  'Content-Length': Buffer.byteLength(body),
  'Content-Type': 'text/plain; charset=utf-8',
});
```

`Content-Length` is given in bytes not characters. The `Buffer.byteLength()` API may be used to determine the number of bytes in a given encoding. On outbound messages, Node.js does not check if `Content-Length` and the length of the body being transmitted are equal or not. However, when receiving messages, Node.js will automatically reject messages when the `Content-Length` does not match the actual payload size.

This method may be called at most one time on a message before `response.end()` is called.

If `response.write()` or `response.end()` are called before calling this, the implicit/mutable headers will be calculated and call this function.

When headers have been set with `response.setHeader()`, they will be merged with any headers passed to `response.writeHead()`, with the headers passed to `response.writeHead()` given precedence.

```
// Returns content-type = text/plain
const server = http2.createServer((req, res) => {
  res.setHeader('Content-Type', 'text/html; charset=utf-8');
  res.setHeader('X-Foo', 'bar');
  res.writeHead(200, { 'Content-Type': 'text/plain; charset=utf-8' });
  res.end('ok');
});
```

Attempting to set a header field name or value that contains invalid characters will result in a `TypeError` being thrown.

Collecting HTTP/2 performance metrics

The `PerformanceObserver` API can be used to collect basic performance metrics for each `Http2Session` and `Http2Stream` instance.

```
const { PerformanceObserver } = require('perf_hooks');

const obs = new PerformanceObserver((items) => {
  const entry = items.getEntries()[0];
  console.log(entry.entryType); // prints 'http2'
  if (entry.name === 'Http2Session') {
    // Entry contains statistics about the Http2Session
  } else if (entry.name === 'Http2Stream') {
    // Entry contains statistics about the Http2Stream
  }
});
obs.observe({ entryTypes: ['http2'] });
```

The `entryType` property of the `PerformanceEntry` will be equal to `'http2'`.

The `name` property of the `PerformanceEntry` will be equal to either `'Http2Stream'` or `'Http2Session'`.

If `name` is equal to `Http2Stream`, the `PerformanceEntry` will contain the following additional properties:

- `bytesRead` `<number>` The number of `DATA` frame bytes received for this `Http2Stream`.
- `bytesWritten` `<number>` The number of `DATA` frame bytes sent for this `Http2Stream`.
- `id` `<number>` The identifier of the associated `Http2Stream`
- `timeToFirstByte` `<number>` The number of milliseconds elapsed between the `PerformanceEntry` `startTime` and the reception of the first `DATA` frame.
- `timeToFirstByteSent` `<number>` The number of milliseconds elapsed between the `PerformanceEntry` `startTime` and sending of the first `DATA` frame.
- `timeToFirstHeader` `<number>` The number of milliseconds elapsed between the `PerformanceEntry` `startTime` and the reception of the first header.

If `name` is equal to `Http2Session`, the `PerformanceEntry` will contain the following additional properties:

- `bytesRead` `<number>` The number of bytes received for this `Http2Session`.
- `bytesWritten` `<number>` The number of bytes sent for this `Http2Session`.
- `framesReceived` `<number>` The number of HTTP/2 frames received by the `Http2Session`.
- `framesSent` `<number>` The number of HTTP/2 frames sent by the `Http2Session`.
- `maxConcurrentStreams` `<number>` The maximum number of streams concurrently open during the lifetime of the `Http2Session`.
- `pingRTT` `<number>` The number of milliseconds elapsed since the transmission of a `PING` frame and the reception of its acknowledgment. Only present if a `PING` frame has been sent on the `Http2Session`.
- `streamAverageDuration` `<number>` The average duration (in milliseconds) for all `Http2Stream` instances.
- `streamCount` `<number>` The number of `Http2Stream` instances processed by the `Http2Session`.
- `type` `<string>` Either `'server'` or `'client'` to identify the type of `Http2Session`.

Note on `:authority` and `host`

HTTP/2 requires requests to have either the `:authority` pseudo-header or the `host` header. Prefer `:authority` when constructing an HTTP/2 request directly, and `host` when converting from HTTP/1 (in proxies, for instance).

The compatibility API falls back to `host` if `:authority` is not present. See `request.authority` for more information. However, if you don't use the compatibility API (or use `req.headers` directly), you need to implement any fall-back behavior yourself.

HTTPS

Stability: 2 - Stable

Source Code: <lib/https.js>

HTTPS is the HTTP protocol over TLS/SSL. In Node.js this is implemented as a separate module.

Class: `https.Agent`

An `Agent` object for HTTPS similar to `http.Agent`. See `https.request()` for more information.

`new Agent([options])`

- `options` `<Object>` Set of configurable options to set on the agent. Can have the same fields as for `http.Agent(options)`, and
 - `maxCachedSessions` `<number>` maximum number of TLS cached sessions. Use `0` to disable TLS session caching. **Default:** `100`.
 - `servername` `<string>` the value of `Server Name Indication extension` to be sent to the server. Use empty string `''` to disable sending the extension. **Default:** host name of the target server, unless the target server is specified using an IP address, in which case the default is `''` (no extension).

See `Session Resumption` for information about TLS session reuse.

Event: `'keylog'`

- `line` `<Buffer>` Line of ASCII text, in NSS `SSLKEYLOGFILE` format.
- `tlsSocket` `<tls.TLSSocket>` The `tls.TLSSocket` instance on which it was generated.

The `keylog` event is emitted when key material is generated or received by a connection managed by this agent (typically before handshake has completed, but not necessarily). This keying material can be stored for debugging, as it allows captured TLS traffic to be decrypted. It may be emitted multiple times for each socket.

A typical use case is to append received lines to a common text file, which is later used by software (such as Wireshark) to decrypt the traffic:

```
// ...
https.globalAgent.on('keylog', (line, tlsSocket) => {
  fs.appendFileSync('/tmp/ssl-keys.log', line, { mode: 0o600 });
});
```

Class: `https.Server`

- Extends: `<tls.Server>`

See `http.Server` for more information.

server.close([callback])

- `callback` `<Function>`
- Returns: `<https.Server>`

See `server.close()` from the HTTP module for details.

server.headersTimeout

- `<number>` Default: `60000`

See `http.Server#headersTimeout`.

server.listen()

Starts the HTTPS server listening for encrypted connections. This method is identical to `server.listen()` from `net.Server`.

server.maxHeadersCount

- `<number>` Default: `2000`

See `http.Server#maxHeadersCount`.

server.requestTimeout

- `<number>` Default: `0`

See `http.Server#requestTimeout`.

server.setTimeout([msecs][, callback])

- `msecs` `<number>` Default: `120000` (2 minutes)
- `callback` `<Function>`
- Returns: `<https.Server>`

See `http.Server#setTimeout()`.

server.timeout

- `<number>` Default: `0` (no timeout)

See `http.Server#timeout`.

server.keepAliveTimeout

- `<number>` Default: `5000` (5 seconds)

See `http.Server#keepAliveTimeout`.

https.createServer([options][, requestListener])

- `options` `<Object>` Accepts options from `tls.createServer()`, `tls.createSecureContext()` and `http.createServer()`.
- `requestListener` `<Function>` A listener to be added to the `'request'` event.
- Returns: `<https.Server>`

```
// curl -k https://localhost:8000/  
const https = require('https');
```

```
const fs = require('fs');

const options = {
  key: fs.readFileSync('test/fixtures/keys/agent2-key.pem'),
  cert: fs.readFileSync('test/fixtures/keys/agent2-cert.pem')
};

https.createServer(options, (req, res) => {
  res.writeHead(200);
  res.end('hello world\n');
}).listen(8000);
```

Or

```
const https = require('https');
const fs = require('fs');

const options = {
  pfx: fs.readFileSync('test/fixtures/test_cert.pfx'),
  passphrase: 'sample'
};

https.createServer(options, (req, res) => {
  res.writeHead(200);
  res.end('hello world\n');
}).listen(8000);
```

https.get(options[, callback])

https.get(url[, options][, callback])

- `url` `<string>` | `<URL>`
- `options` `<Object>` | `<string>` | `<URL>` Accepts the same `options` as `https.request()`, with the `method` always set to `GET`.
- `callback` `<Function>`

Like `http.get()` but for HTTPS.

`options` can be an object, a string, or a `URL` object. If `options` is a string, it is automatically parsed with `new URL()`. If it is a `URL` object, it will be automatically converted to an ordinary `options` object.

```
const https = require('https');

https.get('https://encrypted.google.com/', (res) => {
  console.log('statusCode:', res.statusCode);
  console.log('headers:', res.headers);

  res.on('data', (d) => {
    process.stdout.write(d);
  });
});
```



```
}).on('error', (e) => {
  console.error(e);
});
```

https.globalAgent

Global instance of `https.Agent` for all HTTPS client requests.

https.request(options[, callback])

https.request(url[, options][, callback])

- `url` `<string>` | `<URL>`
- `options` `<Object>` | `<string>` | `<URL>` Accepts all options from `http.request()`, with some differences in default values:
 - `protocol` **Default:** `'https:'`
 - `port` **Default:** `443`
 - `agent` **Default:** `https.globalAgent`
- `callback` `<Function>`
- Returns: `<http.ClientRequest>`

Makes a request to a secure web server.

The following additional options from `tls.connect()` are also accepted: `ca`, `cert`, `ciphers`, `clientCertEngine`, `crl`, `dhparam`, `ecdhCurve`, `honorCipherOrder`, `key`, `passphrase`, `pfx`, `rejectUnauthorized`, `secureOptions`, `secureProtocol`, `servername`, `sessionIdContext`, `highWaterMark`.

`options` can be an object, a string, or a `URL` object. If `options` is a string, it is automatically parsed with `new URL()`. If it is a `URL` object, it will be automatically converted to an ordinary `options` object.

`https.request()` returns an instance of the `http.ClientRequest` class. The `ClientRequest` instance is a writable stream. If one needs to upload a file with a POST request, then write to the `ClientRequest` object.

```
const https = require('https');

const options = {
  hostname: 'encrypted.google.com',
  port: 443,
  path: '/',
  method: 'GET'
};

const req = https.request(options, (res) => {
  console.log('statusCode:', res.statusCode);
  console.log('headers:', res.headers);

  res.on('data', (d) => {
    process.stdout.write(d);
  });
});
```

```
});

req.on('error', (e) => {
  console.error(e);
});

req.end();
```

Example using options from `tls.connect()`:

```
const options = {
  hostname: 'encrypted.google.com',
  port: 443,
  path: '/',
  method: 'GET',
  key: fs.readFileSync('test/fixtures/keys/agent2-key.pem'),
  cert: fs.readFileSync('test/fixtures/keys/agent2-cert.pem')
};

options.agent = new https.Agent(options);

const req = https.request(options, (res) => {
  // ...
});
```

Alternatively, opt out of connection pooling by not using an `Agent`.

```
const options = {
  hostname: 'encrypted.google.com',
  port: 443,
  path: '/',
  method: 'GET',
  key: fs.readFileSync('test/fixtures/keys/agent2-key.pem'),
  cert: fs.readFileSync('test/fixtures/keys/agent2-cert.pem'),
  agent: false
};

const req = https.request(options, (res) => {
  // ...
});
```

Example using a `URL` as `options`:

```
const options = new URL('https://abc:xyz@example.com');

const req = https.request(options, (res) => {
  // ...
});
```

Example pinning on certificate fingerprint, or the public key (similar to `pin-sha256`):

```

const tls = require('tls');
const https = require('https');
const crypto = require('crypto');

function sha256(s) {
  return crypto.createHash('sha256').update(s).digest('base64');
}

const options = {
  hostname: 'github.com',
  port: 443,
  path: '/',
  method: 'GET',
  checkServerIdentity: function(host, cert) {
    // Make sure the certificate is issued to the host we are connected to
    const err = tls.checkServerIdentity(host, cert);
    if (err) {
      return err;
    }

    // Pin the public key, similar to HPKP pin-sha25 pinning
    const pubkey256 = 'pL1+qb9HTMRZJmuC/bB/ZI9d302BYrrqiVuRyW+DGrU=';
    if (sha256(cert.pubkey) !== pubkey256) {
      const msg = 'Certificate verification error: ' +
        `The public key of '${cert.subject.CN}' ` +
        'does not match our pinned fingerprint';
      return new Error(msg);
    }

    // Pin the exact certificate, rather than the pub key
    const cert256 = '25:FE:39:32:D9:63:8C:8A:FC:A1:9A:29:87:' +
      'D8:3E:4C:1D:98:DB:71:E4:1A:48:03:98:EA:22:6A:BD:8B:93:16';
    if (cert.fingerprint256 !== cert256) {
      const msg = 'Certificate verification error: ' +
        `The certificate of '${cert.subject.CN}' ` +
        'does not match our pinned fingerprint';
      return new Error(msg);
    }

    // This loop is informational only.
    // Print the certificate and public key fingerprints of all certs in the
    // chain. Its common to pin the public key of the issuer on the public
    // internet, while pinning the public key of the service in sensitive
    // environments.
    do {
      console.log('Subject Common Name:', cert.subject.CN);
      console.log(' Certificate SHA256 fingerprint:', cert.fingerprint256);

      hash = crypto.createHash('sha256');
      console.log(' Public key ping-sha256:', sha256(cert.pubkey));
    } while (cert.issuerCertificate);
  }
};

```

```

    lastprint256 = cert.fingerprint256;
    cert = cert.issuerCertificate;
  } while (cert.fingerprint256 !== lastprint256);

  },
};

options.agent = new https.Agent(options);
const req = https.request(options, (res) => {
  console.log('All OK. Server matched our pinned cert or public key');
  console.log('statusCode:', res.statusCode);
  // Print the HPKP values
  console.log('headers:', res.headers['public-key-pins']);

  res.on('data', (d) => {});
});

req.on('error', (e) => {
  console.error(e.message);
});
req.end();

```

Outputs for example:

```

Subject Common Name: github.com
  Certificate SHA256 fingerprint: 25:FE:39:32:D9:63:8C:8A:FC:A1:9A:29:87:D8:3E:4C:1D:98:DB:71:E4:1A:48:03:98:EA:22:6A:BD:
  Public key ping-sha256: pL1+qb9HTMRZJmUC/bB/ZI9d302BYrrqiVuRyW+DGrU=
Subject Common Name: DigiCert SHA2 Extended Validation Server CA
  Certificate SHA256 fingerprint: 40:3E:06:2A:26:53:05:91:13:28:5B:AF:80:A0:D4:AE:42:2C:84:8C:9F:78:FA:D0:1F:C9:4B:C5:B8:
  Public key ping-sha256: RRM1dGqnDFsCJXBTHky16vi1ob0lCgFFn/y0hI/y+ho=
Subject Common Name: DigiCert High Assurance EV Root CA
  Certificate SHA256 fingerprint: 74:31:E5:F4:C3:C1:CE:46:90:77:4F:0B:61:E0:54:40:88:3B:A9:A0:1E:D0:0B:A6:AB:D7:80:6E:D3:
  Public key ping-sha256: WoiWRyIOVNa9ihaBciRSC7XHjliYS9VwUGOIud4PB18=
All OK. Server matched our pinned cert or public key
statusCode: 200
headers: max-age=0; pin-sha256="WoiWRyIOVNa9ihaBciRSC7XHjliYS9VwUGOIud4PB18="; pin-sha256="RRM1dGqnDFsCJXBTHky16vi1ob0lCg

```

Inspector

Stability: 2 - Stable

Source Code: [lib/inspector.js](#)

The `inspector` module provides an API for interacting with the V8 inspector.

It can be accessed using:

```
const inspector = require('inspector');
```

inspector.close()

Deactivate the inspector. Blocks until there are no active connections.

inspector.console

- `<Object>` An object to send messages to the remote inspector console.

```
require('inspector').console.log('a message');
```

The inspector console does not have API parity with Node.js console.

inspector.open([port[, host[, wait]]])

- `port` `<number>` Port to listen on for inspector connections. Optional. **Default:** what was specified on the CLI.
- `host` `<string>` Host to listen on for inspector connections. Optional. **Default:** what was specified on the CLI.
- `wait` `<boolean>` Block until a client has connected. Optional. **Default:** `false`.

Activate inspector on host and port. Equivalent to `node --inspect=[host:]port`, but can be done programmatically after node has started.

If wait is `true`, will block until a client has connected to the inspect port and flow control has been passed to the debugger client.

See the [security warning](#) regarding the `host` parameter usage.

inspector.url()

- Returns: `<string>` | `<undefined>`

Return the URL of the active inspector, or `undefined` if there is none.

```
$ node --inspect -p 'inspector.url()'
Debugger listening on ws://127.0.0.1:9229/166e272e-7a30-4d09-97ce-f1c012b43c34
For help see https://nodejs.org/en/docs/inspector
ws://127.0.0.1:9229/166e272e-7a30-4d09-97ce-f1c012b43c34
```

```
$ node --inspect=localhost:3000 -p 'inspector.url()'
Debugger listening on ws://localhost:3000/51cf8d0e-3c36-4c59-8efd-54519839e56a
For help see https://nodejs.org/en/docs/inspector
ws://localhost:3000/51cf8d0e-3c36-4c59-8efd-54519839e56a
```

```
$ node -p 'inspector.url()'
undefined
```

inspector.waitForDebugger()

Blocks until a client (existing or connected later) has sent `Runtime.runIfWaitingForDebugger` command.

An exception will be thrown if there is no active inspector.

Class: `inspector.Session`

- Extends: `<EventEmitter>`

The `inspector.Session` is used for dispatching messages to the V8 inspector back-end and receiving message responses and notifications.

`new inspector.Session()`

Create a new instance of the `inspector.Session` class. The inspector session needs to be connected through `session.connect()` before the messages can be dispatched to the inspector backend.

Event: `'inspectorNotification'`

- `<Object>` The notification message object

Emitted when any notification from the V8 Inspector is received.

```
session.on('inspectorNotification', (message) => console.log(message.method));  
// Debugger.paused  
// Debugger.resumed
```

It is also possible to subscribe only to notifications with specific method:

Event: `<inspector-protocol-method>` ;

- `<Object>` The notification message object

Emitted when an inspector notification is received that has its method field set to the `<inspector-protocol-method>` value.

The following snippet installs a listener on the `'Debugger.paused'` event, and prints the reason for program suspension whenever program execution is suspended (through breakpoints, for example):

```
session.on('Debugger.paused', ({ params }) => {  
  console.log(params.hitBreakpoints);  
});  
// [ '/the/file/that/has/the/breakpoint.js:11:0' ]
```

`session.connect()`

Connects a session to the inspector back-end.

`session.connectToMainThread()`

Connects a session to the main thread inspector back-end. An exception will be thrown if this API was not called on a Worker thread.

`session.disconnect()`

Immediately close the session. All pending message callbacks will be called with an error. `session.connect()` will need to be called to be able to send messages again. Reconnected session will lose all inspector state, such as enabled agents or configured breakpoints.

`session.post(method[, params][, callback])`

- `method` `<string>`
- `params` `<Object>`
- `callback` `<Function>`

Posts a message to the inspector back-end. `callback` will be notified when a response is received. `callback` is a function that accepts two optional arguments: error and message-specific result.

```
session.post('Runtime.evaluate', { expression: '2 + 2' },
            (error, { result }) => console.log(result));
// Output: { type: 'number', value: 4, description: '4' }
```

The latest version of the V8 inspector protocol is published on the [Chrome DevTools Protocol Viewer](#).

Node.js inspector supports all the Chrome DevTools Protocol domains declared by V8. Chrome DevTools Protocol domain provides an interface for interacting with one of the runtime agents used to inspect the application state and listen to the run-time events.

Example usage

Apart from the debugger, various V8 Profilers are available through the DevTools protocol.

CPU profiler

Here's an example showing how to use the [CPU Profiler](#):

```
const inspector = require('inspector');
const fs = require('fs');
const session = new inspector.Session();
session.connect();

session.post('Profiler.enable', () => {
  session.post('Profiler.start', () => {
    // Invoke business logic under measurement here...

    // some time later...
    session.post('Profiler.stop', (err, { profile }) => {
      // Write profile to disk, upload, etc.
      if (!err) {
        fs.writeFileSync('./profile.cpubprofile', JSON.stringify(profile));
      }
    });
  });
});
```

Heap profiler

Here's an example showing how to use the [Heap Profiler](#):

```
const inspector = require('inspector');
const fs = require('fs');
const session = new inspector.Session();
```

```
const fd = fs.openSync('profile.heapsnapshot', 'w');

session.connect();

session.on('HeapProfiler.addHeapSnapshotChunk', (m) => {
  fs.writeSync(fd, m.params.chunk);
});

session.post('HeapProfiler.takeHeapSnapshot', null, (err, r) => {
  console.log('HeapProfiler.takeHeapSnapshot done:', err, r);
  session.disconnect();
  fs.closeSync(fd);
});
```

Internationalization support

Node.js has many features that make it easier to write internationalized programs. Some of them are:

- Locale-sensitive or Unicode-aware functions in the [ECMAScript Language Specification](#) :
 - `String.prototype.normalize()`
 - `String.prototype.toLowerCase()`
 - `String.prototype.toUpperCase()`
- All functionality described in the [ECMAScript Internationalization API Specification](#) (aka ECMA-402):
 - `Intl` object
 - Locale-sensitive methods like `String.prototype.localeCompare()` and `Date.prototype.toLocaleString()`
- The [WHATWG URL parser](#)'s internationalized domain names (IDNs) support
- `require('buffer').transcode()`
- More accurate [REPL](#) line editing
- `require('util').TextDecoder`
- [RegExp Unicode Property Escapes](#)

Node.js and the underlying V8 engine use [International Components for Unicode \(ICU\)](#) to implement these features in native C/C++ code. The full ICU data set is provided by Node.js by default. However, due to the size of the ICU data file, several options are provided for customizing the ICU data set either when building or running Node.js.

Options for building Node.js

To control how ICU is used in Node.js, four `configure` options are available during compilation. Additional details on how to compile Node.js are documented in [BUILDING.md](#).

- `--with-intl=none` / `--without-intl`
- `--with-intl=system-icu`
- `--with-intl=small-icu`
- `--with-intl=full-icu` (default)

An overview of available Node.js and JavaScript features for each `configure` option:

Feature	none	system-icu	small-icu	full-icu
<code>String.prototype.normalize()</code>	none (function is no-op)	full	full	full
<code>String.prototype.to*Case()</code>	full	full	full	full
<code>Intl</code>	none (object does not exist)	partial/full (depends on OS)	partial (English-only)	full
<code>String.prototype.localeCompare()</code>	partial (not locale-aware)	full	full	full
<code>String.prototype.toLocale*Case()</code>	partial (not locale-aware)	full	full	full
<code>Number.prototype.toLocaleString()</code>	partial (not locale-aware)	partial/full (depends on OS)	partial (English-only)	full
<code>Date.prototype.toLocale*String()</code>	partial (not locale-aware)	partial/full (depends on OS)	partial (English-only)	full
Legacy URL Parser	partial (no IDN support)	full	full	full
WHATWG URL Parser	partial (no IDN support)	full	full	full
<code>require('buffer').transcode()</code>	none (function does not exist)	full	full	full
REPL	partial (inaccurate line editing)	full	full	full
<code>require('util').TextDecoder</code>	partial (basic encodings support)	partial/full (depends on OS)	partial (Unicode-only)	full
RegExp Unicode Property Escapes	none (invalid <code>RegExp</code> error)	full	full	full

The "(not locale-aware)" designation denotes that the function carries out its operation just like the non-`Locale` version of the function, if one exists. For example, under `none` mode, `Date.prototype.toLocaleString()`'s operation is identical to that of `Date.prototype.toString()`.

Disable all internationalization features (`none`)

If this option is chosen, ICU is disabled and most internationalization features mentioned above will be **unavailable** in the resulting `node` binary.

Build with a pre-installed ICU (`system-icu`)

Node.js can link against an ICU build already installed on the system. In fact, most Linux distributions already come with ICU installed, and this option would make it possible to reuse the same set of data used by other components in the OS.

Functionalities that only require the ICU library itself, such as `String.prototype.normalize()` and the **WHATWG URL parser**, are fully supported under `system-icu`. Features that require ICU locale data in addition, such as `Intl.DateTimeFormat` may be fully or partially supported, depending on the completeness of the ICU data installed on the system.

Embed a limited set of ICU data (`small-icu`)

This option makes the resulting binary link against the ICU library statically, and includes a subset of ICU data (typically only the English locale) within the `node` executable.

Functionalities that only require the ICU library itself, such as `String.prototype.normalize()` and the [WHATWG URL parser](#), are fully supported under `small-icu`. Features that require ICU locale data in addition, such as `Intl.DateTimeFormat`, generally only work with the English locale:

```
const january = new Date(9e8);
const english = new Intl.DateTimeFormat('en', { month: 'long' });
const spanish = new Intl.DateTimeFormat('es', { month: 'long' });

console.log(english.format(january));
// Prints "January"
console.log(spanish.format(january));
// Prints "M01" on small-icu
// Should print "enero"
```

This mode provides a balance between features and binary size.

Providing ICU data at runtime

If the `small-icu` option is used, one can still provide additional locale data at runtime so that the JS methods would work for all ICU locales. Assuming the data file is stored at `/some/directory`, it can be made available to ICU through either:

- The `NODE_ICU_DATA` environment variable:

```
env NODE_ICU_DATA=/some/directory node
```

- The `--icu-data-dir` CLI parameter:

```
node --icu-data-dir=/some/directory
```

(If both are specified, the `--icu-data-dir` CLI parameter takes precedence.)

ICU is able to automatically find and load a variety of data formats, but the data must be appropriate for the ICU version, and the file correctly named. The most common name for the data file is `icudt6X[b].dat`, where `6X` denotes the intended ICU version, and `b` or `l` indicates the system's endianness. Check ["ICU Data"](#) article in the ICU User Guide for other supported formats and more details on ICU data in general.

The `full-icu` npm module can greatly simplify ICU data installation by detecting the ICU version of the running `node` executable and downloading the appropriate data file. After installing the module through `npm i full-icu`, the data file will be available at `./node_modules/full-icu`. This path can be then passed either to `NODE_ICU_DATA` or `--icu-data-dir` as shown above to enable full `Intl` support.

Embed the entire ICU (full-icu)

This option makes the resulting binary link against ICU statically and include a full set of ICU data. A binary created this way has no further external dependencies and supports all locales, but might be rather large. This is the default behavior if no `--with-intl` flag is passed. The official binaries are also built in this mode.

Detecting internationalization support

To verify that ICU is enabled at all (`system-icu`, `small-icu`, or `full-icu`), simply checking the existence of `Intl` should suffice:

```
const hasICU = typeof Intl === 'object';
```

Alternatively, checking for `process.versions.icu`, a property defined only when ICU is enabled, works too:

```
const hasICU = typeof process.versions.icu === 'string';
```

To check for support for a non-English locale (i.e. `full-icu` or `system-icu`), `Intl.DateTimeFormat` can be a good distinguishing factor:

```
const hasFullICU = (() => {
  try {
    const january = new Date(9e8);
    const spanish = new Intl.DateTimeFormat('es', { month: 'long' });
    return spanish.format(january) === 'enero';
  } catch (err) {
    return false;
  }
})();
```

For more verbose tests for `Intl` support, the following resources may be found to be helpful:

- [btest402](#): Generally used to check whether Node.js with `Intl` support is built correctly.
- [Test262](#): ECMAScript's official conformance test suite includes a section dedicated to ECMA-402.

Modules: CommonJS modules

Stability: 2 - Stable

In the Node.js module system, each file is treated as a separate module. For example, consider a file named `foo.js`:

```
const circle = require('./circle.js');
console.log(`The area of a circle of radius 4 is ${circle.area(4)}`);
```

On the first line, `foo.js` loads the module `circle.js` that is in the same directory as `foo.js`.

Here are the contents of `circle.js`:

```
const { PI } = Math;

exports.area = (r) => PI * r ** 2;

exports.circumference = (r) => 2 * PI * r;
```

The module `circle.js` has exported the functions `area()` and `circumference()`. Functions and objects are added to the root of a module by specifying additional properties on the special `exports` object.

Variables local to the module will be private, because the module is wrapped in a function by Node.js (see [module wrapper](#)). In this example, the variable `PI` is private to `circle.js`.

The `module.exports` property can be assigned a new value (such as a function or object).

Below, `bar.js` makes use of the `square` module, which exports a `Square` class:

```
const Square = require('./square.js');
const mySquare = new Square(2);
console.log(`The area of mySquare is ${mySquare.area()}`);
```

The `square` module is defined in `square.js`:

```
// Assigning to exports will not modify module, must use module.exports
module.exports = class Square {
  constructor(width) {
    this.width = width;
  }

  area() {
    return this.width ** 2;
  }
};
```

The module system is implemented in the `require('module')` module.

Accessing the main module

When a file is run directly from Node.js, `require.main` is set to its `module`. That means that it is possible to determine whether a file has been run directly by testing `require.main === module`.

For a file `foo.js`, this will be `true` if run via `node foo.js`, but `false` if run by `require('./foo')`.

Because `module` provides a `filename` property (normally equivalent to `__filename`), the entry point of the current application can be obtained by checking `require.main.filename`.

Package manager tips

The semantics of the Node.js `require()` function were designed to be general enough to support reasonable directory structures. Package manager programs such as `dpkg`, `rpm`, and `npm` will hopefully find it possible to build native packages from Node.js modules without modification.

Below we give a suggested directory structure that could work:

Let's say that we wanted to have the folder at `/usr/lib/node/<some-package>/<some-version>` hold the contents of a specific version of a package.

Packages can depend on one another. In order to install package `foo`, it may be necessary to install a specific version of package `bar`. The `bar` package may itself have dependencies, and in some cases, these may even collide or form cyclic dependencies.

Because Node.js looks up the `realpath` of any modules it loads (that is, it resolves symlinks) and then [looks for their dependencies in `node_modules` folders](#), this situation can be resolved with the following architecture:

- `/usr/lib/node/foo/1.2.3/`: Contents of the `foo` package, version 1.2.3.
- `/usr/lib/node/bar/4.3.2/`: Contents of the `bar` package that `foo` depends on.
- `/usr/lib/node/foo/1.2.3/node_modules/bar`: Symbolic link to `/usr/lib/node/bar/4.3.2/`.

- `/usr/lib/node/bar/4.3.2/node_modules/*`: Symbolic links to the packages that `bar` depends on.

Thus, even if a cycle is encountered, or if there are dependency conflicts, every module will be able to get a version of its dependency that it can use.

When the code in the `foo` package does `require('bar')`, it will get the version that is symlinked into `/usr/lib/node/foo/1.2.3/node_modules/bar`. Then, when the code in the `bar` package calls `require('quux')`, it'll get the version that is symlinked into `/usr/lib/node/bar/4.3.2/node_modules/quux`.

Furthermore, to make the module lookup process even more optimal, rather than putting packages directly in `/usr/lib/node`, we could put them in `/usr/lib/node_modules/<name>/<version>`. Then Node.js will not bother looking for missing dependencies in `/usr/node_modules` or `/node_modules`.

In order to make modules available to the Node.js REPL, it might be useful to also add the `/usr/lib/node_modules` folder to the `$NODE_PATH` environment variable. Since the module lookups using `node_modules` folders are all relative, and based on the real path of the files making the calls to `require()`, the packages themselves can be anywhere.

The `.mjs` extension

It is not possible to `require()` files that have the `.mjs` extension. Attempting to do so will throw an error. The `.mjs` extension is reserved for [ECMAScript Modules](#) which cannot be loaded via `require()`. See [ECMAScript Modules](#) for more details.

All together...

To get the exact filename that will be loaded when `require()` is called, use the `require.resolve()` function.

Putting together all of the above, here is the high-level algorithm in pseudocode of what `require()` does:

```
require(X) from module at path Y
1. If X is a core module,
  a. return the core module
  b. STOP
2. If X begins with '/'
  a. set Y to be the filesystem root
3. If X begins with './' or '/' or '../'
  a. LOAD_AS_FILE(Y + X)
  b. LOAD_AS_DIRECTORY(Y + X)
  c. THROW "not found"
4. If X begins with '#'
  a. LOAD_PACKAGE_IMPORTS(X, dirname(Y))
5. LOAD_PACKAGE_SELF(X, dirname(Y))
6. LOAD_NODE_MODULES(X, dirname(Y))
7. THROW "not found"

LOAD_AS_FILE(X)
1. If X is a file, load X as its file extension format. STOP
2. If X.js is a file, load X.js as JavaScript text. STOP
3. If X.json is a file, parse X.json to a JavaScript Object. STOP
4. If X.node is a file, load X.node as binary addon. STOP

LOAD_INDEX(X)
1. If X/index.js is a file, load X/index.js as JavaScript text. STOP
2. If X/index.json is a file, parse X/index.json to a JavaScript object. STOP
```

3. If X/index.node is a file, load X/index.node as binary addon. STOP

LOAD_AS_DIRECTORY(X)

1. If X/package.json is a file,
 - a. Parse X/package.json, and look for "main" field.
 - b. If "main" is a falsy value, GOTO 2.
 - c. let M = X + (json main field)
 - d. LOAD_AS_FILE(M)
 - e. LOAD_INDEX(M)
 - f. LOAD_INDEX(X) DEPRECATED
 - g. THROW "not found"
2. LOAD_INDEX(X)

LOAD_NODE_MODULES(X, START)

1. let DIRS = NODE_MODULES_PATHS(START)
2. for each DIR in DIRS:
 - a. LOAD_PACKAGE_EXPORTS(X, DIR)
 - b. LOAD_AS_FILE(DIR/X)
 - c. LOAD_AS_DIRECTORY(DIR/X)

NODE_MODULES_PATHS(START)

1. let PARTS = path split(START)
2. let I = count of PARTS - 1
3. let DIRS = [GLOBAL_FOLDERS]
4. while I >= 0,
 - a. if PARTS[I] = "node_modules" CONTINUE
 - b. DIR = path join(PARTS[0 .. I] + "node_modules")
 - c. DIRS = DIRS + DIR
 - d. let I = I - 1
5. return DIRS

LOAD_PACKAGE_IMPORTS(X, DIR)

1. Find the closest package scope SCOPE to DIR.
2. If no scope was found, return.
3. If the SCOPE/package.json "imports" is null or undefined, return.
4. let MATCH = PACKAGE_IMPORTS_RESOLVE(X, pathToFileURL(SCOPE),
["node", "require"]) *defined in the ESM resolver*.
5. RESOLVE_ESM_MATCH(MATCH).

LOAD_PACKAGE_EXPORTS(X, DIR)

1. Try to interpret X as a combination of NAME and SUBPATH where the name may have a @scope/ prefix and the subpath begins with a slash (`/`).
2. If X does not match this pattern or DIR/NAME/package.json is not a file, return.
3. Parse DIR/NAME/package.json, and look for "exports" field.
4. If "exports" is null or undefined, return.
5. let MATCH = PACKAGE_EXPORTS_RESOLVE(pathToFileURL(DIR/NAME), "." + SUBPATH,
`package.json` "exports", ["node", "require"]) *defined in the ESM resolver*.
6. RESOLVE_ESM_MATCH(MATCH)

LOAD_PACKAGE_SELF(X, DIR)

1. Find the closest package scope SCOPE to DIR.
2. If no scope was found, return.
3. If the SCOPE/package.json "exports" is null or undefined, return.

```

4. If the SCOPE/package.json "name" is not the first segment of X, return.
5. let MATCH = PACKAGE_EXPORTS_RESOLVE(pathToFileURL(SCOPE),
   "." + X.slice("name".length), `package.json` "exports", ["node", "require"])
   defined in the ESM resolver.
6. RESOLVE_ESM_MATCH(MATCH)

RESOLVE_ESM_MATCH(MATCH)
1. let { RESOLVED, EXACT } = MATCH
2. let RESOLVED_PATH = fileURLToPath(RESOLVED)
3. If EXACT is true,
   a. If the file at RESOLVED_PATH exists, load RESOLVED_PATH as its extension
      format. STOP
4. Otherwise, if EXACT is false,
   a. LOAD_AS_FILE(RESOLVED_PATH)
   b. LOAD_AS_DIRECTORY(RESOLVED_PATH)
5. THROW "not found"

```

Caching

Modules are cached after the first time they are loaded. This means (among other things) that every call to `require('foo')` will get exactly the same object returned, if it would resolve to the same file.

Provided `require.cache` is not modified, multiple calls to `require('foo')` will not cause the module code to be executed multiple times. This is an important feature. With it, "partially done" objects can be returned, thus allowing transitive dependencies to be loaded even when they would cause cycles.

To have a module execute code multiple times, export a function, and call that function.

Module caching caveats

Modules are cached based on their resolved filename. Since modules may resolve to a different filename based on the location of the calling module (loading from `node_modules` folders), it is not a *guarantee* that `require('foo')` will always return the exact same object, if it would resolve to different files.

Additionally, on case-insensitive file systems or operating systems, different resolved filenames can point to the same file, but the cache will still treat them as different modules and will reload the file multiple times. For example, `require('./foo')` and `require('./F00')` return two different objects, irrespective of whether or not `./foo` and `./F00` are the same file.

Core modules

Node.js has several modules compiled into the binary. These modules are described in greater detail elsewhere in this documentation.

The core modules are defined within the Node.js source and are located in the `lib/` folder.

Core modules are always preferentially loaded if their identifier is passed to `require()`. For instance, `require('http')` will always return the built in HTTP module, even if there is a file by that name.

Core modules can also be identified using the `node:` prefix, in which case it bypasses the `require` cache. For instance, `require('node:http')` will always return the built in HTTP module, even if there is `require.cache` entry by that name.

Cycles

When there are circular `require()` calls, a module might not have finished executing when it is returned.

Consider this situation:

`a.js`:

```
console.log('a starting');
exports.done = false;
const b = require('./b.js');
console.log('in a, b.done = %j', b.done);
exports.done = true;
console.log('a done');
```

`b.js`:

```
console.log('b starting');
exports.done = false;
const a = require('./a.js');
console.log('in b, a.done = %j', a.done);
exports.done = true;
console.log('b done');
```

`main.js`:

```
console.log('main starting');
const a = require('./a.js');
const b = require('./b.js');
console.log('in main, a.done = %j, b.done = %j', a.done, b.done);
```

When `main.js` loads `a.js`, then `a.js` in turn loads `b.js`. At that point, `b.js` tries to load `a.js`. In order to prevent an infinite loop, an **unfinished copy** of the `a.js` exports object is returned to the `b.js` module. `b.js` then finishes loading, and its `exports` object is provided to the `a.js` module.

By the time `main.js` has loaded both modules, they're both finished. The output of this program would thus be:

```
$ node main.js
main starting
a starting
b starting
in b, a.done = false
b done
in a, b.done = true
a done
in main, a.done = true, b.done = true
```

Careful planning is required to allow cyclic module dependencies to work correctly within an application.

File modules

If the exact filename is not found, then Node.js will attempt to load the required filename with the added extensions: `.js`, `.json`, and finally `.node`.

`.js` files are interpreted as JavaScript text files, and `.json` files are parsed as JSON text files. `.node` files are interpreted as compiled add-on modules loaded with `process.dlopen()`.

A required module prefixed with `'/'` is an absolute path to the file. For example, `require('/home/marco/foo.js')` will load the file at `/home/marco/foo.js`.

A required module prefixed with `'./'` is relative to the file calling `require()`. That is, `circle.js` must be in the same directory as `foo.js` for `require('./circle')` to find it.

Without a leading `'/'`, `'./'`, or `'../'` to indicate a file, the module must either be a core module or is loaded from a `node_modules` folder.

If the given path does not exist, `require()` will throw an `Error` with its `code` property set to `'MODULE_NOT_FOUND'`.

Folders as modules

It is convenient to organize programs and libraries into self-contained directories, and then provide a single entry point to those directories. There are three ways in which a folder may be passed to `require()` as an argument.

The first is to create a `package.json` file in the root of the folder, which specifies a `main` module. An example `package.json` file might look like this:

```
{ "name" : "some-library",  
  "main" : "./lib/some-library.js" }
```

If this was in a folder at `./some-library`, then `require('./some-library')` would attempt to load `./some-library/lib/some-library.js`.

This is the extent of the awareness of `package.json` files within Node.js.

If there is no `package.json` file present in the directory, or if the `"main"` entry is missing or cannot be resolved, then Node.js will attempt to load an `index.js` or `index.node` file out of that directory. For example, if there was no `package.json` file in the previous example, then `require('./some-library')` would attempt to load:

- `./some-library/index.js`
- `./some-library/index.node`

If these attempts fail, then Node.js will report the entire module as missing with the default error:

```
Error: Cannot find module 'some-library'
```

Loading from `node_modules` folders

If the module identifier passed to `require()` is not a `core` module, and does not begin with `'/'`, `'../'`, or `'./'`, then Node.js starts at the parent directory of the current module, and adds `/node_modules`, and attempts to load the module from that location. Node.js will not append `node_modules` to a path already ending in `node_modules`.

If it is not found there, then it moves to the parent directory, and so on, until the root of the file system is reached.

For example, if the file at `/home/ry/projects/foo.js` called `require('bar.js')`, then Node.js would look in the following locations, in this order:

- `/home/ry/projects/node_modules/bar.js`

- `/home/ry/node_modules/bar.js`
- `/home/node_modules/bar.js`
- `/node_modules/bar.js`

This allows programs to localize their dependencies, so that they do not clash.

It is possible to require specific files or sub modules distributed with a module by including a path suffix after the module name. For instance `require('example-module/path/to/file')` would resolve `path/to/file` relative to where `example-module` is located. The suffixed path follows the same module resolution semantics.

Loading from the global folders

If the `NODE_PATH` environment variable is set to a colon-delimited list of absolute paths, then Node.js will search those paths for modules if they are not found elsewhere.

On Windows, `NODE_PATH` is delimited by semicolons (`;`) instead of colons.

`NODE_PATH` was originally created to support loading modules from varying paths before the current `module resolution` algorithm was defined.

`NODE_PATH` is still supported, but is less necessary now that the Node.js ecosystem has settled on a convention for locating dependent modules. Sometimes deployments that rely on `NODE_PATH` show surprising behavior when people are unaware that `NODE_PATH` must be set. Sometimes a module's dependencies change, causing a different version (or even a different module) to be loaded as the `NODE_PATH` is searched.

Additionally, Node.js will search in the following list of GLOBAL_FOLDERS:

- 1: `$HOME/.node_modules`
- 2: `$HOME/.node_modules`
- 3: `$PREFIX/lib/node`

Where `$HOME` is the user's home directory, and `$PREFIX` is the Node.js configured `node_prefix`.

These are mostly for historic reasons.

It is strongly encouraged to place dependencies in the local `node_modules` folder. These will be loaded faster, and more reliably.

The module wrapper

Before a module's code is executed, Node.js will wrap it with a function wrapper that looks like the following:

```
(function(exports, require, module, __filename, __dirname) {
  // Module code actually lives in here
});
```

By doing this, Node.js achieves a few things:

- It keeps top-level variables (defined with `var`, `const` or `let`) scoped to the module rather than the global object.
- It helps to provide some global-looking variables that are actually specific to the module, such as:
 - The `module` and `exports` objects that the implementor can use to export values from the module.
 - The convenience variables `__filename` and `__dirname`, containing the module's absolute filename and directory path.

The module scope

`__dirname`

- `<string>`

The directory name of the current module. This is the same as the `path.dirname()` of the `__filename`.

Example: running `node example.js` from `/Users/mjr`

```
console.log(__dirname);  
// Prints: /Users/mjr  
console.log(path.dirname(__filename));  
// Prints: /Users/mjr
```

`__filename`

- `<string>`

The file name of the current module. This is the current module file's absolute path with symlinks resolved.

For a main program this is not necessarily the same as the file name used in the command line.

See `__dirname` for the directory name of the current module.

Examples:

Running `node example.js` from `/Users/mjr`

```
console.log(__filename);  
// Prints: /Users/mjr/example.js  
console.log(__dirname);  
// Prints: /Users/mjr
```

Given two modules: `a` and `b`, where `b` is a dependency of `a` and there is a directory structure of:

- `/Users/mjr/app/a.js`
- `/Users/mjr/app/node_modules/b/b.js`

References to `__filename` within `b.js` will return `/Users/mjr/app/node_modules/b/b.js` while references to `__filename` within `a.js` will return `/Users/mjr/app/a.js`.

`exports`

- `<Object>`

A reference to the `module.exports` that is shorter to type. See the section about the `exports shortcut` for details on when to use `exports` and when to use `module.exports`.

`module`

- `<module>`

A reference to the current module, see the section about the `module object`. In particular, `module.exports` is used for defining what a module exports and makes available through `require()`.

require(id)

- `id` `<string>` module name or path
- Returns: `<any>` exported module content

Used to import modules, `JSON`, and local files. Modules can be imported from `node_modules`. Local modules and JSON files can be imported using a relative path (e.g. `./`, `./foo`, `./bar/baz`, `../foo`) that will be resolved against the directory named by `__dirname` (if defined) or the current working directory. The relative paths of POSIX style are resolved in an OS independent fashion, meaning that the examples above will work on Windows in the same way they would on Unix systems.

```
// Importing a local module with a path relative to the `__dirname` or current
// working directory. (On Windows, this would resolve to .\path\myLocalModule.)
const myLocalModule = require('./path/myLocalModule');

// Importing a JSON file:
const jsonData = require('./path/filename.json');

// Importing a module from node_modules or Node.js built-in module:
const crypto = require('crypto');
```

require.cache

- `<Object>`

Modules are cached in this object when they are required. By deleting a key value from this object, the next `require` will reload the module. This does not apply to `native addons`, for which reloading will result in an error.

Adding or replacing entries is also possible. This cache is checked before native modules and if a name matching a native module is added to the cache, only `node: -`prefixed require calls are going to receive the native module. Use with care!

```
const assert = require('assert');
const realFs = require('fs');

const fakeFs = {};
require.cache.fs = { exports: fakeFs };

assert.strictEqual(require('fs'), fakeFs);
assert.strictEqual(require('node:fs'), realFs);
```

require.extensions

Stability: 0 - Deprecated

- `<Object>`

Instruct `require` on how to handle certain file extensions.

Process files with the extension `.sjs` as `.js`:

```
require.extensions['.sjs'] = require.extensions['.js'];
```

Deprecated. In the past, this list has been used to load non-JavaScript modules into Node.js by compiling them on-demand. However, in practice, there are much better ways to do this, such as loading modules via some other Node.js program, or compiling them to JavaScript ahead of time.

Avoid using `require.extensions`. Use could cause subtle bugs and resolving the extensions gets slower with each registered extension.

`require.main`

- `<module>`

The `Module` object representing the entry script loaded when the Node.js process launched. See "[Accessing the main module](#)".

In `entry.js` script:

```
console.log(require.main);
```

```
node entry.js
```

```
Module {
  id: '.',
  path: '/absolute/path/to',
  exports: {},
  filename: '/absolute/path/to/entry.js',
  loaded: false,
  children: [],
  paths:
    [ '/absolute/path/to/node_modules',
      '/absolute/path/node_modules',
      '/absolute/node_modules',
      '/node_modules' ] }
```

`require.resolve(request[, options])`

- `request` `<string>` The module path to resolve.
- `options` `<Object>`
 - `paths` `<string[]>` Paths to resolve module location from. If present, these paths are used instead of the default resolution paths, with the exception of `GLOBAL_FOLDERS` like `$HOME/.node_modules`, which are always included. Each of these paths is used as a starting point for the module resolution algorithm, meaning that the `node_modules` hierarchy is checked from this location.
- Returns: `<string>`

Use the internal `require()` machinery to look up the location of a module, but rather than loading the module, just return the resolved filename.

If the module can not be found, a `MODULE_NOT_FOUND` error is thrown.

`require.resolve.paths(request)`

- `request` `<string>` The module path whose lookup paths are being retrieved.
- Returns: `<string[]> | <null>`

Returns an array containing the paths searched during resolution of `request` or `null` if the `request` string references a core module, for example `http` or `fs`.

The module object

- `<Object>`

In each module, the `module` free variable is a reference to the object representing the current module. For convenience, `module.exports` is also accessible via the `exports` module-global. `module` is not actually a global but rather local to each module.

module.children

- `<module[]>`

The module objects required for the first time by this one.

module.exports

- `<Object>`

The `module.exports` object is created by the `Module` system. Sometimes this is not acceptable; many want their module to be an instance of some class. To do this, assign the desired export object to `module.exports`. Assigning the desired object to `exports` will simply rebind the local `exports` variable, which is probably not what is desired.

For example, suppose we were making a module called `a.js`:

```
const EventEmitter = require('events');

module.exports = new EventEmitter();

// Do some work, and after some time emit
// the 'ready' event from the module itself.
setTimeout(() => {
  module.exports.emit('ready');
}, 1000);
```

Then in another file we could do:

```
const a = require('./a');
a.on('ready', () => {
  console.log('module "a" is ready');
});
```

Assignment to `module.exports` must be done immediately. It cannot be done in any callbacks. This does not work:

`x.js`:

```
setTimeout(() => {
  module.exports = { a: 'hello' };
}, 0);
```

`y.js`:

```
const x = require('./x');
console.log(x.a);
```

exports shortcut

The `exports` variable is available within a module's file-level scope, and is assigned the value of `module.exports` before the module is evaluated.

It allows a shortcut, so that `module.exports.f = ...` can be written more succinctly as `exports.f = ...`. However, be aware that like any variable, if a new value is assigned to `exports`, it is no longer bound to `module.exports`:

```
module.exports.hello = true; // Exported from require of module
exports = { hello: false }; // Not exported, only available in the module
```

When the `module.exports` property is being completely replaced by a new object, it is common to also reassign `exports`:

```
module.exports = exports = function Constructor() {
  // ... etc.
};
```

To illustrate the behavior, imagine this hypothetical implementation of `require()`, which is quite similar to what is actually done by `require()`:

```
function require(/* ... */) {
  const module = { exports: {} };
  ((module, exports) => {
    // Module code here. In this example, define a function.
    function someFunc() {}
    exports = someFunc;
    // At this point, exports is no longer a shortcut to module.exports, and
    // this module will still export an empty default object.
    module.exports = someFunc;
    // At this point, the module will now export someFunc, instead of the
    // default object.
  })(module, module.exports);
  return module.exports;
}
```

module.filename

- `<string>`

The fully resolved filename of the module.

module.id

- `<string>`

The identifier for the module. Typically this is the fully resolved filename.

module.isPreloading

- Type: `<boolean>` `true` if the module is running during the Node.js preload phase.

module.loaded

- `<boolean>`

Whether or not the module is done loading, or is in the process of loading.

module.parent

Stability: 0 - Deprecated: Please use `require.main` and `module.children` instead.

- `<module>` | `<null>` | `<undefined>`

The module that first required this one, or `null` if the current module is the entry point of the current process, or `undefined` if the module was loaded by something that is not a CommonJS module (E.G.: REPL or `import`).

module.path

- `<string>`

The directory name of the module. This is usually the same as the `path.dirname()` of the `module.id`.

module.paths

- `<string[]>`

The search paths for the module.

module.require(id)

- `id` `<string>`
- Returns: `<any>` exported module content

The `module.require()` method provides a way to load a module as if `require()` was called from the original module.

In order to do this, it is necessary to get a reference to the `module` object. Since `require()` returns the `module.exports`, and the `module` is typically *only* available within a specific module's code, it must be explicitly exported in order to be used.

The Module object

This section was moved to [Modules: module core module](#).

- `module.builtinModules`
- `module.createRequire(filename)`
- `module.syncBuiltinESMExports()`

Source map v3 support

This section was moved to [Modules: module core module](#).

- `module.findSourceMap(path)`
- `Class: module.SourceMap`
 - `new SourceMap(payload)`
 - `sourceMap.payload`
 - `sourceMap.findEntry(lineNumber, columnNumber)`

Modules: ECMAScript modules

Stability: 2 - Stable

Introduction

ECMAScript modules are [the official standard format](#) to package JavaScript code for reuse. Modules are defined using a variety of `import` and `export` statements.

The following example of an ES module exports a function:

```
// addTwo.mjs
function addTwo(num) {
  return num + 2;
}

export { addTwo };
```

The following example of an ES module imports the function from `addTwo.mjs`:

```
// app.mjs
import { addTwo } from './addTwo.mjs';

// Prints: 6
console.log(addTwo(4));
```

Node.js fully supports ECMAScript modules as they are currently specified and provides interoperability between them and its original module format, [CommonJS](#).

Enabling

Node.js treats JavaScript code as CommonJS modules by default. Authors can tell Node.js to treat JavaScript code as ECMAScript modules via the `.mjs` file extension, the `package.json` `"type"` field, or the `--input-type` flag. See [Modules: Packages](#) for more details.

Packages

This section was moved to [Modules: Packages](#).

import Specifiers

Terminology

The *specifier* of an `import` statement is the string after the `from` keyword, e.g. `'path'` in `import { sep } from 'path'`. Specifiers are also used in `export from` statements, and as the argument to an `import()` expression.

There are three types of specifiers:

- *Relative specifiers* like `'./startup.js'` or `'../config.mjs'`. They refer to a path relative to the location of the importing file. *The file extension is always necessary for these.*
- *Bare specifiers* like `'some-package'` or `'some-package/shuffle'`. They can refer to the main entry point of a package by the package name, or a specific feature module within a package prefixed by the package name as per the examples respectively. *Including the file extension is only necessary for packages without an `"exports"` field.*
- *Absolute specifiers* like `'file:///opt/nodejs/config.js'`. They refer directly and explicitly to a full path.

Bare specifier resolutions are handled by the [Node.js module resolution algorithm](#). All other specifier resolutions are always only resolved with the standard relative [URL](#) resolution semantics.

Like in CommonJS, module files within packages can be accessed by appending a path to the package name unless the package's `package.json` contains an `"exports"` field, in which case files within packages can only be accessed via the paths defined in `"exports"`.

For details on these package resolution rules that apply to bare specifiers in the Node.js module resolution, see the [packages documentation](#).

Mandatory file extensions

A file extension must be provided when using the `import` keyword to resolve relative or absolute specifiers. Directory indexes (e.g. `'./startup/index.js'`) must also be fully specified.

This behavior matches how `import` behaves in browser environments, assuming a typically configured server.

URLs

ES modules are resolved and cached as URLs. This means that files containing special characters such as `#` and `?` need to be escaped.

`file:`, `node:`, and `data:` URL schemes are supported. A specifier like `'https://example.com/app.js'` is not supported natively in Node.js unless using a [custom HTTPS loader](#).

file: URLs

Modules are loaded multiple times if the `import` specifier used to resolve them has a different query or fragment.

```
import './foo.mjs?query=1'; // loads ./foo.mjs with query of "?query=1"
import './foo.mjs?query=2'; // loads ./foo.mjs with query of "?query=2"
```

The volume root may be referenced via `/`, `//` or `file:///`. Given the differences between [URL](#) and path resolution (such as percent encoding details), it is recommended to use [url.pathToFileURL](#) when importing a path.

data: Imports

`data:` URLs are supported for importing with the following MIME types:

- `text/javascript` for ES Modules
- `application/json` for JSON
- `application/wasm` for Wasm

`data:` URLs only resolve *Bare specifiers* for builtin modules and *Absolute specifiers*. Resolving *Relative specifiers* does not work because `data:` is not a [special scheme](#). For example, attempting to load `./foo` from `data:text/javascript,import './foo';` fails to resolve because there is no concept of relative resolution for `data:` URLs. An example of a `data:` URLs being used is:

```
import 'data:text/javascript,console.log("hello!");';
import _ from 'data:application/json,"world!";
```

node: Imports

`node:` URLs are supported as an alternative means to load Node.js builtin modules. This URL scheme allows for builtin modules to be referenced by valid absolute URL strings.

```
import fs from 'node:fs/promises';
```

Builtin modules

`Core modules` provide named exports of their public API. A default export is also provided which is the value of the CommonJS exports. The default export can be used for, among other things, modifying the named exports. Named exports of builtin modules are updated only by calling `module.syncBuiltinESMExports()`.

```
import EventEmitter from 'events';
const e = new EventEmitter();
```

```
import { readFile } from 'fs';
readFile('./foo.txt', (err, source) => {
  if (err) {
    console.error(err);
  } else {
    console.log(source);
  }
});
```

```
import fs, { readFileSync } from 'fs';
import { syncBuiltinESMExports } from 'module';
import { Buffer } from 'buffer';

fs.readFileSync = () => Buffer.from('Hello, ESM');
syncBuiltinESMExports();

fs.readFileSync === readFileSync;
```

import() expressions

`Dynamic import()` is supported in both CommonJS and ES modules. In CommonJS modules it can be used to load ES modules.

import.meta

- `<Object>`

The `import.meta` meta property is an `Object` that contains the following properties.

import.meta.url

- `<string>` The absolute `file:` URL of the module.

This is defined exactly the same as it is in browsers providing the URL of the current module file.

This enables useful patterns such as relative file loading:

```
import { readFileSync } from 'fs';
const buffer = readFileSync(new URL('./data.proto', import.meta.url));
```

import.meta.resolve(specifier[, parent])

Stability: 1 - Experimental

This feature is only available with the `--experimental-import-meta-resolve` command flag enabled.

- `specifier` `<string>` The module specifier to resolve relative to `parent`.
- `parent` `<string>` | `<URL>` The absolute parent module URL to resolve from. If none is specified, the value of `import.meta.url` is used as the default.
- Returns: `<Promise>`

Provides a module-relative resolution function scoped to each module, returning the URL string.

```
const dependencyAsset = await import.meta.resolve('component-lib/asset.css');
```

`import.meta.resolve` also accepts a second argument which is the parent module from which to resolve from:

```
await import.meta.resolve('./dep', import.meta.url);
```

This function is asynchronous because the ES module resolver in Node.js is allowed to be asynchronous.

Interoperability with CommonJS

import statements

An `import` statement can reference an ES module or a CommonJS module. `import` statements are permitted only in ES modules, but dynamic `import()` expressions are supported in CommonJS for loading ES modules.

When importing `CommonJS modules`, the `module.exports` object is provided as the default export. Named exports may be available, provided by static analysis as a convenience for better ecosystem compatibility.

require

The CommonJS module `require` always treats the files it references as CommonJS.

Using `require` to load an ES module is not supported because ES modules have asynchronous execution. Instead, use `import()` to load an ES module from a CommonJS module.

CommonJS Namespaces

CommonJS modules consist of a `module.exports` object which can be of any type.

When importing a CommonJS module, it can be reliably imported using the ES module default import or its corresponding sugar syntax:

```
import { default as cjs } from 'cjs';

// The following import statement is "syntax sugar" (equivalent but sweeter)
// for `{ default as cjsSugar }` in the above import statement:
import cjsSugar from 'cjs';

console.log(cjs);
console.log(cjs === cjsSugar);
// Prints:
//   <module.exports>
//   true
```

The ECMAScript Module Namespace representation of a CommonJS module is always a namespace with a `default` export key pointing to the CommonJS `module.exports` value.

This Module Namespace Exotic Object can be directly observed either when using `import * as m from 'cjs'` or a dynamic import:

```
import * as m from 'cjs';
console.log(m);
console.log(m === await import('cjs'));
// Prints:
//   [Module] { default: <module.exports> }
//   true
```

For better compatibility with existing usage in the JS ecosystem, Node.js in addition attempts to determine the CommonJS named exports of every imported CommonJS module to provide them as separate ES module exports using a static analysis process.

For example, consider a CommonJS module written:

```
// cjs.cjs
exports.name = 'exported';
```

The preceding module supports named imports in ES modules:

```
import { name } from './cjs.cjs';
console.log(name);
// Prints: 'exported'

import cjs from './cjs.cjs';
console.log(cjs);
// Prints: { name: 'exported' }

import * as m from './cjs.cjs';
console.log(m);
// Prints: [Module] { default: { name: 'exported' }, name: 'exported' }
```

As can be seen from the last example of the Module Namespace Exotic Object being logged, the `name` export is copied off of the `module.exports` object and set directly on the ES module namespace when the module is imported.

Live binding updates or new exports added to `module.exports` are not detected for these named exports.

The detection of named exports is based on common syntax patterns but does not always correctly detect named exports. In these cases, using the default import form described above can be a better option.

Named exports detection covers many common export patterns, reexport patterns and build tool and transpiler outputs. See [cjs-module-lexer](#) for the exact semantics implemented.

Differences between ES modules and CommonJS

No `require`, `exports` or `module.exports`

In most cases, the ES module `import` can be used to load CommonJS modules.

If needed, a `require` function can be constructed within an ES module using `module.createRequire()`.

No `__filename` or `__dirname`

These CommonJS variables are not available in ES modules.

`__filename` and `__dirname` use cases can be replicated via `import.meta.url`.

No JSON Module Loading

JSON imports are still experimental and only supported via the `--experimental-json-modules` flag.

Local JSON files can be loaded relative to `import.meta.url` with `fs` directly:

```
import { readFile } from 'fs/promises';
const json = JSON.parse(await readFile(new URL('./dat.json', import.meta.url)));
```

Alternatively `module.createRequire()` can be used.

No Native Module Loading

Native modules are not currently supported with ES module imports.

They can instead be loaded with `module.createRequire()` or `process.dlopen`.

No `require.resolve`

Relative resolution can be handled via `new URL('./local', import.meta.url)`.

For a complete `require.resolve` replacement, there is a flagged experimental `import.meta.resolve` API.

Alternatively `module.createRequire()` can be used.

No `NODE_PATH`

`NODE_PATH` is not part of resolving `import` specifiers. Please use symlinks if this behavior is desired.

No `require.extensions`

`require.extensions` is not used by `import`. The expectation is that loader hooks can provide this workflow in the future.

No `require.cache`

`require.cache` is not used by `import` as the ES module loader has its own separate cache.

JSON modules

Stability: 1 - Experimental

Currently importing JSON modules are only supported in the `commonjs` mode and are loaded using the CJS loader. [WHATWG JSON modules specification](#) are still being standardized, and are experimentally supported by including the additional flag `--experimental-json-modules` when running Node.js.

When the `--experimental-json-modules` flag is included, both the `commonjs` and `module` mode use the new experimental JSON loader. The imported JSON only exposes a `default`. There is no support for named exports. A cache entry is created in the CommonJS cache to avoid duplication. The same object is returned in CommonJS if the JSON module has already been imported from the same path.

Assuming an `index.mjs` with

```
import packageConfig from './package.json';
```

The `--experimental-json-modules` flag is needed for the module to work.

```
node index.mjs # fails
node --experimental-json-modules index.mjs # works
```

Wasm modules

Stability: 1 - Experimental

Importing Web Assembly modules is supported under the `--experimental-wasm-modules` flag, allowing any `.wasm` files to be imported as normal modules while also supporting their module imports.

This integration is in line with the [ES Module Integration Proposal for Web Assembly](#).

For example, an `index.mjs` containing:

```
import * as M from './module.wasm';
console.log(M);
```

executed under:

```
node --experimental-wasm-modules index.mjs
```

would provide the exports interface for the instantiation of `module.wasm`.

Top-level `await`

Stability: 1 - Experimental

The `await` keyword may be used in the top level (outside of async functions) within modules as per the [ECMAScript Top-Level await proposal](#).

Assuming an `a.mjs` with

```
export const five = await Promise.resolve(5);
```

And a `b.mjs` with

```
import { five } from './a.mjs';

console.log(five); // Logs `5`
```

```
node b.mjs # works
```

Loaders

Stability: 1 - Experimental

Note: This API is currently being redesigned and will still change.

To customize the default module resolution, loader hooks can optionally be provided via a `--experimental-loader ./loader-name.mjs` argument to Node.js.

When hooks are used they only apply to ES module loading and not to any CommonJS modules loaded.

Hooks

`resolve(specifier, context, defaultResolve)`

Note: The loaders API is being redesigned. This hook may disappear or its signature may change. Do not rely on the API described below.

- `specifier` `<string>`
- `context` `<Object>`
 - `conditions` `<string[]>`
 - `parentURL` `<string>`
- `defaultResolve` `<Function>`
- Returns: `<Object>`
 - `url` `<string>`

The `resolve` hook returns the resolved file URL for a given module specifier and parent URL. The module specifier is the string in an `import` statement or `import()` expression, and the parent URL is the URL of the module that imported this one, or `undefined` if this is the main entry point for the application.

The `conditions` property on the `context` is an array of conditions for [Conditional exports](#) that apply to this resolution request. They can be used for looking up conditional mappings elsewhere or to modify the list when calling the default resolution logic.

The current `package exports conditions` are always in the `context.conditions` array passed into the hook. To guarantee *default Node.js module specifier resolution behavior* when calling `defaultResolve`, the `context.conditions` array passed to it *must* include *all* elements of the `context.conditions` array originally passed into the `resolve` hook.

```
/**
 * @param {string} specifier
 * @param {{
 *   conditions: !Array<string>,
 *   parentURL: !(string | undefined),
 * }} context
 * @param {Function} defaultResolve
 * @returns {Promise<{ url: string }>}
 */
export async function resolve(specifier, context, defaultResolve) {
  const { parentURL = null } = context;
  if (Math.random() > 0.5) { // Some condition.
    // For some or all specifiers, do some custom logic for resolving.
    // Always return an object of the form {url: <string>}.
    return {
      url: parentURL ?
        new URL(specifier, parentURL).href :
        new URL(specifier).href,
    };
  }
  if (Math.random() < 0.5) { // Another condition.
    // When calling `defaultResolve`, the arguments can be modified. In this
    // case it's adding another value for matching conditional exports.
    return defaultResolve(specifier, {
      ...context,
      conditions: [...context.conditions, 'another-condition'],
    });
  }
  // Defer to Node.js for all other specifiers.
  return defaultResolve(specifier, context, defaultResolve);
}
```

`getFormat(url, context, defaultGetFormat)`

Note: The loaders API is being redesigned. This hook may disappear or its signature may change. Do not rely on the API described below.

- `url` `<string>`
- `context` `<Object>`
- `defaultGetFormat` `<Function>`
- Returns: `<Object>`
 - `format` `<string>`

The `getFormat` hook provides a way to define a custom method of determining how a URL should be interpreted. The `format` returned also affects what the acceptable forms of source values are for a module when parsing. This can be one of the following:

format	Description	Acceptable Types For <code>source</code> Returned by <code>getSource</code> or <code>transformSource</code>
'builtin'	Load a Node.js builtin module	Not applicable
'commonjs'	Load a Node.js CommonJS module	Not applicable
'json'	Load a JSON file	{ <code>string</code> , <code>ArrayBuffer</code> , <code>TypedArray</code> }
'module'	Load an ES module	{ <code>string</code> , <code>ArrayBuffer</code> , <code>TypedArray</code> }
'wasm'	Load a WebAssembly module	{ <code>ArrayBuffer</code> , <code>TypedArray</code> }

Note: These types all correspond to classes defined in ECMAScript.

- The specific `ArrayBuffer` object is a `SharedArrayBuffer`.
- The specific `TypedArray` object is a `Uint8Array`.

Note: If the source value of a text-based format (i.e., `'json'`, `'module'`) is not a string, it is converted to a string using `util.TextDecoder`.

```
/**
 * @param {string} url
 * @param {Object} context (currently empty)
 * @param {Function} defaultGetFormat
 * @returns {Promise<{ format: string }>}
 */
export async function getFormat(url, context, defaultGetFormat) {
  if (Math.random() > 0.5) { // Some condition.
    // For some or all URLs, do some custom logic for determining format.
    // Always return an object of the form {format: <string>}, where the
    // format is one of the strings in the preceding table.
    return {
      format: 'module',
    };
  }
  // Defer to Node.js for all other URLs.
  return defaultGetFormat(url, context, defaultGetFormat);
}
```

getSource(url, context, defaultGetSource)

Note: The loaders API is being redesigned. This hook may disappear or its signature may change. Do not rely on the API described below.

- `url` `<string>`
- `context` `<Object>`
 - `format` `<string>`
- `defaultGetSource` `<Function>`
- Returns: `<Object>`
 - `source` `<string> | <SharedArrayBuffer> | <Uint8Array>`

The `getSource` hook provides a way to define a custom method for retrieving the source code of an ES module specifier. This would allow a loader to potentially avoid reading files from disk.

```
/**
 * @param {string} url
 * @param [{ format: string }] context
 * @param {Function} defaultGetSource
 * @returns {Promise<{ source: !(string | SharedArrayBuffer | Uint8Array) }>}
 */
export async function getSource(url, context, defaultGetSource) {
  const { format } = context;
  if (Math.random() > 0.5) { // Some condition.
    // For some or all URLs, do some custom logic for retrieving the source.
    // Always return an object of the form {source: <string|buffer>}.
    return {
      source: '...',
    };
  }
  // Defer to Node.js for all other URLs.
  return defaultGetSource(url, context, defaultGetSource);
}
```

transformSource(source, context, defaultTransformSource)

Note: The loaders API is being redesigned. This hook may disappear or its signature may change. Do not rely on the API described below.

- `source` `<string> | <SharedArrayBuffer> | <Uint8Array>`
- `context` `<Object>`
 - `format` `<string>`
 - `url` `<string>`
- Returns: `<Object>`
 - `source` `<string> | <SharedArrayBuffer> | <Uint8Array>`

The `transformSource` hook provides a way to modify the source code of a loaded ES module file after the source string has been loaded but before Node.js has done anything with it.

If this hook is used to convert unknown-to-Node.js file types into executable JavaScript, a resolve hook is also necessary in order to register any unknown-to-Node.js file extensions. See the [transpiler loader example](#) below.

```
/**
 * @param {!(string | SharedArrayBuffer | Uint8Array)} source
 * @param [{
 *   format: string,
 *   url: string,
 * }] context
 * @param {Function} defaultTransformSource
 * @returns {Promise<{ source: !(string | SharedArrayBuffer | Uint8Array) }>}
 */
export async function transformSource(source, context, defaultTransformSource) {
  const { url, format } = context;
```

```

if (Math.random() > 0.5) { // Some condition.
  // For some or all URLs, do some custom logic for modifying the source.
  // Always return an object of the form {source: <string|buffer>}.
  return {
    source: '...',
  };
}
// Defer to Node.js for all other sources.
return defaultTransformSource(source, context, defaultTransformSource);
}

```

getGlobalPreloadCode()

Note: The loaders API is being redesigned. This hook may disappear or its signature may change. Do not rely on the API described below.

- Returns: `<string>`

Sometimes it might be necessary to run some code inside of the same global scope that the application runs in. This hook allows the return of a string that is run as sloppy-mode script on startup.

Similar to how CommonJS wrappers work, the code runs in an implicit function scope. The only argument is a `require`-like function that can be used to load builtins like "fs": `getBuiltin(request: string)`.

If the code needs more advanced `require` features, it has to construct its own `require` using `module.createRequire()`.

```

/**
 * @returns {string} Code to run before application startup
 */
export function getGlobalPreloadCode() {
  return `
globalThis.someInjectedProperty = 42;
console.log('I just set some globals!');

const { createRequire } = getBuiltin('module');
const { cwd } = getBuiltin('process');

const require = createRequire(cwd() + '/<preload>');
// [...]
`;
}

```

Examples

The various loader hooks can be used together to accomplish wide-ranging customizations of Node.js' code loading and evaluation behaviors.

HTTPS loader

In current Node.js, specifiers starting with `https://` are unsupported. The loader below registers hooks to enable rudimentary support for such specifiers. While this may seem like a significant improvement to Node.js core functionality, there are substantial downsides to actually using this loader: performance is much slower than loading files from disk, there is no caching, and there is no security.

```

// https-loader.mjs
import { get } from 'https';

export function resolve(specifier, context, defaultResolve) {
  const { parentURL = null } = context;

  // Normally Node.js would error on specifiers starting with 'https://', so
  // this hook intercepts them and converts them into absolute URLs to be
  // passed along to the later hooks below.
  if (specifier.startsWith('https://')) {
    return {
      url: specifier
    };
  } else if (parentURL && parentURL.startsWith('https://')) {
    return {
      url: new URL(specifier, parentURL).href
    };
  }

  // Let Node.js handle all other specifiers.
  return defaultResolve(specifier, context, defaultResolve);
}

export function getFormat(url, context, defaultGetFormat) {
  // This loader assumes all network-provided JavaScript is ES module code.
  if (url.startsWith('https://')) {
    return {
      format: 'module'
    };
  }

  // Let Node.js handle all other URLs.
  return defaultGetFormat(url, context, defaultGetFormat);
}

export function getSource(url, context, defaultGetSource) {
  // For JavaScript to be loaded over the network, we need to fetch and
  // return it.
  if (url.startsWith('https://')) {
    return new Promise((resolve, reject) => {
      get(url, (res) => {
        let data = '';
        res.on('data', (chunk) => data += chunk);
        res.on('end', () => resolve({ source: data }));
      }).on('error', (err) => reject(err));
    });
  }

  // Let Node.js handle all other URLs.

```

```
    return defaultGetSource(url, context, defaultGetSource);
}
```

```
// main.mjs
import { VERSION } from 'https://coffeescript.org/browser-compiler-modern/coffeescript.js';

console.log(VERSION);
```

With the preceding loader, running `node --experimental-loader ./https-loader.mjs ./main.mjs` prints the current version of CoffeeScript per the module at the URL in `main.mjs`.

Transpiler loader

Sources that are in formats Node.js doesn't understand can be converted into JavaScript using the `transformSource` hook. Before that hook gets called, however, other hooks need to tell Node.js not to throw an error on unknown file types; and to tell Node.js how to load this new file type.

This is less performant than transpiling source files before running Node.js; a transpiler loader should only be used for development and testing purposes.

```
// coffeescript-loader.mjs
import { URL, pathToFileURL } from 'url';
import CoffeeScript from 'coffeescript';
import { cwd } from 'process';

const baseUrl = pathToFileURL(`${cwd()}/`).href;

// CoffeeScript files end in .coffee, .litcoffee or .coffee.md.
const extensionsRegex = /\.coffee$|\.litcoffee$|\.coffee\.md$/;

export function resolve(specifier, context, defaultResolve) {
    const { parentURL = baseUrl } = context;

    // Node.js normally errors on unknown file extensions, so return a URL for
    // specifiers ending in the CoffeeScript file extensions.
    if (extensionsRegex.test(specifier)) {
        return {
            url: new URL(specifier, parentURL).href
        };
    }

    // Let Node.js handle all other specifiers.
    return defaultResolve(specifier, context, defaultResolve);
}

export function getFormat(url, context, defaultGetFormat) {
    // Now that we patched resolve to let CoffeeScript URLs through, we need to
    // tell Node.js what format such URLs should be interpreted as. For the
    // purposes of this loader, all CoffeeScript URLs are ES modules.
    if (extensionsRegex.test(url)) {
        return {

```

```

    format: 'module'
  };
}

// Let Node.js handle all other URLs.
return defaultGetFormat(url, context, defaultGetFormat);
}

export function transformSource(source, context, defaultTransformSource) {
  const { url, format } = context;

  if (extensionsRegex.test(url)) {
    return {
      source: CoffeeScript.compile(source, { bare: true })
    };
  }

  // Let Node.js handle all other sources.
  return defaultTransformSource(source, context, defaultTransformSource);
}

```

```

# main.coffee
import { scream } from './scream.coffee'
console.log scream 'hello, world'

import { version } from 'process'
console.log "Brought to you by Node.js version #{version}"

```

```

# scream.coffee
export scream = (str) -> str.toUpperCase()

```

With the preceding loader, running `node --experimental-loader ./coffeescript-loader.mjs main.coffee` causes `main.coffee` to be turned into JavaScript after its source code is loaded from disk but before Node.js executes it; and so on for any `.coffee`, `.litcoffee` or `.coffee.md` files referenced via `import` statements of any loaded file.

Resolution algorithm

Features

The resolver has the following properties:

- FileURL-based resolution as is used by ES modules
- Support for builtin module loading
- Relative and absolute URL resolution
- No default extensions
- No folder mains
- Bare specifier package resolution lookup through `node_modules`

Resolver algorithm

The algorithm to load an ES module specifier is given through the **ESM_RESOLVE** method below. It returns the resolved URL for a module specifier relative to a parentURL.

The algorithm to determine the module format of a resolved URL is provided by **ESM_FORMAT**, which returns the unique module format for any file. The *"module"* format is returned for an ECMAScript Module, while the *"commonjs"* format is used to indicate loading through the legacy CommonJS loader. Additional formats such as *"addon"* can be extended in future updates.

In the following algorithms, all subroutine errors are propagated as errors of these top-level routines unless stated otherwise.

defaultConditions is the conditional environment name array, `["node", "import"]`.

The resolver can throw the following errors:

- *Invalid Module Specifier*: Module specifier is an invalid URL, package name or package subpath specifier.
- *Invalid Package Configuration*: package.json configuration is invalid or contains an invalid configuration.
- *Invalid Package Target*: Package exports or imports define a target module for the package that is an invalid type or string target.
- *Package Path Not Exported*: Package exports do not define or permit a target subpath in the package for the given module.
- *Package Import Not Defined*: Package imports do not define the specifier.
- *Module Not Found*: The package or module requested does not exist.
- *Unsupported Directory Import*: The resolved path corresponds to a directory, which is not a supported target for module imports.

Resolver Algorithm Specification

ESM_RESOLVE(*specifier*, *parentURL*)

1. Let *resolved* be **undefined**.
2. If *specifier* is a valid URL, then
 1. Set *resolved* to the result of parsing and reserializing *specifier* as a URL.
3. Otherwise, if *specifier* starts with *"/"*, *"/."* or *"/.."*, then
 1. Set *resolved* to the URL resolution of *specifier* relative to *parentURL*.
4. Otherwise, if *specifier* starts with *"#"*, then
 1. Set *resolved* to the destructured value of the result of **PACKAGE_IMPORTS_RESOLVE**(*specifier*, *parentURL*, *defaultConditions*).
5. Otherwise,
 1. Note: *specifier* is now a bare specifier.
 2. Set *resolved* the result of **PACKAGE_RESOLVE**(*specifier*, *parentURL*).
6. If *resolved* contains any percent encodings of *"/"* or *"\"* (*"%2f"* and *"%5C"* respectively), then
 1. Throw an *Invalid Module Specifier* error.
7. If the file at *resolved* is a directory, then
 1. Throw an *Unsupported Directory Import* error.
8. If the file at *resolved* does not exist, then
 1. Throw a *Module Not Found* error.
9. Set *resolved* to the real path of *resolved*.
10. Let *format* be the result of **ESM_FORMAT**(*resolved*).
11. Load *resolved* as module format, *format*.
12. Return *resolved*.

PACKAGE_RESOLVE(*packageSpecifier*, *parentURL*)

1. Let *packageName* be **undefined**.
2. If *packageSpecifier* is an empty string, then
 1. Throw an *Invalid Module Specifier* error.
3. If *packageSpecifier* does not start with *"@"*, then
 1. Set *packageName* to the substring of *packageSpecifier* until the first *"/"* separator or the end of the string.
4. Otherwise,

1. If *packageSpecifier* does not contain a "/" separator, then
 1. Throw an *Invalid Module Specifier* error.
2. Set *packageName* to the substring of *packageSpecifier* until the second "/" separator or the end of the string.
5. If *packageName* starts with "." or contains "\" or "%", then
 1. Throw an *Invalid Module Specifier* error.
6. Let *packageSubpath* be "." concatenated with the substring of *packageSpecifier* from the position at the length of *packageName*.
7. Let *selfUrl* be the result of **PACKAGE_SELF_RESOLVE**(*packageName*, *packageSubpath*, *parentURL*).
8. If *selfUrl* is not **undefined**, return *selfUrl*.
9. If *packageSubpath* is "." and *packageName* is a Node.js builtin module, then
 1. Return the string "node:" concatenated with *packageSpecifier*.
10. While *parentURL* is not the file system root,
 1. Let *packageURL* be the URL resolution of "node_modules/" concatenated with *packageSpecifier*, relative to *parentURL*.
 2. Set *parentURL* to the parent folder URL of *parentURL*.
 3. If the folder at *packageURL* does not exist, then
 1. Set *parentURL* to the parent URL path of *parentURL*.
 2. Continue the next loop iteration.
 4. Let *pjson* be the result of **READ_PACKAGE_JSON**(*packageURL*).
 5. If *pjson* is not **null** and *pjson.exports* is not **null** or **undefined**, then
 1. Let *exports* be *pjson.exports*.
 2. Return the *resolved* destructured value of the result of **PACKAGE_EXPORTS_RESOLVE**(*packageURL*, *packageSubpath*, *pjson.exports*, *defaultConditions*).
 6. Otherwise, if *packageSubpath* is equal to ".", then
 1. Return the result applying the legacy **LOAD_AS_DIRECTORY** CommonJS resolver to *packageURL*, throwing a *Module Not Found* error for no resolution.
 7. Otherwise,
 1. Return the URL resolution of *packageSubpath* in *packageURL*.
11. Throw a *Module Not Found* error.

PACKAGE_SELF_RESOLVE(*packageName*, *packageSubpath*, *parentURL*)

1. Let *packageURL* be the result of **READ_PACKAGE_SCOPE**(*parentURL*).
2. If *packageURL* is **null**, then
 1. Return **undefined**.
3. Let *pjson* be the result of **READ_PACKAGE_JSON**(*packageURL*).
4. If *pjson* is **null** or if *pjson.exports* is **null** or **undefined**, then
 1. Return **undefined**.
5. If *pjson.name* is equal to *packageName*, then
 1. Return the *resolved* destructured value of the result of **PACKAGE_EXPORTS_RESOLVE**(*packageURL*, *subpath*, *pjson.exports*, *defaultConditions*).
6. Otherwise, return **undefined**.

PACKAGE_EXPORTS_RESOLVE(*packageURL*, *subpath*, *exports*, *conditions*)

1. If *exports* is an Object with both a key starting with "." and a key not starting with ".", throw an *Invalid Package Configuration* error.
2. If *subpath* is equal to ".", then
 1. Let *mainExport* be **undefined**.
2. If *exports* is a String or Array, or an Object containing no keys starting with ".", then
 1. Set *mainExport* to *exports*.
3. Otherwise if *exports* is an Object containing a "." property, then
 1. Set *mainExport* to *exports*["."].
4. If *mainExport* is not **undefined**, then
 1. Let *resolved* be the result of **PACKAGE_TARGET_RESOLVE**(*packageURL*, *mainExport*, "", **false**, **false**, *conditions*).
 2. If *resolved* is not **null** or **undefined**, then

1. Return *resolved*.
3. Otherwise, if *exports* is an Object and all keys of *exports* start with ".", then
 1. Let *matchKey* be the string "/" concatenated with *subpath*.
 2. Let *resolvedMatch* be result of **PACKAGE_IMPORTS_EXPORTS_RESOLVE**(*matchKey*, *exports*, *packageURL*, **false**, *conditions*).
 3. If *resolvedMatch.resolve* is not **null** or **undefined**, then
 1. Return *resolvedMatch*.
4. Throw a *Package Path Not Exported* error.

PACKAGE_IMPORTS_RESOLVE(*specifier*, *parentURL*, *conditions*)

1. Assert: *specifier* begins with "#".
2. If *specifier* is exactly equal to "#" or starts with "#/", then
 1. Throw an *Invalid Module Specifier* error.
3. Let *packageURL* be the result of **READ_PACKAGE_SCOPE**(*parentURL*).
4. If *packageURL* is not **null**, then
 1. Let *pjson* be the result of **READ_PACKAGE_JSON**(*packageURL*).
 2. If *pjson.imports* is a non-null Object, then
 1. Let *resolvedMatch* be the result of **PACKAGE_IMPORTS_EXPORTS_RESOLVE**(*specifier*, *pjson.imports*, *packageURL*, **true**, *conditions*).
 2. If *resolvedMatch.resolve* is not **null** or **undefined**, then
 1. Return *resolvedMatch*.
5. Throw a *Package Import Not Defined* error.

PACKAGE_IMPORTS_EXPORTS_RESOLVE(*matchKey*, *matchObj*, *packageURL*, *isImports*, *conditions*)

1. If *matchKey* is a key of *matchObj*, and does not end in "***", then
 1. Let *target* be the value of *matchObj*[*matchKey*].
 2. Let *resolved* be the result of **PACKAGE_TARGET_RESOLVE**(*packageURL*, *target*, "", **false**, *isImports*, *conditions*).
 3. Return the object { *resolved*, *exact*: **true** }.
2. Let *expansionKeys* be the list of keys of *matchObj* ending in "/" or "***", sorted by length descending.
3. For each key *expansionKey* in *expansionKeys*, do
 1. If *expansionKey* ends in "***" and *matchKey* starts with but is not equal to the substring of *expansionKey* excluding the last "***" character, then
 1. Let *target* be the value of *matchObj*[*expansionKey*].
 2. Let *subpath* be the substring of *matchKey* starting at the index of the length of *expansionKey* minus one.
 3. Let *resolved* be the result of **PACKAGE_TARGET_RESOLVE**(*packageURL*, *target*, *subpath*, **true**, *isImports*, *conditions*).
 4. Return the object { *resolved*, *exact*: **true** }.
 2. If *matchKey* starts with *expansionKey*, then
 1. Let *target* be the value of *matchObj*[*expansionKey*].
 2. Let *subpath* be the substring of *matchKey* starting at the index of the length of *expansionKey*.
 3. Let *resolved* be the result of **PACKAGE_TARGET_RESOLVE**(*packageURL*, *target*, *subpath*, **false**, *isImports*, *conditions*).
 4. Return the object { *resolved*, *exact*: **false** }.
4. Return the object { *resolved*: **null**, *exact*: **true** }.

PACKAGE_TARGET_RESOLVE(*packageURL*, *target*, *subpath*, *pattern*, *internal*, *conditions*)

1. If *target* is a String, then
 1. If *pattern* is **false**, *subpath* has non-zero length and *target* does not end with "/", throw an *Invalid Module Specifier* error.
 2. If *target* does not start with "./", then
 1. If *internal* is **true** and *target* does not start with "../" or "/" and is not a valid URL, then
 1. If *pattern* is **true**, then
 1. Return **PACKAGE_RESOLVE**(*target* with every instance of "***" replaced by *subpath*, *packageURL* + "/").
 2. Return **PACKAGE_RESOLVE**(*target* + *subpath*, *packageURL* + "/").

2. Otherwise, throw an *Invalid Package Target* error.
3. If *target* split on "/" or "\" contains any ".", ".." or "node_modules" segments after the first segment, throw an *Invalid Package Target* error.
4. Let *resolvedTarget* be the URL resolution of the concatenation of *packageURL* and *target*.
5. Assert: *resolvedTarget* is contained in *packageURL*.
6. If *subpath* split on "/" or "\" contains any ".", ".." or "node_modules" segments, throw an *Invalid Module Specifier* error.
7. If *pattern* is **true**, then
 1. Return the URL resolution of *resolvedTarget* with every instance of "*" replaced with *subpath*.
8. Otherwise,
 1. Return the URL resolution of the concatenation of *subpath* and *resolvedTarget*.
2. Otherwise, if *target* is a non-null Object, then
 1. If *exports* contains any index property keys, as defined in ECMA-262 6.1.7 *Array Index*, throw an *Invalid Package Configuration* error.
 2. For each property *p* of *target*, in object insertion order as,
 1. If *p* equals "default" or *conditions* contains an entry for *p*, then
 1. Let *targetValue* be the value of the *p* property in *target*.
 2. Let *resolved* be the result of **PACKAGE_TARGET_RESOLVE**(*packageURL*, *targetValue*, *subpath*, *pattern*, *internal*, *conditions*).
 3. If *resolved* is equal to **undefined**, continue the loop.
 4. Return *resolved*.
 3. Return **undefined**.
3. Otherwise, if *target* is an Array, then
 1. If *_target.length* is zero, return **null**.
 2. For each item *targetValue* in *target*, do
 1. Let *resolved* be the result of **PACKAGE_TARGET_RESOLVE**(*packageURL*, *targetValue*, *subpath*, *pattern*, *internal*, *conditions*), continuing the loop on any *Invalid Package Target* error.
 2. If *resolved* is **undefined**, continue the loop.
 3. Return *resolved*.
 3. Return or throw the last fallback resolution **null** return or error.
4. Otherwise, if *target* is **null**, return **null**.
5. Otherwise throw an *Invalid Package Target* error.

ESM_FORMAT(*url*)

1. Assert: *url* corresponds to an existing file.
2. Let *pjson* be the result of **READ_PACKAGE_SCOPE**(*url*).
3. If *url* ends in ".mjs", then
 1. Return "module".
4. If *url* ends in ".cjs", then
 1. Return "commonjs".
5. If *pjson?.type* exists and is "module", then
 1. If *url* ends in ".js", then
 1. Return "module".
 2. Throw an *Unsupported File Extension* error.
6. Otherwise,
 1. Throw an *Unsupported File Extension* error.

READ_PACKAGE_SCOPE(*url*)

1. Let *scopeURL* be *url*.
2. While *scopeURL* is not the file system root,
 1. Set *scopeURL* to the parent URL of *scopeURL*.
 2. If *scopeURL* ends in a "node_modules" path segment, return **null**.
3. Let *pjson* be the result of **READ_PACKAGE_JSON**(*scopeURL*).

4. If *pjson* is not **null**, then
 1. Return *pjson*.
3. Return **null**.

READ_PACKAGE_JSON(*packageURL*)

1. Let *pjsonURL* be the resolution of "*package.json*" within *packageURL*.
2. If the file at *pjsonURL* does not exist, then
 1. Return **null**.
3. If the file at *packageURL* does not parse as valid JSON, then
 1. Throw an *Invalid Package Configuration* error.
4. Return the parsed JSON source of the file at *pjsonURL*.

Customizing ESM specifier resolution algorithm

Stability: 1 - Experimental

The current specifier resolution does not support all default behavior of the CommonJS loader. One of the behavior differences is automatic resolution of file extensions and the ability to import directories that have an index file.

The `--experimental-specifier-resolution=[mode]` flag can be used to customize the extension resolution algorithm. The default mode is `explicit`, which requires the full path to a module be provided to the loader. To enable the automatic extension resolution and importing from directories that include an index file use the `node` mode.

```
$ node index.mjs
success!
$ node index # Failure!
Error: Cannot find module
$ node --experimental-specifier-resolution=node index
success!
```

Modules: module API

The Module object

- `<Object>`

Provides general utility methods when interacting with instances of `Module`, the `module` variable often seen in `CommonJS` modules. Accessed via `import 'module'` or `require('module')`.

module.builtinModules

- `<string[]>`

A list of the names of all modules provided by Node.js. Can be used to verify if a module is maintained by a third party or not.

`module` in this context isn't the same object that's provided by the `module wrapper`. To access it, require the `Module` module:

```
// module.mjs
// In an ECMAScript module
import { builtinModules as builtin } from 'module'; // module.cjs
```

```
// In a CommonJS module
const builtin = require('module').builtinModules;
```

module.createRequire(filename)

- `filename` `<string>` | `<URL>` Filename to be used to construct the require function. Must be a file URL object, file URL string, or absolute path string.
- Returns: `<require>` Require function

```
import { createRequire } from 'module';
const require = createRequire(import.meta.url);

// sibling-module.js is a CommonJS module.
const siblingModule = require('./sibling-module');
```

module.syncBuiltinESMExports()

The `module.syncBuiltinESMExports()` method updates all the live bindings for builtin `ES Modules` to match the properties of the `CommonJS` exports. It does not add or remove exported names from the `ES Modules`.

```
const fs = require('fs');
const assert = require('assert');
const { syncBuiltinESMExports } = require('module');

fs.readFile = newAPI;

delete fs.readFileSync;

function newAPI() {
  // ...
}

fs.newAPI = newAPI;

syncBuiltinESMExports();

import('fs').then((esmFS) => {
  // It syncs the existing readFile property with the new value
  assert.strictEqual(esmFS.readFile, newAPI);
  // readFileSync has been deleted from the required fs
  assert.strictEqual('readFileSync' in fs, false);
  // syncBuiltinESMExports() does not remove readFileSync from esmFS
  assert.strictEqual('readFileSync' in esmFS, true);
  // syncBuiltinESMExports() does not add names
  assert.strictEqual(esmFS.newAPI, undefined);
});
```

Source map v3 support

Helpers for interacting with the source map cache. This cache is populated when source map parsing is enabled and **source map include directives** are found in a modules' footer.

To enable source map parsing, Node.js must be run with the flag `--enable-source-maps`, or with code coverage enabled by setting `NODE_V8_COVERAGE=dir`.

```
// module.mjs
// In an ECMAScript module
import { findSourceMap, SourceMap } from 'module';// module.cjs
// In a CommonJS module
const { findSourceMap, SourceMap } = require('module');
```

module.findSourceMap(path)

- `path` `<string>`
- Returns: `<module.SourceMap>`

`path` is the resolved path for the file for which a corresponding source map should be fetched.

Class: module.SourceMap

new SourceMap(payload)

- `payload` `<Object>`

Creates a new `sourceMap` instance.

`payload` is an object with keys matching the **Source map v3 format** :

- `file`: `<string>`
- `version`: `<number>`
- `sources`: `<string[]>`
- `sourcesContent`: `<string[]>`
- `names`: `<string[]>`
- `mappings`: `<string>`
- `sourceRoot`: `<string>`

sourceMap.payload

- Returns: `<Object>`

Getter for the payload used to construct the `SourceMap` instance.

sourceMap.findEntry(lineNumber, columnNumber)

- `lineNumber` `<number>`
- `columnNumber` `<number>`
- Returns: `<Object>`

Given a line number and column number in the generated source file, returns an object representing the position in the original file. The object returned consists of the following keys:

- `generatedLine`: `<number>`
- `generatedColumn`: `<number>`
- `originalSource`: `<string>`
- `originalLine`: `<number>`
- `originalColumn`: `<number>`
- `name`: `<string>`

Modules: Packages

Introduction

A package is a folder tree described by a `package.json` file. The package consists of the folder containing the `package.json` file and all subfolders until the next folder containing another `package.json` file, or a folder named `node_modules`.

This page provides guidance for package authors writing `package.json` files along with a reference for the `package.json` fields defined by Node.js.

Determining module system

Node.js will treat the following as `ES modules` when passed to `node` as the initial input, or when referenced by `import` statements within ES module code:

- Files ending in `.mjs`.
- Files ending in `.js` when the nearest parent `package.json` file contains a top-level `"type"` field with a value of `"module"`.
- Strings passed in as an argument to `--eval`, or piped to `node` via `STDIN`, with the flag `--input-type=module`.

Node.js will treat as `CommonJS` all other forms of input, such as `.js` files where the nearest parent `package.json` file contains no top-level `"type"` field, or string input without the flag `--input-type`. This behavior is to preserve backward compatibility. However, now that Node.js supports both CommonJS and ES modules, it is best to be explicit whenever possible. Node.js will treat the following as CommonJS when passed to `node` as the initial input, or when referenced by `import` statements within ES module code:

- Files ending in `.cjs`.
- Files ending in `.js` when the nearest parent `package.json` file contains a top-level field `"type"` with a value of `"commonjs"`.
- Strings passed in as an argument to `--eval` or `--print`, or piped to `node` via `STDIN`, with the flag `--input-type=commonjs`.

Package authors should include the `"type"` field, even in packages where all sources are CommonJS. Being explicit about the `type` of the package will future-proof the package in case the default type of Node.js ever changes, and it will also make things easier for build tools and loaders to determine how the files in the package should be interpreted.

`package.json` and file extensions

Within a package, the `package.json` `"type"` field defines how Node.js should interpret `.js` files. If a `package.json` file does not have a `"type"` field, `.js` files are treated as `CommonJS`.

A `package.json` `"type"` value of `"module"` tells Node.js to interpret `.js` files within that package as using `ES module` syntax.

The `"type"` field applies not only to initial entry points (`node my-app.js`) but also to files referenced by `import` statements and `import()` expressions.

```
// my-app.js, treated as an ES module because there is a package.json
// file in the same folder with "type": "module".

import './startup/init.js';
// Loaded as ES module since ./startup contains no package.json file,
// and therefore inherits the "type" value from one level up.

import 'commonjs-package';
// Loaded as CommonJS since ./node_modules/commonjs-package/package.json
// lacks a "type" field or contains "type": "commonjs".

import './node_modules/commonjs-package/index.js';
// Loaded as CommonJS since ./node_modules/commonjs-package/package.json
// lacks a "type" field or contains "type": "commonjs".
```

Files ending with `.mjs` are always loaded as **ES modules** regardless of the nearest parent `package.json`.

Files ending with `.cjs` are always loaded as **CommonJS** regardless of the nearest parent `package.json`.

```
import './legacy-file.cjs';
// Loaded as CommonJS since .cjs is always loaded as CommonJS.

import 'commonjs-package/src/index.mjs';
// Loaded as ES module since .mjs is always loaded as ES module.
```

The `.mjs` and `.cjs` extensions can be used to mix types within the same package:

- Within a `"type": "module"` package, Node.js can be instructed to interpret a particular file as **CommonJS** by naming it with a `.cjs` extension (since both `.js` and `.mjs` files are treated as ES modules within a `"module"` package).
- Within a `"type": "commonjs"` package, Node.js can be instructed to interpret a particular file as an **ES module** by naming it with an `.mjs` extension (since both `.js` and `.cjs` files are treated as CommonJS within a `"commonjs"` package).

--input-type flag

Strings passed in as an argument to `--eval` (or `-e`), or piped to `node` via `STDIN`, are treated as **ES modules** when the `--input-type=module` flag is set.

```
node --input-type=module --eval "import { sep } from 'path'; console.log(sep);"

echo "import { sep } from 'path'; console.log(sep);" | node --input-type=module
```

For completeness there is also `--input-type=commonjs`, for explicitly running string input as CommonJS. This is the default behavior if `--input-type` is unspecified.

Package entry points

In a package's `package.json` file, two fields can define entry points for a package: `"main"` and `"exports"`. The `"main"` field is supported in all versions of Node.js, but its capabilities are limited: it only defines the main entry point of the package.

The `"exports"` field provides an alternative to `"main"` where the package main entry point can be defined while also encapsulating the package, **preventing any other entry points besides those defined in `"exports"`**. This encapsulation allows module authors to define a public interface for their package.

If both `"exports"` and `"main"` are defined, the `"exports"` field takes precedence over `"main"`. `"exports"` are not specific to ES modules or CommonJS; `"main"` is overridden by `"exports"` if it exists. As such `"main"` cannot be used as a fallback for CommonJS but it can be used as a fallback for legacy versions of Node.js that do not support the `"exports"` field.

Conditional exports can be used within `"exports"` to define different package entry points per environment, including whether the package is referenced via `require` or via `import`. For more information about supporting both CommonJS and ES Modules in a single package please consult [the dual CommonJS/ES module packages section](#).

Warning: Introducing the `"exports"` field prevents consumers of a package from using any entry points that are not defined, including the `package.json` (e.g. `require('your-package/package.json')`). **This will likely be a breaking change.**

To make the introduction of `"exports"` non-breaking, ensure that every previously supported entry point is exported. It is best to explicitly specify entry points so that the package's public API is well-defined. For example, a project that previous exported `main`, `lib`, `feature`, and the `package.json` could use the following `package.exports`:

```
{
  "name": "my-mod",
  "exports": {
    ".": "./lib/index.js",
    "./lib": "./lib/index.js",
    "./lib/index": "./lib/index.js",
    "./lib/index.js": "./lib/index.js",
    "./feature": "./feature/index.js",
    "./feature/index.js": "./feature/index.js",
    "./package.json": "./package.json"
  }
}
```

Alternatively a project could choose to export entire folders:

```
{
  "name": "my-mod",
  "exports": {
    ".": "./lib/index.js",
    "./lib": "./lib/index.js",
    "./lib/*": "./lib/*.js",
    "./feature": "./feature/index.js",
    "./feature/*": "./feature/*.js",
    "./package.json": "./package.json"
  }
}
```

As a last resort, package encapsulation can be disabled entirely by creating an export for the root of the package `"./*": "./*"`. This exposes every file in the package at the cost of disabling the encapsulation and potential tooling benefits this provides. As the ES Module loader in Node.js enforces the use of [the full specifier path](#), exporting the root rather than being explicit about entry is less expressive than either of the

prior examples. Not only is encapsulation lost but module consumers are unable to `import feature from 'my-mod/feature'` as they need to provide the full path `import feature from 'my-mod/feature/index.js'`.

Main entry point export

To set the main entry point for a package, it is advisable to define both `"exports"` and `"main"` in the package's `package.json` file:

```
{
  "main": "./main.js",
  "exports": "./main.js"
}
```

When the `"exports"` field is defined, all subpaths of the package are encapsulated and no longer available to importers. For example, `require('pkg/subpath.js')` throws an `ERR_PACKAGE_PATH_NOT_EXPORTED` error.

This encapsulation of exports provides more reliable guarantees about package interfaces for tools and when handling semver upgrades for a package. It is not a strong encapsulation since a direct require of any absolute subpath of the package such as `require('/path/to/node_modules/pkg/subpath.js')` will still load `subpath.js`.

Subpath exports

When using the `"exports"` field, custom subpaths can be defined along with the main entry point by treating the main entry point as the `"."` subpath:

```
{
  "main": "./main.js",
  "exports": {
    ".": "./main.js",
    "./submodule": "./src/submodule.js"
  }
}
```

Now only the defined subpath in `"exports"` can be imported by a consumer:

```
import submodule from 'es-module-package/submodule';
// Loads ./node_modules/es-module-package/src/submodule.js
```

While other subpaths will error:

```
import submodule from 'es-module-package/private-module.js';
// Throws ERR_PACKAGE_PATH_NOT_EXPORTED
```

Subpath imports

In addition to the `"exports"` field, it is possible to define internal package import maps that only apply to import specifiers from within the package itself.

Entries in the imports field must always start with `#` to ensure they are disambiguated from package specifiers.

For example, the imports field can be used to gain the benefits of conditional exports for internal modules:

```
// package.json
{
  "imports": {
    "#dep": {
      "node": "dep-node-native",
      "default": "../dep-polyfill.js"
    }
  },
  "dependencies": {
    "dep-node-native": "^1.0.0"
  }
}
```

where `import '#dep'` does not get the resolution of the external package `dep-node-native` (including its exports in turn), and instead gets the local file `../dep-polyfill.js` relative to the package in other environments.

Unlike the `"exports"` field, the `"imports"` field permits mapping to external packages.

The resolution rules for the imports field are otherwise analogous to the exports field.

Subpath patterns

For packages with a small number of exports or imports, we recommend explicitly listing each exports subpath entry. But for packages that have large numbers of subpaths, this might cause `package.json` bloat and maintenance issues.

For these use cases, subpath export patterns can be used instead:

```
// ./node_modules/es-module-package/package.json
{
  "exports": {
    "../features/*": "../src/features/*.js"
  },
  "imports": {
    "#internal/*": "../src/internal/*.js"
  }
}
```

*** maps expose nested subpaths as it is a string replacement syntax only.**

The left hand matching pattern must always end in `*`. All instances of `*` on the right hand side will then be replaced with this value, including if it contains any `/` separators.

```
import featureX from 'es-module-package/features/x';
// Loads ./node_modules/es-module-package/src/features/x.js

import featureY from 'es-module-package/features/y/y';
// Loads ./node_modules/es-module-package/src/features/y/y.js

import internalZ from '#internal/z';
// Loads ./node_modules/es-module-package/src/internal/z.js
```

This is a direct static replacement without any special handling for file extensions. In the previous example, `pkg/features/x.json` would be resolved to `./src/features/x.json.js` in the mapping.

The property of exports being statically enumerable is maintained with exports patterns since the individual exports for a package can be determined by treating the right hand side target pattern as a `**` glob against the list of files within the package. Because `node_modules` paths are forbidden in exports targets, this expansion is dependent on only the files of the package itself.

To exclude private subfolders from patterns, `null` targets can be used:

```
// ./node_modules/es-module-package/package.json
{
  "exports": {
    "./features/*": "./src/features/*.js",
    "./features/private-internal/*": null
  }
}
```

```
import featureInternal from 'es-module-package/features/private-internal/m';
// Throws: ERR_PACKAGE_PATH_NOT_EXPORTED

import featureX from 'es-module-package/features/x';
// Loads ./node_modules/es-module-package/src/features/x.js
```

Subpath folder mappings

Stability: 0 - Deprecated: Use subpath patterns instead.

Before subpath patterns were supported, a trailing `"/"` suffix was used to support folder mappings:

```
{
  "exports": {
    "./features/": "./features/"
  }
}
```

This feature will be removed in a future release.

Instead, use direct **subpath patterns** :

```
{
  "exports": {
    "./features/*": "./features/*.js"
  }
}
```

The benefit of patterns over folder exports is that packages can always be imported by consumers without subpath file extensions being necessary.

Exports sugar

If the `"."` export is the only export, the `"exports"` field provides sugar for this case being the direct `"exports"` field value.

If the `"."` export has a fallback array or string value, then the `"exports"` field can be set to this value directly.

```
{
  "exports": {
    ".": "./main.js"
  }
}
```

can be written:

```
{
  "exports": "./main.js"
}
```

Conditional exports

Conditional exports provide a way to map to different paths depending on certain conditions. They are supported for both CommonJS and ES module imports.

For example, a package that wants to provide different ES module exports for `require()` and `import` can be written:

```
// package.json
{
  "main": "./main-require.cjs",
  "exports": {
    "import": "./main-module.js",
    "require": "./main-require.cjs"
  },
  "type": "module"
}
```

Node.js implements the following conditions:

- `"import"` - matches when the package is loaded via `import` or `import()`, or via any top-level import or resolve operation by the ECMAScript module loader. Applies regardless of the module format of the target file. *Always mutually exclusive with `"require"`.*
- `"require"` - matches when the package is loaded via `require()`. The referenced file should be loadable with `require()` although the condition matches regardless of the module format of the target file. Expected formats include CommonJS, JSON, and native addons but not ES modules as `require()` doesn't support them. *Always mutually exclusive with `"import"`.*
- `"node"` - matches for any Node.js environment. Can be a CommonJS or ES module file. *This condition should always come after `"import"` or `"require"`.*
- `"default"` - the generic fallback that always matches. Can be a CommonJS or ES module file. *This condition should always come last.*

Within the `"exports"` object, key order is significant. During condition matching, earlier entries have higher priority and take precedence over later entries. *The general rule is that conditions should be from most specific to least specific in object order.*

Using the `"import"` and `"require"` conditions can lead to some hazards, which are further explained in [the dual CommonJS/ES module packages section](#).

Conditional exports can also be extended to exports subpaths, for example:

```
{
  "main": "./main.js",
  "exports": {
    ".": "./main.js",
    "./feature": {
      "node": "./feature-node.js",
      "default": "./feature.js"
    }
  }
}
```

Defines a package where `require('pkg/feature')` and `import 'pkg/feature'` could provide different implementations between Node.js and other JS environments.

When using environment branches, always include a `"default"` condition where possible. Providing a `"default"` condition ensures that any unknown JS environments are able to use this universal implementation, which helps avoid these JS environments from having to pretend to be existing environments in order to support packages with conditional exports. For this reason, using `"node"` and `"default"` condition branches is usually preferable to using `"node"` and `"browser"` condition branches.

Nested conditions

In addition to direct mappings, Node.js also supports nested condition objects.

For example, to define a package that only has dual mode entry points for use in Node.js but not the browser:

```
{
  "main": "./main.js",
  "exports": {
    "node": {
      "import": "./feature-node.mjs",
      "require": "./feature-node.cjs"
    },
    "default": "./feature.mjs",
  }
}
```

Conditions continue to be matched in order as with flat conditions. If a nested conditional does not have any mapping it will continue checking the remaining conditions of the parent condition. In this way nested conditions behave analogously to nested JavaScript `if` statements.

Resolving user conditions

When running Node.js, custom user conditions can be added with the `--conditions` flag:

```
node --conditions=development main.js
```

which would then resolve the `"development"` condition in package imports and exports, while resolving the existing `"node"`, `"default"`, `"import"`, and `"require"` conditions as appropriate.

Any number of custom conditions can be set with repeat flags.

Conditions Definitions

The `"import"`, `"require"`, `"node"` and `"default"` conditions are defined and implemented in Node.js core, [as specified above](#).

Other condition strings are unknown to Node.js and thus ignored by default. Runtimes or tools other than Node.js can use them at their discretion.

These user conditions can be enabled in Node.js via the `--conditions` flag.

The following condition definitions are currently endorsed by Node.js:

- `"browser"` - any environment which implements a standard subset of global browser APIs available from JavaScript in web browsers, including the DOM APIs.
- `"development"` - can be used to define a development-only environment entry point. *Must always be mutually exclusive with "production"*.
- `"production"` - can be used to define a production environment entry point. *Must always be mutually exclusive with "development"*.

The above user conditions can be enabled in Node.js via the `--conditions` flag.

Platform specific conditions such as `"deno"`, `"electron"`, or `"react-native"` may be used, but while there remain no implementation or integration intent from these platforms, the above are not explicitly endorsed by Node.js.

New conditions definitions may be added to this list by creating a pull request to the [Node.js documentation for this section](#). The requirements for listing a new condition definition here are that:

- The definition should be clear and unambiguous for all implementers.
- The use case for why the condition is needed should be clearly justified.
- There should exist sufficient existing implementation usage.
- The condition name should not conflict with another condition definition or condition in wide usage.
- The listing of the condition definition should provide a coordination benefit to the ecosystem that wouldn't otherwise be possible. For example, this would not necessarily be the case for company-specific or application-specific conditions.

The above definitions may be moved to a dedicated conditions registry in due course.

Self-referencing a package using its name

Within a package, the values defined in the package's `package.json` `"exports"` field can be referenced via the package's name. For example, assuming the `package.json` is:

```
// package.json
{
  "name": "a-package",
  "exports": {
    ".": "./main.mjs",
    "./foo": "./foo.js"
  }
}
```

Then any module *in that package* can reference an export in the package itself:

```
// ./a-module.mjs
import { something } from 'a-package'; // Imports "something" from ./main.mjs.
```

Self-referencing is available only if `package.json` has `"exports"`, and will allow importing only what that `"exports"` (in the `package.json`) allows. So the code below, given the previous package, will generate a runtime error:

```
// ./another-module.mjs

// Imports "another" from ./m.mjs. Fails because
// the "package.json" "exports" field
// does not provide an export named "./m.mjs".
import { another } from 'a-package/m.mjs';
```

Self-referencing is also available when using `require`, both in an ES module, and in a CommonJS one. For example, this code will also work:

```
// ./a-module.js
const { something } = require('a-package/foo'); // Loads from ./foo.js.
```

Dual CommonJS/ES module packages

Prior to the introduction of support for ES modules in Node.js, it was a common pattern for package authors to include both CommonJS and ES module JavaScript sources in their package, with `package.json` `"main"` specifying the CommonJS entry point and `package.json` `"module"` specifying the ES module entry point. This enabled Node.js to run the CommonJS entry point while build tools such as bundlers used the ES module entry point, since Node.js ignored (and still ignores) the top-level `"module"` field.

Node.js can now run ES module entry points, and a package can contain both CommonJS and ES module entry points (either via separate specifiers such as `'pkg'` and `'pkg/es-module'`, or both at the same specifier via [Conditional exports](#)). Unlike in the scenario where `"module"` is only used by bundlers, or ES module files are transpiled into CommonJS on the fly before evaluation by Node.js, the files referenced by the ES module entry point are evaluated as ES modules.

Dual package hazard

When an application is using a package that provides both CommonJS and ES module sources, there is a risk of certain bugs if both versions of the package get loaded. This potential comes from the fact that the `pkgInstance` created by `const pkgInstance = require('pkg')` is not the same as the `pkgInstance` created by `import pkgInstance from 'pkg'` (or an alternative main path like `'pkg/module'`). This is the “dual package hazard,” where two versions of the same package can be loaded within the same runtime environment. While it is unlikely that an application or package would intentionally load both versions directly, it is common for an application to load one version while a dependency of the application loads the other version. This hazard can happen because Node.js supports intermixing CommonJS and ES modules, and can lead to unexpected behavior.

If the package main export is a constructor, an `instanceof` comparison of instances created by the two versions returns `false`, and if the export is an object, properties added to one (like `pkgInstance.foo = 3`) are not present on the other. This differs from how `import` and `require` statements work in all-CommonJS or all-ES module environments, respectively, and therefore is surprising to users. It also differs from the behavior users are familiar with when using transpilation via tools like [Babel](#) or [esm](#).

Writing dual packages while avoiding or minimizing hazards

First, the hazard described in the previous section occurs when a package contains both CommonJS and ES module sources and both sources are provided for use in Node.js, either via separate main entry points or exported paths. A package might instead be written where any version of Node.js receives only CommonJS sources, and any separate ES module sources the package might contain are intended only for other environments such as browsers. Such a package would be usable by any version of Node.js, since `import` can refer to CommonJS files; but it would not provide any of the advantages of using ES module syntax.

A package might also switch from CommonJS to ES module syntax in a [breaking change](#) version bump. This has the disadvantage that the newest version of the package would only be usable in ES module-supporting versions of Node.js.

Every pattern has tradeoffs, but there are two broad approaches that satisfy the following conditions:

1. The package is usable via both `require` and `import`.
2. The package is usable in both current Node.js and older versions of Node.js that lack support for ES modules.
3. The package main entry point, e.g. `'pkg'` can be used by both `require` to resolve to a CommonJS file and by `import` to resolve to an ES module file. (And likewise for exported paths, e.g. `'pkg/feature'`.)
4. The package provides named exports, e.g. `import { name } from 'pkg'` rather than `import pkg from 'pkg'; pkg.name`.
5. The package is potentially usable in other ES module environments such as browsers.
6. The hazards described in the previous section are avoided or minimized.

Approach #1: Use an ES module wrapper

Write the package in CommonJS or transpile ES module sources into CommonJS, and create an ES module wrapper file that defines the named exports. Using [Conditional exports](#), the ES module wrapper is used for `import` and the CommonJS entry point for `require`.

```
// ./node_modules/pkg/package.json
{
  "type": "module",
  "main": "./index.cjs",
  "exports": {
    "import": "./wrapper.mjs",
    "require": "./index.cjs"
  }
}
```

The preceding example uses explicit extensions `.mjs` and `.cjs`. If your files use the `.js` extension, `"type": "module"` will cause such files to be treated as ES modules, just as `"type": "commonjs"` would cause them to be treated as CommonJS. See [Enabling](#).

```
// ./node_modules/pkg/index.cjs
exports.name = 'value';
```

```
// ./node_modules/pkg/wrapper.mjs
import cjsModule from './index.cjs';
export const name = cjsModule.name;
```

In this example, the `name` from `import { name } from 'pkg'` is the same singleton as the `name` from `const { name } = require('pkg')`. Therefore `===` returns `true` when comparing the two `name`s and the divergent specifier hazard is avoided.

If the module is not simply a list of named exports, but rather contains a unique function or object export like `module.exports = function () { ... }`, or if support in the wrapper for the `import pkg from 'pkg'` pattern is desired, then the wrapper would instead be written to export the default optionally along with any named exports as well:

```
import cjsModule from './index.cjs';
export const name = cjsModule.name;
export default cjsModule;
```

This approach is appropriate for any of the following use cases:

- The package is currently written in CommonJS and the author would prefer not to refactor it into ES module syntax, but wishes to provide named exports for ES module consumers.

- The package has other packages that depend on it, and the end user might install both this package and those other packages. For example a `utilities` package is used directly in an application, and a `utilities-plus` package adds a few more functions to `utilities`. Because the wrapper exports underlying CommonJS files, it doesn't matter if `utilities-plus` is written in CommonJS or ES module syntax; it will work either way.
- The package stores internal state, and the package author would prefer not to refactor the package to isolate its state management. See the next section.

A variant of this approach not requiring conditional exports for consumers could be to add an export, e.g. `"/module"`, to point to an all-ES module-syntax version of the package. This could be used via `import 'pkg/module'` by users who are certain that the CommonJS version will not be loaded anywhere in the application, such as by dependencies; or if the CommonJS version can be loaded but doesn't affect the ES module version (for example, because the package is stateless):

```
// ./node_modules/pkg/package.json
{
  "type": "module",
  "main": "./index.cjs",
  "exports": {
    ".": "./index.cjs",
    "/module": "./wrapper.mjs"
  }
}
```

Approach #2: Isolate state

A `package.json` file can define the separate CommonJS and ES module entry points directly:

```
// ./node_modules/pkg/package.json
{
  "type": "module",
  "main": "./index.cjs",
  "exports": {
    "import": "./index.mjs",
    "require": "./index.cjs"
  }
}
```

This can be done if both the CommonJS and ES module versions of the package are equivalent, for example because one is the transpiled output of the other; and the package's management of state is carefully isolated (or the package is stateless).

The reason that state is an issue is because both the CommonJS and ES module versions of the package might get used within an application; for example, the user's application code could `import` the ES module version while a dependency `require`s the CommonJS version. If that were to occur, two copies of the package would be loaded in memory and therefore two separate states would be present. This would likely cause hard-to-troubleshoot bugs.

Aside from writing a stateless package (if JavaScript's `Math` were a package, for example, it would be stateless as all of its methods are static), there are some ways to isolate state so that it's shared between the potentially loaded CommonJS and ES module instances of the package:

1. If possible, contain all state within an instantiated object. JavaScript's `Date`, for example, needs to be instantiated to contain state; if it were a package, it would be used like this:

```
import Date from 'date';
const someDate = new Date();
```

```
// someDate contains state; Date does not
```

The `new` keyword isn't required; a package's function can return a new object, or modify a passed-in object, to keep the state external to the package.

2. Isolate the state in one or more CommonJS files that are shared between the CommonJS and ES module versions of the package. For example, if the CommonJS and ES module entry points are `index.cjs` and `index.mjs`, respectively:

```
// ./node_modules/pkg/index.cjs
const state = require('./state.cjs');
module.exports.state = state;
```

```
// ./node_modules/pkg/index.mjs
import state from './state.cjs';
export {
  state
};
```

Even if `pkg` is used via both `require` and `import` in an application (for example, via `import` in application code and via `require` by a dependency) each reference of `pkg` will contain the same state; and modifying that state from either module system will apply to both.

Any plugins that attach to the package's singleton would need to separately attach to both the CommonJS and ES module singletons.

This approach is appropriate for any of the following use cases:

- The package is currently written in ES module syntax and the package author wants that version to be used wherever such syntax is supported.
- The package is stateless or its state can be isolated without too much difficulty.
- The package is unlikely to have other public packages that depend on it, or if it does, the package is stateless or has state that need not be shared between dependencies or with the overall application.

Even with isolated state, there is still the cost of possible extra code execution between the CommonJS and ES module versions of a package.

As with the previous approach, a variant of this approach not requiring conditional exports for consumers could be to add an export, e.g. `"./module"`, to point to an all-ES module-syntax version of the package:

```
// ./node_modules/pkg/package.json
{
  "type": "module",
  "main": "./index.cjs",
  "exports": {
    ".": "./index.cjs",
    "./module": "./index.mjs"
  }
}
```

Node.js `package.json` field definitions

This section describes the fields used by the Node.js runtime. Other tools (such as `npm`) use additional fields which are ignored by Node.js and not documented here.

The following fields in `package.json` files are used in Node.js:

- `"name"` - Relevant when using named imports within a package. Also used by package managers as the name of the package.
- `"main"` - The default module when loading the package, if `exports` is not specified, and in versions of Node.js prior to the introduction of `exports`.
- `"type"` - The package type determining whether to load `.js` files as CommonJS or ES modules.
- `"exports"` - Package exports and conditional exports. When present, limits which submodules can be loaded from within the package.
- `"imports"` - Package imports, for use by modules within the package itself.

"name"

- Type: `<string>`

```
{
  "name": "package-name"
}
```

The `"name"` field defines your package's name. Publishing to the *npm* registry requires a name that satisfies [certain requirements](#).

The `"name"` field can be used in addition to the `"exports"` field to [self-reference](#) a package using its name.

"main"

- Type: `<string>`

```
{
  "main": "./main.js"
}
```

The `"main"` field defines the script that is used when the [package directory is loaded via `require\(\)`](#). Its value is a path.

```
require('./path/to/directory'); // This resolves to ./path/to/directory/main.js.
```

When a package has an `"exports"` field, this will take precedence over the `"main"` field when importing the package by name.

"type"

- Type: `<string>`

The `"type"` field defines the module format that Node.js uses for all `.js` files that have that `package.json` file as their nearest parent.

Files ending with `.js` are loaded as ES modules when the nearest parent `package.json` file contains a top-level field `"type"` with a value of `"module"`.

The nearest parent `package.json` is defined as the first `package.json` found when searching in the current folder, that folder's parent, and so on up until a `node_modules` folder or the volume root is reached.

```
// package.json
{
  "type": "module"
}
```

```
# In same folder as preceding package.json
node my-app.js # Runs as ES module
```

If the nearest parent `package.json` lacks a `"type"` field, or contains `"type": "commonjs"`, `.js` files are treated as **CommonJS**. If the volume root is reached and no `package.json` is found, `.js` files are treated as **CommonJS**.

`import` statements of `.js` files are treated as ES modules if the nearest parent `package.json` contains `"type": "module"`.

```
// my-app.js, part of the same example as above
import './startup.js'; // Loaded as ES module because of package.json
```

Regardless of the value of the `"type"` field, `.mjs` files are always treated as ES modules and `.cjs` files are always treated as CommonJS.

"exports"

- Type: `<Object> | <string> | <string[]>`

```
{
  "exports": "./index.js"
}
```

The `"exports"` field allows defining the **entry points** of a package when imported by name loaded either via a `node_modules` lookup or a **self-reference** to its own name. It is supported in Node.js 12+ as an alternative to the `"main"` that can support defining **subpath exports** and **conditional exports** while encapsulating internal unexported modules.

Conditional Exports can also be used within `"exports"` to define different package entry points per environment, including whether the package is referenced via `require` or via `import`.

All paths defined in the `"exports"` must be relative file URLs starting with `./`.

"imports"

- Type: `<Object>`

```
// package.json
{
  "imports": {
    "#dep": {
      "node": "dep-node-native",
      "default": "./dep-polyfill.js"
    }
  },
  "dependencies": {
    "dep-node-native": "^1.0.0"
  }
}
```

Entries in the imports field must be strings starting with `#`.

Import maps permit mapping to external packages.

This field defines `subpath imports` for the current package.

Net

Stability: 2 - Stable

Source Code: [lib/net.js](#)

The `net` module provides an asynchronous network API for creating stream-based TCP or `IPC` servers (`net.createServer()`) and clients (`net.createConnection()`).

It can be accessed using:

```
const net = require('net');
```

IPC support

The `net` module supports IPC with named pipes on Windows, and Unix domain sockets on other operating systems.

Identifying paths for IPC connections

`net.connect()`, `net.createConnection()`, `server.listen()` and `socket.connect()` take a `path` parameter to identify IPC endpoints.

On Unix, the local domain is also known as the Unix domain. The path is a filesystem pathname. It gets truncated to an OS-dependent length of `sizeof(sockaddr_un.sun_path) - 1`. Typical values are 107 bytes on Linux and 103 bytes on macOS. If a Node.js API abstraction creates the Unix domain socket, it will unlink the Unix domain socket as well. For example, `net.createServer()` may create a Unix domain socket and `server.close()` will unlink it. But if a user creates the Unix domain socket outside of these abstractions, the user will need to remove it. The same applies when a Node.js API creates a Unix domain socket but the program then crashes. In short, a Unix domain socket will be visible in the filesystem and will persist until unlinked.

On Windows, the local domain is implemented using a named pipe. The path *must* refer to an entry in `\\?\pipe\` or `\\.pipe\`. Any characters are permitted, but the latter may do some processing of pipe names, such as resolving `...` sequences. Despite how it might look, the pipe namespace is flat. Pipes will *not persist*. They are removed when the last reference to them is closed. Unlike Unix domain sockets, Windows will close and remove the pipe when the owning process exits.

JavaScript string escaping requires paths to be specified with extra backslash escaping such as:

```
net.createServer().listen(
  path.join('\\\\\\?\\pipe', process.cwd(), 'myctl'));
```

Class: `net.BlockList`

The `BlockList` object can be used with some network APIs to specify rules for disabling inbound or outbound access to specific IP addresses, IP ranges, or IP subnets.

`blockList.addAddress(address[, type])`

- `address` `<string>` | `<net.SocketAddress>` An IPv4 or IPv6 address.

- `type` `<string>` Either `'ipv4'` or `'ipv6'`. **Default:** `'ipv4'`.

Adds a rule to block the given IP address.

`blockList.addRange(start, end[, type])`

- `start` `<string>` | `<net.SocketAddress>` The starting IPv4 or IPv6 address in the range.
- `end` `<string>` | `<net.SocketAddress>` The ending IPv4 or IPv6 address in the range.
- `type` `<string>` Either `'ipv4'` or `'ipv6'`. **Default:** `'ipv4'`.

Adds a rule to block a range of IP addresses from `start` (inclusive) to `end` (inclusive).

`blockList.addSubnet(net, prefix[, type])`

- `net` `<string>` | `<net.SocketAddress>` The network IPv4 or IPv6 address.
- `prefix` `<number>` The number of CIDR prefix bits. For IPv4, this must be a value between `0` and `32`. For IPv6, this must be between `0` and `128`.
- `type` `<string>` Either `'ipv4'` or `'ipv6'`. **Default:** `'ipv4'`.

Adds a rule to block a range of IP addresses specified as a subnet mask.

`blockList.check(address[, type])`

- `address` `<string>` | `<net.SocketAddress>` The IP address to check
- `type` `<string>` Either `'ipv4'` or `'ipv6'`. **Default:** `'ipv4'`.
- Returns: `<boolean>`

Returns `true` if the given IP address matches any of the rules added to the `BlockList`.

```
const blockList = new net.BlockList();
blockList.addAddress('123.123.123.123');
blockList.addRange('10.0.0.1', '10.0.0.10');
blockList.addSubnet('8592:757c:efae:4e45::', 64, 'ipv6');

console.log(blockList.check('123.123.123.123')); // Prints: true
console.log(blockList.check('10.0.0.3')); // Prints: true
console.log(blockList.check('222.111.111.222')); // Prints: false

// IPv6 notation for IPv4 addresses works:
console.log(blockList.check('::ffff:7b7b:7b7b', 'ipv6')); // Prints: true
console.log(blockList.check('::ffff:123.123.123.123', 'ipv6')); // Prints: true
```

`blockList.rules`

- Type: `<string[]>`

The list of rules added to the blocklist.

Class: `net.SocketAddress`

`new net.SocketAddress([options])`

- `options` `<Object>`

- `address` `<string>` The network address as either an IPv4 or IPv6 string. **Default:** `'127.0.0.1'` if `family` is `'ipv4'`; `:::` if `family` is `'ipv6'`.
- `family` `<string>` One of either `'ipv4'` or `'ipv6'`. ****Default**:** `'ipv4'`.
- `flowlabel` `<number>` An IPv6 flow-label used only if `family` is `'ipv6'`.
- `port` `<number>` An IP port.

socketaddress.address

- Type `<string>`

socketaddress.family

- Type `<string>` Either `'ipv4'` or `'ipv6'`.

socketaddress.flowlabel

- Type `<number>`

socketaddress.port

- Type `<number>`

Class: net.Server

- Extends: `<EventEmitter>`

This class is used to create a TCP or `IPC` server.

new net.Server([options][, connectionListener])

- `options` `<Object>` See `net.createServer([options][, connectionListener])`.
- `connectionListener` `<Function>` Automatically set as a listener for the `'connection'` event.
- Returns: `<net.Server>`

`net.Server` is an `EventEmitter` with the following events:

Event: 'close'

Emitted when the server closes. If connections exist, this event is not emitted until all connections are ended.

Event: 'connection'

- `<net.Socket>` The connection object

Emitted when a new connection is made. `socket` is an instance of `net.Socket`.

Event: 'error'

- `<Error>`

Emitted when an error occurs. Unlike `net.Socket`, the `'close'` event will **not** be emitted directly following this event unless `server.close()` is manually called. See the example in discussion of `server.listen()`.

Event: 'listening'

Emitted when the server has been bound after calling `server.listen()`.

server.address()

- Returns: `<Object> | <string> | <null>`

Returns the bound `address`, the address `family` name, and `port` of the server as reported by the operating system if listening on an IP socket (useful to find which port was assigned when getting an OS-assigned address): `{ port: 12346, family: 'IPv4', address: '127.0.0.1' }`.

For a server listening on a pipe or Unix domain socket, the name is returned as a string.

```
const server = net.createServer((socket) => {
  socket.end('goodbye\n');
}).on('error', (err) => {
  // Handle errors here.
  throw err;
});

// Grab an arbitrary unused port.
server.listen(() => {
  console.log('opened server on', server.address());
});
```

`server.address()` returns `null` before the `'listening'` event has been emitted or after calling `server.close()`.

server.close([callback])

- `callback` `<Function>` Called when the server is closed.
- Returns: `<net.Server>`

Stops the server from accepting new connections and keeps existing connections. This function is asynchronous, the server is finally closed when all connections are ended and the server emits a `'close'` event. The optional `callback` will be called once the `'close'` event occurs. Unlike that event, it will be called with an `Error` as its only argument if the server was not open when it was closed.

server.getConnections(callback)

- `callback` `<Function>`
- Returns: `<net.Server>`

Asynchronously get the number of concurrent connections on the server. Works when sockets were sent to forks.

Callback should take two arguments `err` and `count`.

server.listen()

Start a server listening for connections. A `net.Server` can be a TCP or an `IPC` server depending on what it listens to.

Possible signatures:

- `server.listen(handle[, backlog][, callback])`
- `server.listen(options[, callback])`
- `server.listen(path[, backlog][, callback])` for `IPC` servers
- `server.listen([port[, host[, backlog]]][, callback])` for TCP servers

This function is asynchronous. When the server starts listening, the `'listening'` event will be emitted. The last parameter `callback` will be added as a listener for the `'listening'` event.

All `listen()` methods can take a `backlog` parameter to specify the maximum length of the queue of pending connections. The actual length will be determined by the OS through `sysctl` settings such as `tcp_max_syn_backlog` and `somaxconn` on Linux. The default value of this parameter is 511 (not 512).

All `net.Socket` are set to `SO_REUSEADDR` (see `socket(7)` for details).

The `server.listen()` method can be called again if and only if there was an error during the first `server.listen()` call or `server.close()` has been called. Otherwise, an `ERR_SERVER_ALREADY_LISTEN` error will be thrown.

One of the most common errors raised when listening is `EADDRINUSE`. This happens when another server is already listening on the requested `port / path / handle`. One way to handle this would be to retry after a certain amount of time:

```
server.on('error', (e) => {
  if (e.code === 'EADDRINUSE') {
    console.log('Address in use, retrying...');
    setTimeout(() => {
      server.close();
      server.listen(PORT, HOST);
    }, 1000);
  }
});
```

`server.listen(handle[, backlog][, callback])`

- `handle` `<Object>`
- `backlog` `<number>` Common parameter of `server.listen()` functions
- `callback` `<Function>`
- Returns: `<net.Server>`

Start a server listening for connections on a given `handle` that has already been bound to a port, a Unix domain socket, or a Windows named pipe.

The `handle` object can be either a server, a socket (anything with an underlying `_handle` member), or an object with an `fd` member that is a valid file descriptor.

Listening on a file descriptor is not supported on Windows.

`server.listen(options[, callback])`

- `options` `<Object>` Required. Supports the following properties:
 - `port` `<number>`
 - `host` `<string>`
 - `path` `<string>` Will be ignored if `port` is specified. See [Identifying paths for IPC connections](#).
 - `backlog` `<number>` Common parameter of `server.listen()` functions.
 - `exclusive` `<boolean>` **Default: false**
 - `readableAll` `<boolean>` For IPC servers makes the pipe readable for all users. **Default: false**.
 - `writableAll` `<boolean>` For IPC servers makes the pipe writable for all users. **Default: false**.
 - `ipv6only` `<boolean>` For TCP servers, setting `ipv6only` to `true` will disable dual-stack support, i.e., binding to host `::` won't make `0.0.0.0` be bound. **Default: false**.
 - `signal` `<AbortSignal>` An `AbortSignal` that may be used to close a listening server.

- `callback` `<Function>` functions.
- Returns: `<net.Server>`

If `port` is specified, it behaves the same as `server.listen([port[, host[, backlog]]], callback)`. Otherwise, if `path` is specified, it behaves the same as `server.listen(path[, backlog], callback)`. If none of them is specified, an error will be thrown.

If `exclusive` is `false` (default), then cluster workers will use the same underlying handle, allowing connection handling duties to be shared. When `exclusive` is `true`, the handle is not shared, and attempted port sharing results in an error. An example which listens on an exclusive port is shown below.

```
server.listen({
  host: 'localhost',
  port: 80,
  exclusive: true
});
```

Starting an IPC server as root may cause the server path to be inaccessible for unprivileged users. Using `readableAll` and `writableAll` will make the server accessible for all users.

If the `signal` option is enabled, calling `.abort()` on the corresponding `AbortController` is similar to calling `.close()` on the server:

```
const controller = new AbortController();
server.listen({
  host: 'localhost',
  port: 80,
  signal: controller.signal
});
// Later, when you want to close the server.
controller.abort();
```

`server.listen(path[, backlog], callback)`

- `path` `<string>` Path the server should listen to. See [Identifying paths for IPC connections](#).
- `backlog` `<number>` Common parameter of `server.listen()` functions.
- `callback` `<Function>`.
- Returns: `<net.Server>`

Start an `IPC` server listening for connections on the given `path`.

`server.listen([port[, host[, backlog]]], callback)`

- `port` `<number>`
- `host` `<string>`
- `backlog` `<number>` Common parameter of `server.listen()` functions.
- `callback` `<Function>`.
- Returns: `<net.Server>`

Start a TCP server listening for connections on the given `port` and `host`.

If `port` is omitted or is 0, the operating system will assign an arbitrary unused port, which can be retrieved by using `server.address().port` after the `'listening'` event has been emitted.

If `host` is omitted, the server will accept connections on the **unspecified IPv6 address** (`::`) when IPv6 is available, or the **unspecified IPv4 address** (`0.0.0.0`) otherwise.

In most operating systems, listening to the **unspecified IPv6 address** (`::`) may cause the `net.Server` to also listen on the **unspecified IPv4 address** (`0.0.0.0`).

server.listening

- `<boolean>` Indicates whether or not the server is listening for connections.

server.maxConnections

- `<integer>`

Set this property to reject connections when the server's connection count gets high.

It is not recommended to use this option once a socket has been sent to a child with `child_process.fork()`.

server.ref()

- Returns: `<net.Server>`

Opposite of `unref()`, calling `ref()` on a previously `unref`ed server will *not* let the program exit if it's the only server left (the default behavior). If the server is `ref`ed calling `ref()` again will have no effect.

server.unref()

- Returns: `<net.Server>`

Calling `unref()` on a server will allow the program to exit if this is the only active server in the event system. If the server is already `unref`ed calling `unref()` again will have no effect.

Class: net.Socket

- Extends: `<stream.Duplex>`

This class is an abstraction of a TCP socket or a streaming **IPC** endpoint (uses named pipes on Windows, and Unix domain sockets otherwise). It is also an `EventEmitter`.

A `net.Socket` can be created by the user and used directly to interact with a server. For example, it is returned by `net.createConnection()`, so the user can use it to talk to the server.

It can also be created by Node.js and passed to the user when a connection is received. For example, it is passed to the listeners of a `'connection'` event emitted on a `net.Server`, so the user can use it to interact with the client.

new net.Socket([options])

- `options` `<Object>` Available options are:
 - `fd` `<number>` If specified, wrap around an existing socket with the given file descriptor, otherwise a new socket will be created.
 - `allowHalfOpen` `<boolean>` If set to `false`, then the socket will automatically end the writable side when the readable side ends. See `net.createServer()` and the `'end'` event for details. **Default:** `false`.
 - `readable` `<boolean>` Allow reads on the socket when an `fd` is passed, otherwise ignored. **Default:** `false`.
 - `writable` `<boolean>` Allow writes on the socket when an `fd` is passed, otherwise ignored. **Default:** `false`.
 - `signal` `<AbortSignal>` An Abort signal that may be used to destroy the socket.
- Returns: `<net.Socket>`

Creates a new socket object.

The newly created socket can be either a TCP socket or a streaming `IPC` endpoint, depending on what it `connect()` to.

Event: 'close'

- `hadError` `<boolean>` `true` if the socket had a transmission error.

Emitted once the socket is fully closed. The argument `hadError` is a boolean which says if the socket was closed due to a transmission error.

Event: 'connect'

Emitted when a socket connection is successfully established. See `net.createConnection()`.

Event: 'data'

- `<Buffer>` | `<string>`

Emitted when data is received. The argument `data` will be a `Buffer` or `String`. Encoding of data is set by `socket.setEncoding()`.

The data will be lost if there is no listener when a `Socket` emits a `'data'` event.

Event: 'drain'

Emitted when the write buffer becomes empty. Can be used to throttle uploads.

See also: the return values of `socket.write()`.

Event: 'end'

Emitted when the other end of the socket signals the end of transmission, thus ending the readable side of the socket.

By default (`allowHalfOpen` is `false`) the socket will send an end of transmission packet back and destroy its file descriptor once it has written out its pending write queue. However, if `allowHalfOpen` is set to `true`, the socket will not automatically `end()` its writable side, allowing the user to write arbitrary amounts of data. The user must call `end()` explicitly to close the connection (i.e. sending a FIN packet back).

Event: 'error'

- `<Error>`

Emitted when an error occurs. The `'close'` event will be called directly following this event.

Event: 'lookup'

Emitted after resolving the host name but before connecting. Not applicable to Unix sockets.

- `err` `<Error>` | `<null>` The error object. See `dns.lookup()`.
- `address` `<string>` The IP address.
- `family` `<string>` | `<null>` The address type. See `dns.lookup()`.
- `host` `<string>` The host name.

Event: 'ready'

Emitted when a socket is ready to be used.

Triggered immediately after `'connect'`.

Event: 'timeout'

Emitted if the socket times out from inactivity. This is only to notify that the socket has been idle. The user must manually close the connection.

See also: `socket.setTimeout()`.

socket.address()

- Returns: `<Object>`

Returns the bound `address`, the address `family` name and `port` of the socket as reported by the operating system: `{ port: 12346, family: 'IPv4', address: '127.0.0.1' }`

socket.bufferSize

Stability: 0 - Deprecated: Use `writable.writableLength` instead.

- `<integer>`

This property shows the number of characters buffered for writing. The buffer may contain strings whose length after encoding is not yet known. So this number is only an approximation of the number of bytes in the buffer.

`net.Socket` has the property that `socket.write()` always works. This is to help users get up and running quickly. The computer cannot always keep up with the amount of data that is written to a socket. The network connection simply might be too slow. Node.js will internally queue up the data written to a socket and send it out over the wire when it is possible.

The consequence of this internal buffering is that memory may grow. Users who experience large or growing `bufferSize` should attempt to "throttle" the data flows in their program with `socket.pause()` and `socket.resume()`.

socket.bytesRead

- `<integer>`

The amount of received bytes.

socket.bytesWritten

- `<integer>`

The amount of bytes sent.

socket.connect()

Initiate a connection on a given socket.

Possible signatures:

- `socket.connect(options[, connectListener])`
- `socket.connect(path[, connectListener])` for IPC connections.
- `socket.connect(port[, host][, connectListener])` for TCP connections.
- Returns: `<net.Socket>` The socket itself.

This function is asynchronous. When the connection is established, the `'connect'` event will be emitted. If there is a problem connecting, instead of a `'connect'` event, an `'error'` event will be emitted with the error passed to the `'error'` listener. The last parameter `connectListener`, if supplied, will be added as a listener for the `'connect'` event **once**.

This function should only be used for reconnecting a socket after `'close'` has been emitted or otherwise it may lead to undefined behavior.

socket.connect(options[, connectListener])

- `options` `<Object>`
- `connectListener` `<Function>` Common parameter of `socket.connect()` methods. Will be added as a listener for the `'connect'` event once.
- Returns: `<net.Socket>` The socket itself.

Initiate a connection on a given socket. Normally this method is not needed, the socket should be created and opened with `net.createConnection()`. Use this only when implementing a custom Socket.

For TCP connections, available `options` are:

- `port` `<number>` Required. Port the socket should connect to.
- `host` `<string>` Host the socket should connect to. **Default:** `'localhost'`.
- `localAddress` `<string>` Local address the socket should connect from.
- `localPort` `<number>` Local port the socket should connect from.
- `family` `<number>`: Version of IP stack. Must be `4`, `6`, or `0`. The value `0` indicates that both IPv4 and IPv6 addresses are allowed. **Default:** `0`.
- `hints` `<number>` Optional `dns.lookup()` hints.
- `lookup` `<Function>` Custom lookup function. **Default:** `dns.lookup()`.

For IPC connections, available `options` are:

- `path` `<string>` Required. Path the client should connect to. See [Identifying paths for IPC connections](#). If provided, the TCP-specific options above are ignored.

For both types, available `options` include:

- `onread` `<Object>` If specified, incoming data is stored in a single `buffer` and passed to the supplied `callback` when data arrives on the socket. This will cause the streaming functionality to not provide any data. The socket will emit events like `'error'`, `'end'`, and `'close'` as usual. Methods like `pause()` and `resume()` will also behave as expected.
 - `buffer` `<Buffer>` | `<Uint8Array>` | `<Function>` Either a reusable chunk of memory to use for storing incoming data or a function that returns such.
 - `callback` `<Function>` This function is called for every chunk of incoming data. Two arguments are passed to it: the number of bytes written to `buffer` and a reference to `buffer`. Return `false` from this function to implicitly `pause()` the socket. This function will be executed in the global context.

Following is an example of a client using the `onread` option:

```
const net = require('net');
net.connect({
  port: 80,
  onread: {
    // Reuses a 4KiB Buffer for every read from the socket.
    buffer: Buffer.alloc(4 * 1024),
    callback: function(nread, buf) {
      // Received data is available in `buf` from 0 to `nread`.
      console.log(buf.toString('utf8', 0, nread));
    }
  }
});
```

socket.connect(path[, connectListener])

- `path` `<string>` Path the client should connect to. See [Identifying paths for IPC connections](#) .
- `connectListener` `<Function>` Common parameter of `socket.connect()` methods. Will be added as a listener for the `'connect'` event once.
- Returns: `<net.Socket>` The socket itself.

Initiate an `IPC` connection on the given socket.

Alias to `socket.connect(options[, connectListener])` called with `{ path: path }` as `options` .

`socket.connect(port[, host][, connectListener])`

- `port` `<number>` Port the client should connect to.
- `host` `<string>` Host the client should connect to.
- `connectListener` `<Function>` Common parameter of `socket.connect()` methods. Will be added as a listener for the `'connect'` event once.
- Returns: `<net.Socket>` The socket itself.

Initiate a TCP connection on the given socket.

Alias to `socket.connect(options[, connectListener])` called with `{port: port, host: host}` as `options` .

`socket.connecting`

- `<boolean>`

If `true` , `socket.connect(options[, connectListener])` was called and has not yet finished. It will stay `true` until the socket becomes connected, then it is set to `false` and the `'connect'` event is emitted. Note that the `socket.connect(options[, connectListener])` callback is a listener for the `'connect'` event.

`socket.destroy([error])`

- `error` `<Object>`
- Returns: `<net.Socket>`

Ensures that no more I/O activity happens on this socket. Destroys the stream and closes the connection.

See `writable.destroy()` for further details.

`socket.destroyed`

- `<boolean>` Indicates if the connection is destroyed or not. Once a connection is destroyed no further data can be transferred using it.

See `writable.destroyed` for further details.

`socket.end([data[, encoding]][, callback])`

- `data` `<string> | <Buffer> | <Uint8Array>`
- `encoding` `<string>` Only used when data is `string` . Default: `'utf8'` .
- `callback` `<Function>` Optional callback for when the socket is finished.
- Returns: `<net.Socket>` The socket itself.

Half-closes the socket. i.e., it sends a FIN packet. It is possible the server will still send some data.

See `writable.end()` for further details.

`socket.localAddress`

- `<string>`

The string representation of the local IP address the remote client is connecting on. For example, in a server listening on `'0.0.0.0'`, if a client connects on `'192.168.1.1'`, the value of `socket.localAddress` would be `'192.168.1.1'`.

socket.localPort

- `<integer>`

The numeric representation of the local port. For example, `80` or `21`.

socket.pause()

- Returns: `<net.Socket>` The socket itself.

Pauses the reading of data. That is, `'data'` events will not be emitted. Useful to throttle back an upload.

socket.pending

- `<boolean>`

This is `true` if the socket is not connected yet, either because `.connect()` has not yet been called or because it is still in the process of connecting (see `socket.connecting`).

socket.ref()

- Returns: `<net.Socket>` The socket itself.

Opposite of `unref()`, calling `ref()` on a previously `unref`ed socket will *not* let the program exit if it's the only socket left (the default behavior). If the socket is `ref`ed calling `ref` again will have no effect.

socket.remoteAddress

- `<string>`

The string representation of the remote IP address. For example, `'74.125.127.100'` or `'2001:4860:a005::68'`. Value may be `undefined` if the socket is destroyed (for example, if the client disconnected).

socket.remoteFamily

- `<string>`

The string representation of the remote IP family. `'IPv4'` or `'IPv6'`.

socket.remotePort

- `<integer>`

The numeric representation of the remote port. For example, `80` or `21`.

socket.resume()

- Returns: `<net.Socket>` The socket itself.

Resumes reading after a call to `socket.pause()`.

socket.setEncoding([encoding])

- `encoding` `<string>`
- Returns: `<net.Socket>` The socket itself.

Set the encoding for the socket as a [ReadableStream](#). See `readable.setEncoding()` for more information.

`socket.setKeepAlive([enable][, initialDelay])`

- `enable` [<boolean>](#) **Default:** `false`
- `initialDelay` [<number>](#) **Default:** `0`
- Returns: [<net.Socket>](#) The socket itself.

Enable/disable keep-alive functionality, and optionally set the initial delay before the first keepalive probe is sent on an idle socket.

Set `initialDelay` (in milliseconds) to set the delay between the last data packet received and the first keepalive probe. Setting `0` for `initialDelay` will leave the value unchanged from the default (or previous) setting.

Enabling the keep-alive functionality will set the following socket options:

- `SO_KEEPALIVE=1`
- `TCP_KEEPIDLE=initialDelay`
- `TCP_KEEPCNT=10`
- `TCP_KEEPINTVL=1`

`socket.setNoDelay([noDelay])`

- `noDelay` [<boolean>](#) **Default:** `true`
- Returns: [<net.Socket>](#) The socket itself.

Enable/disable the use of Nagle's algorithm.

When a TCP connection is created, it will have Nagle's algorithm enabled.

Nagle's algorithm delays data before it is sent via the network. It attempts to optimize throughput at the expense of latency.

Passing `true` for `noDelay` or not passing an argument will disable Nagle's algorithm for the socket. Passing `false` for `noDelay` will enable Nagle's algorithm.

`socket.setTimeout(timeout[, callback])`

- `timeout` [<number>](#)
- `callback` [<Function>](#)
- Returns: [<net.Socket>](#) The socket itself.

Sets the socket to timeout after `timeout` milliseconds of inactivity on the socket. By default `net.Socket` do not have a timeout.

When an idle timeout is triggered the socket will receive a `'timeout'` event but the connection will not be severed. The user must manually call `socket.end()` or `socket.destroy()` to end the connection.

```
socket.setTimeout(3000);
socket.on('timeout', () => {
  console.log('socket timeout');
  socket.end();
});
```

If `timeout` is 0, then the existing idle timeout is disabled.

The optional `callback` parameter will be added as a one-time listener for the `'timeout'` event.

socket.timeout

- `<number> | <undefined>`

The socket timeout in milliseconds as set by `socket.setTimeout()`. It is `undefined` if a timeout has not been set.

socket.unref()

- Returns: `<net.Socket>` The socket itself.

Calling `unref()` on a socket will allow the program to exit if this is the only active socket in the event system. If the socket is already `unref`ed calling `unref()` again will have no effect.

socket.write(data[, encoding][, callback])

- `data` `<string> | <Buffer> | <Uint8Array>`
- `encoding` `<string>` Only used when data is `string`. Default: `utf8`.
- `callback` `<Function>`
- Returns: `<boolean>`

Sends data on the socket. The second parameter specifies the encoding in the case of a string. It defaults to UTF8 encoding.

Returns `true` if the entire data was flushed successfully to the kernel buffer. Returns `false` if all or part of the data was queued in user memory. `'drain'` will be emitted when the buffer is again free.

The optional `callback` parameter will be executed when the data is finally written out, which may not be immediately.

See `Writable` stream `write()` method for more information.

socket.readyState

- `<string>`

This property represents the state of the connection as a string.

- If the stream is connecting `socket.readyState` is `opening`.
- If the stream is readable and writable, it is `open`.
- If the stream is readable and not writable, it is `readOnly`.
- If the stream is not readable and writable, it is `writeOnly`.

net.connect()

Aliases to `net.createConnection()`.

Possible signatures:

- `net.connect(options[, connectListener])`
- `net.connect(path[, connectListener])` for IPC connections.
- `net.connect(port[, host][, connectListener])` for TCP connections.

net.connect(options[, connectListener])

- `options` `<Object>`
- `connectListener` `<Function>`
- Returns: `<net.Socket>`

Alias to `net.createConnection(options[, connectListener])`.

`net.connect(path[, connectListener])`

- `path` `<string>`
- `connectListener` `<Function>`
- Returns: `<net.Socket>`

Alias to `net.createConnection(path[, connectListener])`.

`net.connect(port[, host][, connectListener])`

- `port` `<number>`
- `host` `<string>`
- `connectListener` `<Function>`
- Returns: `<net.Socket>`

Alias to `net.createConnection(port[, host][, connectListener])`.

`net.createConnection()`

A factory function, which creates a new `net.Socket`, immediately initiates connection with `socket.connect()`, then returns the `net.Socket` that starts the connection.

When the connection is established, a `'connect'` event will be emitted on the returned socket. The last parameter `connectListener`, if supplied, will be added as a listener for the `'connect'` event **once**.

Possible signatures:

- `net.createConnection(options[, connectListener])`
- `net.createConnection(path[, connectListener])` for IPC connections.
- `net.createConnection(port[, host][, connectListener])` for TCP connections.

The `net.connect()` function is an alias to this function.

`net.createConnection(options[, connectListener])`

- `options` `<Object>` Required. Will be passed to both the `new net.Socket([options])` call and the `socket.connect(options[, connectListener])` method.
- `connectListener` `<Function>` Common parameter of the `net.createConnection()` functions. If supplied, will be added as a listener for the `'connect'` event on the returned socket once.
- Returns: `<net.Socket>` The newly created socket used to start the connection.

For available options, see `new net.Socket([options])` and `socket.connect(options[, connectListener])`.

Additional options:

- `timeout` `<number>` If set, will be used to call `socket.setTimeout(timeout)` after the socket is created, but before it starts the connection.

Following is an example of a client of the echo server described in the `net.createServer()` section:

```
const net = require('net');
const client = net.createConnection({ port: 8124 }, () => {
```

```
// 'connect' listener.
console.log('connected to server!');
client.write('world!\r\n');
});
client.on('data', (data) => {
  console.log(data.toString());
  client.end();
});
client.on('end', () => {
  console.log('disconnected from server');
});
```

To connect on the socket `/tmp/echo.sock`:

```
const client = net.createConnection({ path: '/tmp/echo.sock' });
```

net.createConnection(path[, connectListener])

- **path** `<string>` Path the socket should connect to. Will be passed to `socket.connect(path[, connectListener])`. See [Identifying paths for IPC connections](#).
- **connectListener** `<Function>` Common parameter of the `net.createConnection()` functions, an "once" listener for the 'connect' event on the initiating socket. Will be passed to `socket.connect(path[, connectListener])`.
- Returns: `<net.Socket>` The newly created socket used to start the connection.

Initiates an [IPC](#) connection.

This function creates a new `net.Socket` with all options set to default, immediately initiates connection with `socket.connect(path[, connectListener])`, then returns the `net.Socket` that starts the connection.

net.createConnection(port[, host][, connectListener])

- **port** `<number>` Port the socket should connect to. Will be passed to `socket.connect(port[, host][, connectListener])`.
- **host** `<string>` Host the socket should connect to. Will be passed to `socket.connect(port[, host][, connectListener])`. **Default:** 'localhost'.
- **connectListener** `<Function>` Common parameter of the `net.createConnection()` functions, an "once" listener for the 'connect' event on the initiating socket. Will be passed to `socket.connect(port[, host][, connectListener])`.
- Returns: `<net.Socket>` The newly created socket used to start the connection.

Initiates a TCP connection.

This function creates a new `net.Socket` with all options set to default, immediately initiates connection with `socket.connect(port[, host][, connectListener])`, then returns the `net.Socket` that starts the connection.

net.createServer([options][, connectionListener])

- **options** `<Object>`
 - **allowHalfOpen** `<boolean>` If set to `false`, then the socket will automatically end the writable side when the readable side ends. **Default:** `false`.
 - **pauseOnConnect** `<boolean>` Indicates whether the socket should be paused on incoming connections. **Default:** `false`.
- **connectionListener** `<Function>` Automatically set as a listener for the 'connection' event.
- Returns: `<net.Server>`

Creates a new TCP or **IPC** server.

If `allowHalfOpen` is set to `true`, when the other end of the socket signals the end of transmission, the server will only send back the end of transmission when `socket.end()` is explicitly called. For example, in the context of TCP, when a FIN packet is received, a FIN packet is sent back only when `socket.end()` is explicitly called. Until then the connection is half-closed (non-readable but still writable). See `'end'` event and **RFC 1122** (section 4.2.2.13) for more information.

If `pauseOnConnect` is set to `true`, then the socket associated with each incoming connection will be paused, and no data will be read from its handle. This allows connections to be passed between processes without any data being read by the original process. To begin reading data from a paused socket, call `socket.resume()`.

The server can be a TCP server or an **IPC** server, depending on what it `listen()` to.

Here is an example of a TCP echo server which listens for connections on port 8124:

```
const net = require('net');
const server = net.createServer((c) => {
  // 'connection' listener.
  console.log('client connected');
  c.on('end', () => {
    console.log('client disconnected');
  });
  c.write('hello\r\n');
  c.pipe(c);
});
server.on('error', (err) => {
  throw err;
});
server.listen(8124, () => {
  console.log('server bound');
});
```

Test this by using `telnet`:

```
$ telnet localhost 8124
```

To listen on the socket `/tmp/echo.sock`:

```
server.listen('/tmp/echo.sock', () => {
  console.log('server bound');
});
```

Use `nc` to connect to a Unix domain socket server:

```
$ nc -U /tmp/echo.sock
```

net.isIP(input)

- `input` `<string>`

- Returns: `<integer>`

Tests if input is an IP address. Returns `0` for invalid strings, returns `4` for IP version 4 addresses, and returns `6` for IP version 6 addresses.

`net.isIPv4(input)`

- `input` `<string>`
- Returns: `<boolean>`

Returns `true` if input is a version 4 IP address, otherwise returns `false`.

`net.isIPv6(input)`

- `input` `<string>`
- Returns: `<boolean>`

Returns `true` if input is a version 6 IP address, otherwise returns `false`.

OS

Stability: 2 - Stable

Source Code: [lib/os.js](#)

The `os` module provides operating system-related utility methods and properties. It can be accessed using:

```
const os = require('os');
```

`os.EOL`

- `<string>`

The operating system-specific end-of-line marker.

- `\n` on POSIX
- `\r\n` on Windows

`os.arch()`

- Returns: `<string>`

Returns the operating system CPU architecture for which the Node.js binary was compiled. Possible values are `'arm'`, `'arm64'`, `'ia32'`, `'mips'`, `'mipsel'`, `'ppc'`, `'ppc64'`, `'s390'`, `'s390x'`, `'x32'`, and `'x64'`.

The return value is equivalent to `process.arch`.

`os.constants`

- `<Object>`

Contains commonly used operating system-specific constants for error codes, process signals, and so on. The specific constants defined are described in [OS constants](#).

os.cpus()

- Returns: `<Object[]>`

Returns an array of objects containing information about each logical CPU core.

The properties included on each object include:

- `model` `<string>`
- `speed` `<number>` (in MHz)
- `times` `<Object>`
 - `user` `<number>` The number of milliseconds the CPU has spent in user mode.
 - `nice` `<number>` The number of milliseconds the CPU has spent in nice mode.
 - `sys` `<number>` The number of milliseconds the CPU has spent in sys mode.
 - `idle` `<number>` The number of milliseconds the CPU has spent in idle mode.
 - `irq` `<number>` The number of milliseconds the CPU has spent in irq mode.

```
[
  {
    model: 'Intel(R) Core(TM) i7 CPU           860  @ 2.80GHz',
    speed: 2926,
    times: {
      user: 252020,
      nice: 0,
      sys: 30340,
      idle: 1070356870,
      irq: 0
    }
  },
  {
    model: 'Intel(R) Core(TM) i7 CPU           860  @ 2.80GHz',
    speed: 2926,
    times: {
      user: 306960,
      nice: 0,
      sys: 26980,
      idle: 1071569080,
      irq: 0
    }
  },
  {
    model: 'Intel(R) Core(TM) i7 CPU           860  @ 2.80GHz',
    speed: 2926,
    times: {
      user: 248450,
      nice: 0,
      sys: 21750,
```



```

      idle: 1070919370,
      irq: 0
    },
    {
      model: 'Intel(R) Core(TM) i7 CPU           860  @ 2.80GHz',
      speed: 2926,
      times: {
        user: 256880,
        nice: 0,
        sys: 19430,
        idle: 1070905480,
        irq: 20
      }
    },
  ],
]

```

`nice` values are POSIX-only. On Windows, the `nice` values of all processors are always 0.

os.devNull

- `<string>`

The platform-specific file path of the null device.

- `\\.\nul` on Windows
- `/dev/null` on POSIX

os.endianness()

- Returns: `<string>`

Returns a string identifying the endianness of the CPU for which the Node.js binary was compiled.

Possible values are `'BE'` for big endian and `'LE'` for little endian.

os.freemem()

- Returns: `<integer>`

Returns the amount of free system memory in bytes as an integer.

os.getPriority([pid])

- `pid` `<integer>` The process ID to retrieve scheduling priority for. **Default:** `0`.
- Returns: `<integer>`

Returns the scheduling priority for the process specified by `pid`. If `pid` is not provided or is `0`, the priority of the current process is returned.

os.homedir()

- Returns: `<string>`

Returns the string path of the current user's home directory.

On POSIX, it uses the `$HOME` environment variable if defined. Otherwise it uses the `effective UID` to look up the user's home directory.

On Windows, it uses the `USERPROFILE` environment variable if defined. Otherwise it uses the path to the profile directory of the current user.

os.hostname()

- Returns: `<string>`

Returns the host name of the operating system as a string.

os.loadavg()

- Returns: `<number[]>`

Returns an array containing the 1, 5, and 15 minute load averages.

The load average is a measure of system activity calculated by the operating system and expressed as a fractional number.

The load average is a Unix-specific concept. On Windows, the return value is always `[0, 0, 0]`.

os.networkInterfaces()

- Returns: `<Object>`

Returns an object containing network interfaces that have been assigned a network address.

Each key on the returned object identifies a network interface. The associated value is an array of objects that each describe an assigned network address.

The properties available on the assigned network address object include:

- `address` `<string>` The assigned IPv4 or IPv6 address
- `netmask` `<string>` The IPv4 or IPv6 network mask
- `family` `<string>` Either `IPv4` or `IPv6`
- `mac` `<string>` The MAC address of the network interface
- `internal` `<boolean>` `true` if the network interface is a loopback or similar interface that is not remotely accessible; otherwise `false`
- `scopeid` `<number>` The numeric IPv6 scope ID (only specified when `family` is `IPv6`)
- `cidr` `<string>` The assigned IPv4 or IPv6 address with the routing prefix in CIDR notation. If the `netmask` is invalid, this property is set to `null`.

```
{
  lo: [
    {
      address: '127.0.0.1',
      netmask: '255.0.0.0',
      family: 'IPv4',
      mac: '00:00:00:00:00:00',
      internal: true,
      cidr: '127.0.0.1/8'
    },
  ],
}
```

```

{
  address: '::1',
  netmask: 'ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff',
  family: 'IPv6',
  mac: '00:00:00:00:00:00',
  scopeid: 0,
  internal: true,
  cidr: '::1/128'
}
],
eth0: [
  {
    address: '192.168.1.108',
    netmask: '255.255.255.0',
    family: 'IPv4',
    mac: '01:02:03:0a:0b:0c',
    internal: false,
    cidr: '192.168.1.108/24'
  },
  {
    address: 'fe80::a00:27ff:fe4e:66a1',
    netmask: 'ffff:ffff:ffff:ffff::',
    family: 'IPv6',
    mac: '01:02:03:0a:0b:0c',
    scopeid: 1,
    internal: false,
    cidr: 'fe80::a00:27ff:fe4e:66a1/64'
  }
]
}

```

os.platform()

- Returns: `<string>`

Returns a string identifying the operating system platform. The value is set at compile time. Possible values are `'aix'`, `'darwin'`, `'freebsd'`, `'linux'`, `'openbsd'`, `'sunos'`, and `'win32'`.

The return value is equivalent to `process.platform`.

The value `'android'` may also be returned if Node.js is built on the Android operating system. [Android support is experimental](#).

os.release()

- Returns: `<string>`

Returns the operating system as a string.

On POSIX systems, the operating system release is determined by calling `uname(3)`. On Windows, `GetVersionEx()` is used. See <https://en.wikipedia.org/wiki/Uname#Examples> for more information.

os.setPriority([pid,]priority)

- `pid` `<integer>` The process ID to set scheduling priority for. **Default:** `0`.
- `priority` `<integer>` The scheduling priority to assign to the process.

Attempts to set the scheduling priority for the process specified by `pid`. If `pid` is not provided or is `0`, the process ID of the current process is used.

The `priority` input must be an integer between `-20` (high priority) and `19` (low priority). Due to differences between Unix priority levels and Windows priority classes, `priority` is mapped to one of six priority constants in `os.constants.priority`. When retrieving a process priority level, this range mapping may cause the return value to be slightly different on Windows. To avoid confusion, set `priority` to one of the priority constants.

On Windows, setting priority to `PRIORITY_HIGHEST` requires elevated user privileges. Otherwise the set priority will be silently reduced to `PRIORITY_HIGH`.

os.tmpdir()

- Returns: `<string>`

Returns the operating system's default directory for temporary files as a string.

os.totalmem()

- Returns: `<integer>`

Returns the total amount of system memory in bytes as an integer.

os.type()

- Returns: `<string>`

Returns the operating system name as returned by `uname(3)`. For example, it returns `'Linux'` on Linux, `'Darwin'` on macOS, and `'Windows_NT'` on Windows.

See <https://en.wikipedia.org/wiki/Uname#Examples> for additional information about the output of running `uname(3)` on various operating systems.

os.uptime()

- Returns: `<integer>`

Returns the system uptime in number of seconds.

os.userInfo([options])

- `options` `<Object>`
 - `encoding` `<string>` Character encoding used to interpret resulting strings. If `encoding` is set to `'buffer'`, the `username`, `shell`, and `homedir` values will be `Buffer` instances. **Default:** `'utf8'`.
- Returns: `<Object>`

Returns information about the currently effective user. On POSIX platforms, this is typically a subset of the password file. The returned object includes the `username`, `uid`, `gid`, `shell`, and `homedir`. On Windows, the `uid` and `gid` fields are `-1`, and `shell` is `null`.

The value of `homedir` returned by `os.userInfo()` is provided by the operating system. This differs from the result of `os.homedir()`, which queries environment variables for the home directory before falling back to the operating system response.

Throws a `SystemError` if a user has no `username` or `homedir`.

os.version()

- Returns `<string>`

Returns a string identifying the kernel version.

On POSIX systems, the operating system release is determined by calling `uname(3)`. On Windows, `RtlGetVersion()` is used, and if it is not available, `GetVersionExW()` will be used. See <https://en.wikipedia.org/wiki/Uname#Examples> for more information.

OS constants

The following constants are exported by `os.constants`.

Not all constants will be available on every operating system.

Signal constants

The following signal constants are exported by `os.constants.signals`.

Constant	Description
<code>SIGHUP</code>	Sent to indicate when a controlling terminal is closed or a parent process exits.
<code>SIGINT</code>	Sent to indicate when a user wishes to interrupt a process (<code>ctrl</code> + <code>c</code>).
<code>SIGQUIT</code>	Sent to indicate when a user wishes to terminate a process and perform a core dump.
<code>SIGILL</code>	Sent to a process to notify that it has attempted to perform an illegal, malformed, unknown, or privileged instruction.
<code>SIGTRAP</code>	Sent to a process when an exception has occurred.
<code>SIGABRT</code>	Sent to a process to request that it abort.
<code>SIGIOT</code>	Synonym for <code>SIGABRT</code>
<code>SIGBUS</code>	Sent to a process to notify that it has caused a bus error.
<code>SIGFPE</code>	Sent to a process to notify that it has performed an illegal arithmetic operation.
<code>SIGKILL</code>	Sent to a process to terminate it immediately.
<code>SIGUSR1</code> <code>SIGUSR2</code>	Sent to a process to identify user-defined conditions.
<code>SIGSEGV</code>	Sent to a process to notify of a segmentation fault.
<code>SIGPIPE</code>	Sent to a process when it has attempted to write to a disconnected pipe.
<code>SIGALRM</code>	Sent to a process when a system timer elapses.
<code>SIGTERM</code>	Sent to a process to request termination.
<code>SIGCHLD</code>	Sent to a process when a child process terminates.
<code>SIGSTKFLT</code>	Sent to a process to indicate a stack fault on a coprocessor.

SIGCONT	Sent to instruct the operating system to continue a paused process.
SIGSTOP	Sent to instruct the operating system to halt a process.
SIGTSTP	Sent to a process to request it to stop.
SIGBREAK	Sent to indicate when a user wishes to interrupt a process.
SIGTTIN	Sent to a process when it reads from the TTY while in the background.
SIGTTOU	Sent to a process when it writes to the TTY while in the background.
SIGURG	Sent to a process when a socket has urgent data to read.
SIGXCPU	Sent to a process when it has exceeded its limit on CPU usage.
SIGXFSZ	Sent to a process when it grows a file larger than the maximum allowed.
SIGVTALRM	Sent to a process when a virtual timer has elapsed.
SIGPROF	Sent to a process when a system timer has elapsed.
SIGWINCH	Sent to a process when the controlling terminal has changed its size.
SIGIO	Sent to a process when I/O is available.
SIGPOLL	Synonym for SIGIO
SIGLOST	Sent to a process when a file lock has been lost.
SIGPWR	Sent to a process to notify of a power failure.
SIGINFO	Synonym for SIGPWR
SIGSYS	Sent to a process to notify of a bad argument.
SIGUNUSED	Synonym for SIGSYS

Error constants

The following error constants are exported by `os.constants.errno`.

POSIX error constants

Constant	Description
E2BIG	Indicates that the list of arguments is longer than expected.
EACCES	Indicates that the operation did not have sufficient permissions.
EADDRINUSE	Indicates that the network address is already in use.
EADDRNOTAVAIL	Indicates that the network address is currently unavailable for use.
EAFNOSUPPORT	Indicates that the network address family is not supported.
EAGAIN	Indicates that there is no data available and to try the operation again later.

EALREADY	Indicates that the socket already has a pending connection in progress.
EBADF	Indicates that a file descriptor is not valid.
EBADMSG	Indicates an invalid data message.
EBUSY	Indicates that a device or resource is busy.
ECANCELED	Indicates that an operation was canceled.
ECHILD	Indicates that there are no child processes.
ECONNABORTED	Indicates that the network connection has been aborted.
ECONNREFUSED	Indicates that the network connection has been refused.
ECONNRESET	Indicates that the network connection has been reset.
EDEADLK	Indicates that a resource deadlock has been avoided.
EDESTADDRREQ	Indicates that a destination address is required.
EDOM	Indicates that an argument is out of the domain of the function.
EDQUOT	Indicates that the disk quota has been exceeded.
EEXIST	Indicates that the file already exists.
EFAULT	Indicates an invalid pointer address.
EFBIG	Indicates that the file is too large.
EHOSTUNREACH	Indicates that the host is unreachable.
EIDRM	Indicates that the identifier has been removed.
EILSEQ	Indicates an illegal byte sequence.
EINPROGRESS	Indicates that an operation is already in progress.
EINTR	Indicates that a function call was interrupted.
EINVAL	Indicates that an invalid argument was provided.
EIO	Indicates an otherwise unspecified I/O error.
EISCONN	Indicates that the socket is connected.
EISDIR	Indicates that the path is a directory.
ELOOP	Indicates too many levels of symbolic links in a path.
EMFILE	Indicates that there are too many open files.
EMLINK	Indicates that there are too many hard links to a file.

EMSGSIZE	Indicates that the provided message is too long.
EMULTIHOP	Indicates that a multihop was attempted.
ENAMETOOLONG	Indicates that the filename is too long.
ENETDOWN	Indicates that the network is down.
ENETRESET	Indicates that the connection has been aborted by the network.
ENETUNREACH	Indicates that the network is unreachable.
ENFILE	Indicates too many open files in the system.
ENOBUFS	Indicates that no buffer space is available.
ENODATA	Indicates that no message is available on the stream head read queue.
ENODEV	Indicates that there is no such device.
ENOENT	Indicates that there is no such file or directory.
ENOEXEC	Indicates an exec format error.
ENOLCK	Indicates that there are no locks available.
ENOLINK	Indications that a link has been severed.
ENOMEM	Indicates that there is not enough space.
ENOMSG	Indicates that there is no message of the desired type.
ENOPROTOPT	Indicates that a given protocol is not available.
ENOSPC	Indicates that there is no space available on the device.
ENOSR	Indicates that there are no stream resources available.
ENOSTR	Indicates that a given resource is not a stream.
ENOSYS	Indicates that a function has not been implemented.
ENOTCONN	Indicates that the socket is not connected.
ENOTDIR	Indicates that the path is not a directory.
ENOTEMPTY	Indicates that the directory is not empty.
ENOTSOCK	Indicates that the given item is not a socket.
ENOTSUP	Indicates that a given operation is not supported.
ENOTTY	Indicates an inappropriate I/O control operation.
ENXIO	Indicates no such device or address.
EOPNOTSUPP	Indicates that an operation is not supported on the socket. Although ENOTSUP and EOPNOTSUPP have the same value on Linux, according to POSIX.1 these error values should be distinct.)

EOVERFLOW	Indicates that a value is too large to be stored in a given data type.
EPERM	Indicates that the operation is not permitted.
EPIPE	Indicates a broken pipe.
EPROTO	Indicates a protocol error.
EPROTONOSUPPORT	Indicates that a protocol is not supported.
EPROTOTYPE	Indicates the wrong type of protocol for a socket.
ERANGE	Indicates that the results are too large.
EROFS	Indicates that the file system is read only.
ESPIPE	Indicates an invalid seek operation.
ESRCH	Indicates that there is no such process.
ESTALE	Indicates that the file handle is stale.
ETIME	Indicates an expired timer.
ETIMEDOUT	Indicates that the connection timed out.
ETXTBSY	Indicates that a text file is busy.
EWouldBlock	Indicates that the operation would block.
EXDEV	Indicates an improper link.

Windows-specific error constants

The following error codes are specific to the Windows operating system.

Constant	Description
WSAEINTR	Indicates an interrupted function call.
WSAEBADF	Indicates an invalid file handle.
WSAEACCES	Indicates insufficient permissions to complete the operation.
WSAEFAULT	Indicates an invalid pointer address.
WSAEINVAL	Indicates that an invalid argument was passed.
WSAEMFILE	Indicates that there are too many open files.
WSAEWOULDBLOCK	Indicates that a resource is temporarily unavailable.
WSAEINPROGRESS	Indicates that an operation is currently in progress.
WSAEALREADY	Indicates that an operation is already in progress.
WSAENOTSOCK	Indicates that the resource is not a socket.

WSAEDESTADDRREQ	Indicates that a destination address is required.
WSAEMSGSIZE	Indicates that the message size is too long.
WSAEPROTOTYPE	Indicates the wrong protocol type for the socket.
WSAENOPROTOOPT	Indicates a bad protocol option.
WSAEPROTONOSUPPORT	Indicates that the protocol is not supported.
WSAESOCKTNOSUPPORT	Indicates that the socket type is not supported.
WSAEOPNOTSUPP	Indicates that the operation is not supported.
WSAEPFNOSUPPORT	Indicates that the protocol family is not supported.
WSAEAFNOSUPPORT	Indicates that the address family is not supported.
WSAEADDRINUSE	Indicates that the network address is already in use.
WSAEADDRNOTAVAIL	Indicates that the network address is not available.
WSAENETDOWN	Indicates that the network is down.
WSAENETUNREACH	Indicates that the network is unreachable.
WSAENETRESET	Indicates that the network connection has been reset.
WSAECONNABORTED	Indicates that the connection has been aborted.
WSAECONNRESET	Indicates that the connection has been reset by the peer.
WSAENOBUFS	Indicates that there is no buffer space available.
WSAEISCONN	Indicates that the socket is already connected.
WSAENOTCONN	Indicates that the socket is not connected.
WSAESHUTDOWN	Indicates that data cannot be sent after the socket has been shutdown.
WSAETOOMANYREFS	Indicates that there are too many references.
WSAETIMEDOUT	Indicates that the connection has timed out.
WSAECONNREFUSED	Indicates that the connection has been refused.
WSAELOOP	Indicates that a name cannot be translated.
WSAENAMETOOLONG	Indicates that a name was too long.
WSAEHOSTDOWN	Indicates that a network host is down.
WSAEHOSTUNREACH	Indicates that there is no route to a network host.
WSAENOTEMPTY	Indicates that the directory is not empty.
WSAEPROCLIM	Indicates that there are too many processes.
WSAEUSERS	Indicates that the user quota has been exceeded.
WSAEDQUOT	Indicates that the disk quota has been exceeded.
WSAESTALE	Indicates a stale file handle reference.

WSAEREMOTE	Indicates that the item is remote.
WSASYSNOTREADY	Indicates that the network subsystem is not ready.
WSAVERNOTSUPPORTED	Indicates that the winsock.dll version is out of range.
WSANOTINITIALISED	Indicates that successful WSASStartup has not yet been performed.
WSAEDISCON	Indicates that a graceful shutdown is in progress.
WSAENOMORE	Indicates that there are no more results.
WSAECANCELLED	Indicates that an operation has been canceled.
WSAEINVALIDPROCTABLE	Indicates that the procedure call table is invalid.
WSAEINVALIDPROVIDER	Indicates an invalid service provider.
WSAEPROVIDERFAILEDINIT	Indicates that the service provider failed to initialize.
WSASYSCALLFAILURE	Indicates a system call failure.
WSASERVICE_NOT_FOUND	Indicates that a service was not found.
WSATYPE_NOT_FOUND	Indicates that a class type was not found.
WSA_E_NO_MORE	Indicates that there are no more results.
WSA_E_CANCELLED	Indicates that the call was canceled.
WSAEREFUSED	Indicates that a database query was refused.

dlopen constants

If available on the operating system, the following constants are exported in `os.constants.dlopen`. See `dlopen(3)` for detailed information.

Constant	Description
RTLD_LAZY	Perform lazy binding. Node.js sets this flag by default.
RTLD_NOW	Resolve all undefined symbols in the library before dlopen(3) returns.
RTLD_GLOBAL	Symbols defined by the library will be made available for symbol resolution of subsequently loaded libraries.
RTLD_LOCAL	The converse of RTLD_GLOBAL . This is the default behavior if neither flag is specified.
RTLD_DEEPBIND	Make a self-contained library use its own symbols in preference to symbols from previously loaded libraries.

Priority constants

The following process scheduling constants are exported by `os.constants.priority`.

Constant	Description
PRIORITY_LOW	The lowest process scheduling priority. This corresponds to IDLE_PRIORITY_CLASS on Windows, and a nice value of 19 on all other platforms.
PRIORITY_BE	The process scheduling priority above PRIORITY_LOW and below PRIORITY_NORMAL . This corresponds to

LOW_NORMAL	BELOW_NORMAL_PRIORITY_CLASS on Windows, and a nice value of 10 on all other platforms.
PRIORITY_NORMAL	The default process scheduling priority. This corresponds to NORMAL_PRIORITY_CLASS on Windows, and a nice value of 0 on all other platforms.
PRIORITY_ABOVE_NORMAL	The process scheduling priority above PRIORITY_NORMAL and below PRIORITY_HIGH . This corresponds to ABOVE_NORMAL_PRIORITY_CLASS on Windows, and a nice value of -7 on all other platforms.
PRIORITY_HIGHEST	The process scheduling priority above PRIORITY_ABOVE_NORMAL and below PRIORITY_HIGHEST . This corresponds to HIGH_PRIORITY_CLASS on Windows, and a nice value of -14 on all other platforms.
PRIORITY_HIGHEST	The highest process scheduling priority. This corresponds to REALTIME_PRIORITY_CLASS on Windows, and a nice value of -20 on all other platforms.

libuv constants

Constant	Description
UV_UDP_REUSEADDR	

Path

Stability: 2 - Stable

Source Code: [lib/path.js](#)

The `path` module provides utilities for working with file and directory paths. It can be accessed using:

```
const path = require('path');
```

Windows vs. POSIX

The default operation of the `path` module varies based on the operating system on which a Node.js application is running. Specifically, when running on a Windows operating system, the `path` module will assume that Windows-style paths are being used.

So using `path.basename()` might yield different results on POSIX and Windows:

On POSIX:

```
path.basename('C:\\temp\\myfile.html');
// Returns: 'C:\\temp\\myfile.html'
```

On Windows:

```
path.basename('C:\\temp\\myfile.html');
// Returns: 'myfile.html'
```

To achieve consistent results when working with Windows file paths on any operating system, use `path.win32` :

On POSIX and Windows:

```
path.win32.basename('C:\\temp\\myfile.html');  
// Returns: 'myfile.html'
```

To achieve consistent results when working with POSIX file paths on any operating system, use `path.posix` :

On POSIX and Windows:

```
path.posix.basename('/tmp/myfile.html');  
// Returns: 'myfile.html'
```

On Windows Node.js follows the concept of per-drive working directory. This behavior can be observed when using a drive path without a backslash. For example, `path.resolve('C:\\')` can potentially return a different result than `path.resolve('C:')` . For more information, see [this MSDN page](#) .

path.basename(path[, ext])

- `path` `<string>`
- `ext` `<string>` An optional file extension
- Returns: `<string>`

The `path.basename()` method returns the last portion of a `path` , similar to the Unix `basename` command. Trailing directory separators are ignored, see [path.sep](#) .

```
path.basename('/foo/bar/baz/asdf/quux.html');  
// Returns: 'quux.html'  
  
path.basename('/foo/bar/baz/asdf/quux.html', '.html');  
// Returns: 'quux'
```

Although Windows usually treats file names, including file extensions, in a case-insensitive manner, this function does not. For example, `C:\\foo.html` and `C:\\foo.HTML` refer to the same file, but `basename` treats the extension as a case-sensitive string:

```
path.win32.basename('C:\\foo.html', '.html');  
// Returns: 'foo'  
  
path.win32.basename('C:\\foo.HTML', '.html');  
// Returns: 'foo.HTML'
```

A `TypeError` is thrown if `path` is not a string or if `ext` is given and is not a string.

path.delimiter

- `<string>`

Provides the platform-specific path delimiter:

- `;` for Windows

- `:` for POSIX

For example, on POSIX:

```
console.log(process.env.PATH);  
// Prints: '/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin'  
  
process.env.PATH.split(path.delimiter);  
// Returns: ['/usr/bin', '/bin', '/usr/sbin', '/sbin', '/usr/local/bin']
```

On Windows:

```
console.log(process.env.PATH);  
// Prints: 'C:\Windows\system32;C:\Windows;C:\Program Files\node\  
  
process.env.PATH.split(path.delimiter);  
// Returns ['C:\\Windows\\system32', 'C:\\Windows', 'C:\\Program Files\\node\\']
```

path.dirname(path)

- `path` `<string>`
- Returns: `<string>`

The `path.dirname()` method returns the directory name of a `path`, similar to the Unix `dirname` command. Trailing directory separators are ignored, see `path.sep`.

```
path.dirname('/foo/bar/baz/asdf/quux');  
// Returns: '/foo/bar/baz/asdf'
```

A `TypeError` is thrown if `path` is not a string.

path.extname(path)

- `path` `<string>`
- Returns: `<string>`

The `path.extname()` method returns the extension of the `path`, from the last occurrence of the `.` (period) character to end of string in the last portion of the `path`. If there is no `.` in the last portion of the `path`, or if there are no `.` characters other than the first character of the basename of `path` (see `path.basename()`), an empty string is returned.

```
path.extname('index.html');  
// Returns: '.html'  
  
path.extname('index.coffee.md');  
// Returns: '.md'  
  
path.extname('index. ');  
// Returns: ''
```

```

path.extname('index');
// Returns: ''

path.extname('.index');
// Returns: ''

path.extname('.index.md');
// Returns: '.md'

```

A `TypeError` is thrown if `path` is not a string.

path.format(pathObject)

- `pathObject` `<Object>` Any JavaScript object having the following properties:
 - `dir` `<string>`
 - `root` `<string>`
 - `base` `<string>`
 - `name` `<string>`
 - `ext` `<string>`
- Returns: `<string>`

The `path.format()` method returns a path string from an object. This is the opposite of `path.parse()`.

When providing properties to the `pathObject` remember that there are combinations where one property has priority over another:

- `pathObject.root` is ignored if `pathObject.dir` is provided
- `pathObject.ext` and `pathObject.name` are ignored if `pathObject.base` exists

For example, on POSIX:

```

// If `dir`, `root` and `base` are provided,
// `${dir}${path.sep}${base}`
// will be returned. `root` is ignored.
path.format({
  root: '/ignored',
  dir: '/home/user/dir',
  base: 'file.txt'
});
// Returns: '/home/user/dir/file.txt'

// `root` will be used if `dir` is not specified.
// If only `root` is provided or `dir` is equal to `root` then the
// platform separator will not be included. `ext` will be ignored.
path.format({
  root: '/',
  base: 'file.txt',
  ext: 'ignored'
});
// Returns: '/file.txt'

```

```
// `name` + `ext` will be used if `base` is not specified.
path.format({
  root: '/',
  name: 'file',
  ext: '.txt'
});
// Returns: '/file.txt'
```

On Windows:

```
path.format({
  dir: 'C:\\path\\dir',
  base: 'file.txt'
});
// Returns: 'C:\\path\\dir\\file.txt'
```

path.isAbsolute(path)

- `path` `<string>`
- Returns: `<boolean>`

The `path.isAbsolute()` method determines if `path` is an absolute path.

If the given `path` is a zero-length string, `false` will be returned.

For example, on POSIX:

```
path.isAbsolute('/foo/bar'); // true
path.isAbsolute('/baz/.'); // true
path.isAbsolute('qux/'); // false
path.isAbsolute('.'); // false
```

On Windows:

```
path.isAbsolute('//server'); // true
path.isAbsolute('\\\\server'); // true
path.isAbsolute('C:/foo/.'); // true
path.isAbsolute('C:\\foo\\.'); // true
path.isAbsolute('bar\\baz'); // false
path.isAbsolute('bar/baz'); // false
path.isAbsolute('.'); // false
```

A `TypeError` is thrown if `path` is not a string.

path.join([...paths])

- `...paths` `<string>` A sequence of path segments
- Returns: `<string>`

The `path.join()` method joins all given `path` segments together using the platform-specific separator as a delimiter, then normalizes the resulting path.

Zero-length `path` segments are ignored. If the joined path string is a zero-length string then `'.'` will be returned, representing the current working directory.

```
path.join('/foo', 'bar', 'baz/asdf', 'quux', '..');  
// Returns: '/foo/bar/baz/asdf'  
  
path.join('foo', {}, 'bar');  
// Throws 'TypeError: Path must be a string. Received {}'
```

A `TypeError` is thrown if any of the path segments is not a string.

path.normalize(path)

- `path` `<string>`
- Returns: `<string>`

The `path.normalize()` method normalizes the given `path`, resolving `'..'` and `'.'` segments.

When multiple, sequential path segment separation characters are found (e.g. `/` on POSIX and either `\` or `/` on Windows), they are replaced by a single instance of the platform-specific path segment separator (`/` on POSIX and `\` on Windows). Trailing separators are preserved.

If the `path` is a zero-length string, `'.'` is returned, representing the current working directory.

For example, on POSIX:

```
path.normalize('/foo/bar//baz/asdf/quux/..');  
// Returns: '/foo/bar/baz/asdf'
```

On Windows:

```
path.normalize('C:\\temp\\\\foo\\bar\\..\\');  
// Returns: 'C:\\temp\\foo\\'
```

Since Windows recognizes multiple path separators, both separators will be replaced by instances of the Windows preferred separator (`\`):

```
path.win32.normalize('C:///temp\\\\\\//\\//foo/bar');  
// Returns: 'C:\\temp\\foo\\bar'
```

A `TypeError` is thrown if `path` is not a string.

path.parse(path)

- `path` `<string>`
- Returns: `<Object>`

The `path.parse()` method returns an object whose properties represent significant elements of the `path`. Trailing directory separators are ignored, see `path.sep`.

The returned object will have the following properties:

- `dir` `<string>`
- `root` `<string>`
- `base` `<string>`
- `name` `<string>`
- `ext` `<string>`

For example, on POSIX:

```
path.parse('/home/user/dir/file.txt');  
// Returns:  
// { root: '/',  
//   dir: '/home/user/dir',  
//   base: 'file.txt',  
//   ext: '.txt',  
//   name: 'file' }
```

dir		base	
root		name	ext
" / home/user/dir / file .txt "			

(All spaces in the "" line should be ignored. They are purely for formatting.)

On Windows:

```
path.parse('C:\\path\\dir\\file.txt');  
// Returns:  
// { root: 'C:\\',  
//   dir: 'C:\\path\\dir',  
//   base: 'file.txt',  
//   ext: '.txt',  
//   name: 'file' }
```

dir		base	
root		name	ext
" C:\ path\dir \ file .txt "			

(All spaces in the "" line should be ignored. They are purely for formatting.)

A `TypeError` is thrown if `path` is not a string.

path.posix

- `<Object>`

The `path.posix` property provides access to POSIX specific implementations of the `path` methods.

The API is accessible via `require('path').posix` or `require('path/posix')`.

path.relative(from, to)

- `from` `<string>`
- `to` `<string>`
- Returns: `<string>`

The `path.relative()` method returns the relative path from `from` to `to` based on the current working directory. If `from` and `to` each resolve to the same path (after calling `path.resolve()` on each), a zero-length string is returned.

If a zero-length string is passed as `from` or `to`, the current working directory will be used instead of the zero-length strings.

For example, on POSIX:

```
path.relative('/data/orandea/test/aaa', '/data/orandea/impl/bbb');
// Returns: '../../../impl/bbb'
```

On Windows:

```
path.relative('C:\\orandea\\test\\aaa', 'C:\\orandea\\impl\\bbb');
// Returns: '..\\..\\impl\\bbb'
```

A `TypeError` is thrown if either `from` or `to` is not a string.

path.resolve([...paths])

- `...paths` `<string>` A sequence of paths or path segments
- Returns: `<string>`

The `path.resolve()` method resolves a sequence of paths or path segments into an absolute path.

The given sequence of paths is processed from right to left, with each subsequent `path` prepended until an absolute path is constructed. For instance, given the sequence of path segments: `/foo`, `/bar`, `baz`, calling `path.resolve('/foo', '/bar', 'baz')` would return `/bar/baz` because `'baz'` is not an absolute path but `'/bar' + '/' + 'baz'` is.

If, after processing all given `path` segments, an absolute path has not yet been generated, the current working directory is used.

The resulting path is normalized and trailing slashes are removed unless the path is resolved to the root directory.

Zero-length `path` segments are ignored.

If no `path` segments are passed, `path.resolve()` will return the absolute path of the current working directory.

```
path.resolve('/foo/bar', './baz');
// Returns: '/foo/bar/baz'

path.resolve('/foo/bar', '/tmp/file/');
```

```
// Returns: '/tmp/file'

path.resolve('wwwroot', 'static_files/png/', '../gif/image.gif');
// If the current working directory is /home/myself/node,
// this returns '/home/myself/node/wwwroot/static_files/gif/image.gif'
```

A `TypeError` is thrown if any of the arguments is not a string.

path.sep

- `<string>`

Provides the platform-specific path segment separator:

- `\` on Windows
- `/` on POSIX

For example, on POSIX:

```
'foo/bar/baz'.split(path.sep);
// Returns: ['foo', 'bar', 'baz']
```

On Windows:

```
'foo\\bar\\baz'.split(path.sep);
// Returns: ['foo', 'bar', 'baz']
```

On Windows, both the forward slash (`/`) and backward slash (`\`) are accepted as path segment separators; however, the `path` methods only add backward slashes (`\`).

path.toNamespacedPath(path)

- `path` `<string>`
- Returns: `<string>`

On Windows systems only, returns an equivalent `namespace-prefixed path` for the given `path`. If `path` is not a string, `path` will be returned without modifications.

This method is meaningful only on Windows systems. On POSIX systems, the method is non-operational and always returns `path` without modifications.

path.win32

- `<Object>`

The `path.win32` property provides access to Windows-specific implementations of the `path` methods.

The API is accessible via `require('path').win32` or `require('path/win32')`.

Performance measurement APIs

Source Code: [lib/perf_hooks.js](#)

This module provides an implementation of a subset of the W3C [Web Performance APIs](#) as well as additional APIs for Node.js-specific performance measurements.

Node.js supports the following [Web Performance APIs](#) :

- [High Resolution Time](#)
- [Performance Timeline](#)
- [User Timing](#)

```
const { PerformanceObserver, performance } = require('perf_hooks');

const obs = new PerformanceObserver((items) => {
  console.log(items.getEntries()[0].duration);
  performance.clearMarks();
});

obs.observe({ type: 'measure' });
performance.measure('Start to Now');

performance.mark('A');
doSomeLongRunningProcess(() => {
  performance.measure('A to Now', 'A');

  performance.mark('B');
  performance.measure('A to B', 'A', 'B');
});
```

perf_hooks.performance

An object that can be used to collect performance metrics from the current Node.js instance. It is similar to [window.performance](#) in browsers.

performance.clearMarks([name])

- `name` [<string>](#)

If `name` is not provided, removes all `PerformanceMark` objects from the Performance Timeline. If `name` is provided, removes only the named mark.

performance.eventLoopUtilization([utilization1[, utilization2]])

- `utilization1` [<Object>](#) The result of a previous call to `eventLoopUtilization()`.
- `utilization2` [<Object>](#) The result of a previous call to `eventLoopUtilization()` prior to `utilization1`.
- Returns [<Object>](#)
 - `idle` [<number>](#)
 - `active` [<number>](#)
 - `utilization` [<number>](#)

The `eventLoopUtilization()` method returns an object that contains the cumulative duration of time the event loop has been both idle and active as a high resolution milliseconds timer. The `utilization` value is the calculated Event Loop Utilization (ELU).

If bootstrapping has not yet finished on the main thread the properties have the value of `0`. The ELU is immediately available on `Worker threads` since bootstrap happens within the event loop.

Both `utilization1` and `utilization2` are optional parameters.

If `utilization1` is passed, then the delta between the current call's `active` and `idle` times, as well as the corresponding `utilization` value are calculated and returned (similar to `process.hrtime()`).

If `utilization1` and `utilization2` are both passed, then the delta is calculated between the two arguments. This is a convenience option because, unlike `process.hrtime()`, calculating the ELU is more complex than a single subtraction.

ELU is similar to CPU utilization, except that it only measures event loop statistics and not CPU usage. It represents the percentage of time the event loop has spent outside the event loop's event provider (e.g. `epoll_wait`). No other CPU idle time is taken into consideration. The following is an example of how a mostly idle process will have a high ELU.

```
'use strict';
const { eventLoopUtilization } = require('perf_hooks').performance;
const { spawnSync } = require('child_process');

setImmediate(() => {
  const elu = eventLoopUtilization();
  spawnSync('sleep', ['5']);
  console.log(eventLoopUtilization(elu).utilization);
});
```

Although the CPU is mostly idle while running this script, the value of `utilization` is `1`. This is because the call to `child_process.spawnSync()` blocks the event loop from proceeding.

Passing in a user-defined object instead of the result of a previous call to `eventLoopUtilization()` will lead to undefined behavior. The return values are not guaranteed to reflect any correct state of the event loop.

`performance.mark([name[, options]])`

- `name` `<string>`
- `options` `<Object>`
 - `detail` `<any>` Additional optional detail to include with the mark.
 - `startTime` `<number>` An optional timestamp to be used as the mark time. Defaults: `performance.now()`.

Creates a new `PerformanceMark` entry in the Performance Timeline. A `PerformanceMark` is a subclass of `PerformanceEntry` whose `performanceEntry.entryType` is always `'mark'`, and whose `performanceEntry.duration` is always `0`. Performance marks are used to mark specific significant moments in the Performance Timeline.

`performance.measure(name[, startMarkOrOptions[, endMark]])`

- `name` `<string>`
- `startMarkOrOptions` `<string> | <Object>` Optional.
 - `detail` `<any>` Additional optional detail to include with the measure.
 - `duration` `<number>` Duration between start and end times.
 - `end` `<number> | <string>` Timestamp to be used as the end time, or a string identifying a previously recorded mark.
 - `start` `<number> | <string>` Timestamp to be used as the start time, or a string identifying a previously recorded mark.

- `endMark` `<string>` Optional. Must be omitted if `startMarkOrOptions` is an `<Object>`.

Creates a new `PerformanceMeasure` entry in the Performance Timeline. A `PerformanceMeasure` is a subclass of `PerformanceEntry` whose `performanceEntry.entryType` is always `'measure'`, and whose `performanceEntry.duration` measures the number of milliseconds elapsed since `startMark` and `endMark`.

The `startMark` argument may identify any *existing* `PerformanceMark` in the Performance Timeline, or *may* identify any of the timestamp properties provided by the `PerformanceNodeTiming` class. If the named `startMark` does not exist, an error is thrown.

The optional `endMark` argument must identify any *existing* `PerformanceMark` in the Performance Timeline or any of the timestamp properties provided by the `PerformanceNodeTiming` class. `endMark` will be `performance.now()` if no parameter is passed, otherwise if the named `endMark` does not exist, an error will be thrown.

performance.nodeTiming

- `<PerformanceNodeTiming>`

This property is an extension by Node.js. It is not available in Web browsers.

An instance of the `PerformanceNodeTiming` class that provides performance metrics for specific Node.js operational milestones.

performance.now()

- Returns: `<number>`

Returns the current high resolution millisecond timestamp, where 0 represents the start of the current `node` process.

performance.timeOrigin

- `<number>`

The `timeOrigin` specifies the high resolution millisecond timestamp at which the current `node` process began, measured in Unix time.

performance.timerify(fn[, options])

- `fn` `<Function>`
- `options` `<Object>`
 - `histogram` `<RecordableHistogram>` A histogram object created using `perf_hooks.createHistogram()` that will record runtime durations in nanoseconds.

This property is an extension by Node.js. It is not available in Web browsers.

Wraps a function within a new function that measures the running time of the wrapped function. A `PerformanceObserver` must be subscribed to the `'function'` event type in order for the timing details to be accessed.

```
const {
  performance,
  PerformanceObserver
} = require('perf_hooks');

function someFunction() {
  console.log('hello world');
}

const wrapped = performance.timerify(someFunction);

const obs = new PerformanceObserver((list) => {
```

```

    console.log(list.getEntries()[0].duration);
    obs.disconnect();
  });
  obs.observe({ entryTypes: ['function'] });

  // A performance timeline entry will be created
  wrapped();

```

If the wrapped function returns a promise, a finally handler will be attached to the promise and the duration will be reported once the finally handler is invoked.

performance.toJSON()

An object which is JSON representation of the `performance` object. It is similar to `window.performance.toJSON` in browsers.

Class: PerformanceEntry

performanceEntry.details

- `<any>`

Additional detail specific to the `entryType`.

performanceEntry.duration

- `<number>`

The total number of milliseconds elapsed for this entry. This value will not be meaningful for all Performance Entry types.

performanceEntry.entryType

- `<string>`

The type of the performance entry. It may be one of:

- `'node'` (Node.js only)
- `'mark'` (available on the Web)
- `'measure'` (available on the Web)
- `'gc'` (Node.js only)
- `'function'` (Node.js only)
- `'http2'` (Node.js only)
- `'http'` (Node.js only)

performanceEntry.flags

- `<number>`

This property is an extension by Node.js. It is not available in Web browsers.

When `performanceEntry.entryType` is equal to `'gc'`, the `performance.flags` property contains additional information about garbage collection operation. The value may be one of:

- `perf_hooks.constants.NODE_PERFORMANCE_GC_FLAGS_NO`
- `perf_hooks.constants.NODE_PERFORMANCE_GC_FLAGS_CONSTRUCT_RETAINED`

- `perf_hooks.constants.NODE_PERFORMANCE_GC_FLAGS_FORCED`
- `perf_hooks.constants.NODE_PERFORMANCE_GC_FLAGS_SYNCHRONOUS_PHANTOM_PROCESSING`
- `perf_hooks.constants.NODE_PERFORMANCE_GC_FLAGS_ALL_AVAILABLE_GARBAGE`
- `perf_hooks.constants.NODE_PERFORMANCE_GC_FLAGS_ALL_EXTERNAL_MEMORY`
- `perf_hooks.constants.NODE_PERFORMANCE_GC_FLAGS_SCHEDULE_IDLE`

performanceEntry.name

- `<string>`

The name of the performance entry.

performanceEntry.kind

- `<number>`

This property is an extension by Node.js. It is not available in Web browsers.

When `performanceEntry.entryType` is equal to `'gc'`, the `performanceEntry.kind` property identifies the type of garbage collection operation that occurred. The value may be one of:

- `perf_hooks.constants.NODE_PERFORMANCE_GC_MAJOR`
- `perf_hooks.constants.NODE_PERFORMANCE_GC_MINOR`
- `perf_hooks.constants.NODE_PERFORMANCE_GC_INCREMENTAL`
- `perf_hooks.constants.NODE_PERFORMANCE_GC_WEAKCB`

performanceEntry.startTime

- `<number>`

The high resolution millisecond timestamp marking the starting time of the Performance Entry.

Garbage Collection ('gc') Details

When `performanceEntry.type` is equal to `'gc'`, the `performanceEntry.details` property will be an `<Object>` with two properties:

- `kind` `<number>` One of:
 - `perf_hooks.constants.NODE_PERFORMANCE_GC_MAJOR`
 - `perf_hooks.constants.NODE_PERFORMANCE_GC_MINOR`
 - `perf_hooks.constants.NODE_PERFORMANCE_GC_INCREMENTAL`
 - `perf_hooks.constants.NODE_PERFORMANCE_GC_WEAKCB`
- `flags` `<number>` One of:
 - `perf_hooks.constants.NODE_PERFORMANCE_GC_FLAGS_NO`
 - `perf_hooks.constants.NODE_PERFORMANCE_GC_FLAGS_CONSTRUCT_RETAINED`
 - `perf_hooks.constants.NODE_PERFORMANCE_GC_FLAGS_FORCED`
 - `perf_hooks.constants.NODE_PERFORMANCE_GC_FLAGS_SYNCHRONOUS_PHANTOM_PROCESSING`
 - `perf_hooks.constants.NODE_PERFORMANCE_GC_FLAGS_ALL_AVAILABLE_GARBAGE`
 - `perf_hooks.constants.NODE_PERFORMANCE_GC_FLAGS_ALL_EXTERNAL_MEMORY`
 - `perf_hooks.constants.NODE_PERFORMANCE_GC_FLAGS_SCHEDULE_IDLE`

HTTP/2 ('http2') Details

When `performanceEntry.type` is equal to `'http2'`, the `performanceEntry.details` property will be an `<Object>` containing additional performance information.

If `performanceEntry.name` is equal to `Http2Stream`, the `details` will contain the following properties:

- `bytesRead` `<number>` The number of `DATA` frame bytes received for this `Http2Stream`.
- `bytesWritten` `<number>` The number of `DATA` frame bytes sent for this `Http2Stream`.
- `id` `<number>` The identifier of the associated `Http2Stream`.
- `timeToFirstByte` `<number>` The number of milliseconds elapsed between the `PerformanceEntry.startTime` and the reception of the first `DATA` frame.
- `timeToFirstByteSent` `<number>` The number of milliseconds elapsed between the `PerformanceEntry.startTime` and sending of the first `DATA` frame.
- `timeToFirstHeader` `<number>` The number of milliseconds elapsed between the `PerformanceEntry.startTime` and the reception of the first header.

If `performanceEntry.name` is equal to `Http2Session`, the `details` will contain the following properties:

- `bytesRead` `<number>` The number of bytes received for this `Http2Session`.
- `bytesWritten` `<number>` The number of bytes sent for this `Http2Session`.
- `framesReceived` `<number>` The number of HTTP/2 frames received by the `Http2Session`.
- `framesSent` `<number>` The number of HTTP/2 frames sent by the `Http2Session`.
- `maxConcurrentStreams` `<number>` The maximum number of streams concurrently open during the lifetime of the `Http2Session`.
- `pingRTT` `<number>` The number of milliseconds elapsed since the transmission of a `PING` frame and the reception of its acknowledgment. Only present if a `PING` frame has been sent on the `Http2Session`.
- `streamAverageDuration` `<number>` The average duration (in milliseconds) for all `Http2Stream` instances.
- `streamCount` `<number>` The number of `Http2Stream` instances processed by the `Http2Session`.
- `type` `<string>` Either `'server'` or `'client'` to identify the type of `Http2Session`.

Timerify ('function') Details

When `performanceEntry.type` is equal to `'function'`, the `performanceEntry.details` property will be an `<Array>` listing the input arguments to the timed function.

Class: PerformanceNodeTiming

- Extends: `<PerformanceEntry>`

This property is an extension by Node.js. It is not available in Web browsers.

Provides timing details for Node.js itself. The constructor of this class is not exposed to users.

performanceNodeTiming.bootstrapComplete

- `<number>`

The high resolution millisecond timestamp at which the Node.js process completed bootstrapping. If bootstrapping has not yet finished, the property has the value of -1.

performanceNodeTiming.environment

- `<number>`

The high resolution millisecond timestamp at which the Node.js environment was initialized.

performanceNodeTiming.idleTime

- `<number>`

The high resolution millisecond timestamp of the amount of time the event loop has been idle within the event loop's event provider (e.g. `epoll_wait`). This does not take CPU usage into consideration. If the event loop has not yet started (e.g., in the first tick of the main script), the property has the value of 0.

performanceNodeTiming.loopExit

- `<number>`

The high resolution millisecond timestamp at which the Node.js event loop exited. If the event loop has not yet exited, the property has the value of -1. It can only have a value of not -1 in a handler of the `'exit'` event.

performanceNodeTiming.loopStart

- `<number>`

The high resolution millisecond timestamp at which the Node.js event loop started. If the event loop has not yet started (e.g., in the first tick of the main script), the property has the value of -1.

performanceNodeTiming.nodeStart

- `<number>`

The high resolution millisecond timestamp at which the Node.js process was initialized.

performanceNodeTiming.v8Start

- `<number>`

The high resolution millisecond timestamp at which the V8 platform was initialized.

Class: perf_hooks.PerformanceObserver

new PerformanceObserver(callback)

- `callback` `<Function>`
 - `list` `<PerformanceObserverEntryList>`
 - `observer` `<PerformanceObserver>`

`PerformanceObserver` objects provide notifications when new `PerformanceEntry` instances have been added to the Performance Timeline.

```
const {
  performance,
  PerformanceObserver
} = require('perf_hooks');

const obs = new PerformanceObserver((list, observer) => {
  console.log(list.getEntries());
  observer.disconnect();
});

obs.observe({ entryTypes: ['mark'], buffered: true });

performance.mark('test');
```

Because `PerformanceObserver` instances introduce their own additional performance overhead, instances should not be left subscribed to notifications indefinitely. Users should disconnect observers as soon as they are no longer needed.

The `callback` is invoked when a `PerformanceObserver` is notified about new `PerformanceEntry` instances. The callback receives a `PerformanceObserverEntryList` instance and a reference to the `PerformanceObserver`.

`performanceObserver.disconnect()`

Disconnects the `PerformanceObserver` instance from all notifications.

`performanceObserver.observe(options)`

- `options` `<Object>`
 - `type` `<string>` A single `<PerformanceEntry>` type. Must not be given if `entryTypes` is already specified.
 - `entryTypes` `<string[]>` An array of strings identifying the types of `<PerformanceEntry>` instances the observer is interested in. If not provided an error will be thrown.
 - `buffered` `<boolean>` If true, the observer callback is called with a list global `PerformanceEntry` buffered entries. If false, only `PerformanceEntry`s created after the time point are sent to the observer callback. **Default:** `false`.

Subscribes the `<PerformanceObserver>` instance to notifications of new `<PerformanceEntry>` instances identified either by `options.entryTypes` or `options.type`:

```
const {
  performance,
  PerformanceObserver
} = require('perf_hooks');

const obs = new PerformanceObserver((list, observer) => {
  // Called three times synchronously. `list` contains one item.
});
obs.observe({ type: 'mark' });

for (let n = 0; n < 3; n++)
  performance.mark(`test${n}`);
```

Class: `PerformanceObserverEntryList`

The `PerformanceObserverEntryList` class is used to provide access to the `PerformanceEntry` instances passed to a `PerformanceObserver`. The constructor of this class is not exposed to users.

`performanceObserverEntryList.getEntries()`

- Returns: `<PerformanceEntry[]>`

Returns a list of `PerformanceEntry` objects in chronological order with respect to `performanceEntry.startTime`.

```
const {
  performance,
  PerformanceObserver
} = require('perf_hooks');
```

```

const obs = new PerformanceObserver((perfObserverList, observer) => {
  console.log(perfObserverList.getEntries());
  /**
   * [
   *   PerformanceEntry {
   *     name: 'test',
   *     entryType: 'mark',
   *     startTime: 81.465639,
   *     duration: 0
   *   },
   *   PerformanceEntry {
   *     name: 'meow',
   *     entryType: 'mark',
   *     startTime: 81.860064,
   *     duration: 0
   *   }
   * ]
   */
  observer.disconnect();
});
obs.observe({ type: 'mark' });

performance.mark('test');
performance.mark('meow');

```

performanceObserverEntryList.getEntriesByName(name[, type])

- `name` <string>
- `type` <string>
- Returns: <PerformanceEntry[]>

Returns a list of `PerformanceEntry` objects in chronological order with respect to `performanceEntry.startTime` whose `performanceEntry.name` is equal to `name`, and optionally, whose `performanceEntry.entryType` is equal to `type`.

```

const {
  performance,
  PerformanceObserver
} = require('perf_hooks');

const obs = new PerformanceObserver((perfObserverList, observer) => {
  console.log(perfObserverList.getEntriesByName('meow'));
  /**
   * [
   *   PerformanceEntry {
   *     name: 'meow',
   *     entryType: 'mark',
   *     startTime: 98.545991,
   *     duration: 0
   *   }
   * ]
   */
});

```

```

console.log(perfObserverList.getEntriesByName('nope')); // []

console.log(perfObserverList.getEntriesByName('test', 'mark'));
/**
 * [
 *   PerformanceEntry {
 *     name: 'test',
 *     entryType: 'mark',
 *     startTime: 63.518931,
 *     duration: 0
 *   }
 * ]
 */
console.log(perfObserverList.getEntriesByName('test', 'measure')); // []
observer.disconnect();
});
obs.observe({ entryTypes: ['mark', 'measure'] });

performance.mark('test');
performance.mark('meow');

```

performanceObserverEntryList.getEntriesByType(type)

- type `<string>`
- Returns: `<PerformanceEntry[]>`

Returns a list of `PerformanceEntry` objects in chronological order with respect to `performanceEntry.startTime` whose `performanceEntry.entryType` is equal to `type`.

```

const {
  performance,
  PerformanceObserver
} = require('perf_hooks');

const obs = new PerformanceObserver((perfObserverList, observer) => {
  console.log(perfObserverList.getEntriesByType('mark'));
  /**
   * [
   *   PerformanceEntry {
   *     name: 'test',
   *     entryType: 'mark',
   *     startTime: 55.897834,
   *     duration: 0
   *   },
   *   PerformanceEntry {
   *     name: 'meow',
   *     entryType: 'mark',
   *     startTime: 56.350146,
   *     duration: 0
   *   }
   * ]
   */
});

```

```

    */
    observer.disconnect();
  });
  obs.observe({ type: 'mark' });

  performance.mark('test');
  performance.mark('meow');

```

perf_hooks.createHistogram([options])

- options `<Object>`
 - min `<number>` | `<bigint>` The minimum recordable value. Must be an integer value greater than 0. **Default:** 1.
 - max `<number>` | `<bigint>` The maximum recordable value. Must be an integer value greater than min. **Default:** `Number.MAX_SAFE_INTEGER`.
 - figures `<number>` The number of accuracy digits. Must be a number between 1 and 5. **Default:** 3.
- Returns `<RecordableHistogram>`

Returns a `<RecordableHistogram>`.

perf_hooks.monitorEventLoopDelay([options])

- options `<Object>`
 - resolution `<number>` The sampling rate in milliseconds. Must be greater than zero. **Default:** 10.
- Returns: `<IntervalHistogram>`

This property is an extension by Node.js. It is not available in Web browsers.

Creates an `IntervalHistogram` object that samples and reports the event loop delay over time. The delays will be reported in nanoseconds.

Using a timer to detect approximate event loop delay works because the execution of timers is tied specifically to the lifecycle of the libuv event loop. That is, a delay in the loop will cause a delay in the execution of the timer, and those delays are specifically what this API is intended to detect.

```

const { monitorEventLoopDelay } = require('perf_hooks');
const h = monitorEventLoopDelay({ resolution: 20 });
h.enable();
// Do something.
h.disable();
console.log(h.min);
console.log(h.max);
console.log(h.mean);
console.log(h.stddev);
console.log(h.percentiles);
console.log(h.percentile(50));
console.log(h.percentile(99));

```

Class: Histogram

histogram.exceeds

- `<number>`

The number of times the event loop delay exceeded the maximum 1 hour event loop delay threshold.

histogram.max

- `<number>`

The maximum recorded event loop delay.

histogram.mean

- `<number>`

The mean of the recorded event loop delays.

histogram.min

- `<number>`

The minimum recorded event loop delay.

histogram.percentile(percentile)

- `percentile` `<number>` A percentile value in the range (0, 100].
- Returns: `<number>`

Returns the value at the given percentile.

histogram.percentiles

- `<Map>`

Returns a `Map` object detailing the accumulated percentile distribution.

histogram.reset()

Resets the collected histogram data.

histogram.stddev

- `<number>`

The standard deviation of the recorded event loop delays.

Class: IntervalHistogram extends Histogram

A `Histogram` that is periodically updated on a given interval.

histogram.disable()

- Returns: `<boolean>`

Disables the update interval timer. Returns `true` if the timer was stopped, `false` if it was already stopped.

histogram.enable()

- Returns: `<boolean>`

Enables the update interval timer. Returns `true` if the timer was started, `false` if it was already started.

Cloning an IntervalHistogram

`<IntervalHistogram>` instances can be cloned via `<MessagePort>`. On the receiving end, the histogram is cloned as a plain `<Histogram>` object that does not implement the `enable()` and `disable()` methods.

Class: RecordableHistogram extends Histogram

histogram.record(val)

- `val <number> | <bigint>` The amount to record in the histogram.

histogram.recordDelta()

Calculates the amount of time (in nanoseconds) that has passed since the previous call to `recordDelta()` and records that amount in the histogram.

Examples

Measuring the duration of async operations

The following example uses the `Async Hooks` and Performance APIs to measure the actual duration of a Timeout operation (including the amount of time it took to execute the callback).

```
'use strict';
const async_hooks = require('async_hooks');
const {
  performance,
  PerformanceObserver
} = require('perf_hooks');

const set = new Set();
const hook = async_hooks.createHook({
  init(id, type) {
    if (type === 'Timeout') {
      performance.mark(`Timeout-${id}-Init`);
      set.add(id);
    }
  },
  destroy(id) {
    if (set.has(id)) {
      set.delete(id);
      performance.mark(`Timeout-${id}-Destroy`);
      performance.measure(`Timeout-${id}`,
        `Timeout-${id}-Init`,
        `Timeout-${id}-Destroy`);
    }
  }
});
hook.enable();

const obs = new PerformanceObserver((list, observer) => {
```

```

    console.log(list.getEntries()[0]);
    performance.clearMarks();
    observer.disconnect();
  });
  obs.observe({ entryTypes: ['measure'], buffered: true });

  setTimeout(() => {}, 1000);

```

Measuring how long it takes to load dependencies

The following example measures the duration of `require()` operations to load dependencies:

```

'use strict';
const {
  performance,
  PerformanceObserver
} = require('perf_hooks');
const mod = require('module');

// Monkey patch the require function
mod.Module.prototype.require =
  performance.timerify(mod.Module.prototype.require);
require = performance.timerify(require);

// Activate the observer
const obs = new PerformanceObserver((list) => {
  const entries = list.getEntries();
  entries.forEach((entry) => {
    console.log(`require('${entry[0]}')`, entry.duration);
  });
  obs.disconnect();
});
obs.observe({ entryTypes: ['function'], buffered: true });

require('some-module');

```

Policies

Stability: 1 - Experimental

Node.js contains experimental support for creating policies on loading code.

Policies are a security feature intended to allow guarantees about what code Node.js is able to load. The use of policies assumes safe practices for the policy files such as ensuring that policy files cannot be overwritten by the Node.js application by using file permissions.

A best practice would be to ensure that the policy manifest is read-only for the running Node.js application and that the file cannot be changed by the running Node.js application in any way. A typical setup would be to create the policy file as a different user id than the one running Node.js and granting read permissions to the user id running Node.js.

Enabling

The `--experimental-policy` flag can be used to enable features for policies when loading modules.

Once this has been set, all modules must conform to a policy manifest file passed to the flag:

```
node --experimental-policy=policy.json app.js
```

The policy manifest will be used to enforce constraints on code loaded by Node.js.

To mitigate tampering with policy files on disk, an integrity for the policy file itself may be provided via `--policy-integrity`. This allows running `node` and asserting the policy file contents even if the file is changed on disk.

```
node --experimental-policy=policy.json --policy-integrity="sha384-SggXRQHwCG8g+DktYYzxkXRIkTiEYWBHqev0xnpCxYlqMBufKZHAHQM3/boDaI/0"
```

Features

Error behavior

When a policy check fails, Node.js by default will throw an error. It is possible to change the error behavior to one of a few possibilities by defining an "onerror" field in a policy manifest. The following values are available to change the behavior:

- `"exit"`: will exit the process immediately. No cleanup code will be allowed to run.
- `"log"`: will log the error at the site of the failure.
- `"throw"`: will throw a JS error at the site of the failure. This is the default.

```
{
  "onerror": "log",
  "resources": {
    "./app/checked.js": {
      "integrity": "sha384-SggXRQHwCG8g+DktYYzxkXRIkTiEYWBHqev0xnpCxYlqMBufKZHAHQM3/boDaI/0"
    }
  }
}
```

Integrity checks

Policy files must use integrity checks with Subresource Integrity strings compatible with the browser `integrity attribute` associated with absolute URLs.

When using `require()` all resources involved in loading are checked for integrity if a policy manifest has been specified. If a resource does not match the integrity listed in the manifest, an error will be thrown.

An example policy file that would allow loading a file `checked.js`:

```
{
  "resources": {
    "./app/checked.js": {
      "integrity": "sha384-SggXRQHwCG8g+DktYYzxkXRIkTiEYWBHqev0xnpCxYlqMBufKZHAHQM3/boDaI/0"
    }
  }
}
```

```
}  
}  
}
```

Each resource listed in the policy manifest can be of one the following formats to determine its location:

1. A **relative-URL string** to a resource from the manifest such as `./resource.js`, `../resource.js`, or `/resource.js`.
2. A complete URL string to a resource such as `file:///resource.js`.

When loading resources the entire URL must match including search parameters and hash fragment. `./a.js?b` will not be used when attempting to load `./a.js` and vice versa.

To generate integrity strings, a script such as `printf "sha384-$(cat checked.js | openssl dgst -sha384 -binary | base64)"` can be used.

Integrity can be specified as the boolean value `true` to accept any body for the resource which can be useful for local development. It is not recommended in production since it would allow unexpected alteration of resources to be considered valid.

Dependency redirection

An application may need to ship patched versions of modules or to prevent modules from allowing all modules access to all other modules. Redirection can be used by intercepting attempts to load the modules wishing to be replaced.

```
{  
  "resources": {  
    "./app/checked.js": {  
      "dependencies": {  
        "fs": true,  
        "os": "./app/node_modules/alt-os",  
        "http": { "import": true }  
      }  
    }  
  }  
}
```

The dependencies are keyed by the requested specifier string and have values of either `true`, `null`, a string pointing to a module to be resolved, or a conditions object.

The specifier string does not perform any searching and must match exactly what is provided to the `require()` or `import`. Therefore, multiple specifiers may be needed in the policy if it uses multiple different strings to point to the same module (such as excluding the extension).

If the value of the redirection is `true` the default searching algorithms are used to find the module.

If the value of the redirection is a string, it is resolved relative to the manifest and then immediately used without searching.

Any specifier string for which resolution is attempted and that is not listed in the dependencies results in an error according to the policy.

Redirection does not prevent access to APIs through means such as direct access to `require.cache` or through `module.constructor` which allow access to loading modules. Policy redirection only affects specifiers to `require()` and `import`. Other means, such as to prevent undesired access to APIs through variables, are necessary to lock down that path of loading modules.

A boolean value of `true` for the dependencies map can be specified to allow a module to load any specifier without redirection. This can be useful for local development and may have some valid usage in production, but should be used only with care after auditing a module to ensure

its behavior is valid.

Similar to "exports" in `package.json`, dependencies can also be specified to be objects containing conditions which branch how dependencies are loaded. In the preceding example, "http" is allowed when the "import" condition is part of loading it.

A value of `null` for the resolved value causes the resolution to fail. This can be used to ensure some kinds of dynamic access are explicitly prevented.

Unknown values for the resolved module location cause failures but are not guaranteed to be forward compatible.

Example: Patched dependency

Redirected dependencies can provide attenuated or modified functionality as fits the application. For example, log data about timing of function durations by wrapping the original:

```
const original = require('fn');
module.exports = function fn(...args) {
  console.time();
  try {
    return new.target ?
      Reflect.construct(original, args) :
      Reflect.apply(original, this, args);
  } finally {
    console.timeEnd();
  }
};
```

Scopes

Use the "scopes" field of a manifest to set configuration for many resources at once. The "scopes" field works by matching resources by their segments. If a scope or resource includes "cascade": `true`, unknown specifiers will be searched for in their containing scope. The containing scope for cascading is found by recursively reducing the resource URL by removing segments for `special schemes`, keeping trailing `"/"` suffixes, and removing the query and hash fragment. This leads to the eventual reduction of the URL to its origin. If the URL is non-special the scope will be located by the URL's origin. If no scope is found for the origin or in the case of opaque origins, a protocol string can be used as a scope.

Note, `blob:` URLs adopt their origin from the path they contain, and so a scope of `"blob:https://nodejs.org"` will have no effect since no URL can have an origin of `blob:https://nodejs.org`; URLs starting with `blob:https://nodejs.org/` will use `https://nodejs.org` for its origin and thus `https:` for its protocol scope. For opaque origin `blob:` URLs they will have `blob:` for their protocol scope since they do not adopt origins.

Integrity using scopes

Setting an integrity to `true` on a scope will set the integrity for any resource not found in the manifest to `true`.

Setting an integrity to `null` on a scope will set the integrity for any resource not found in the manifest to fail matching.

Not including an integrity is the same as setting the integrity to `null`.

"cascade" for integrity checks will be ignored if "integrity" is explicitly set.

The following example allows loading any file:

```
{
  "scopes": {
    "file": {
      "integrity": true
    }
  }
}
```

Dependency redirection using scopes

The following example, would allow access to `fs` for all resources within `./app/` :

```
{
  "resources": {
    "./app/checked.js": {
      "cascade": true,
      "integrity": true
    }
  },
  "scopes": {
    "./app/": {
      "dependencies": {
        "fs": true
      }
    }
  }
}
```

The following example, would allow access to `fs` for all `data:` resources:

```
{
  "resources": {
    "data:text/javascript,import('fs');": {
      "cascade": true,
      "integrity": true
    }
  },
  "scopes": {
    "data:": {
      "dependencies": {
        "fs": true
      }
    }
  }
}
```

Process

Source Code: [lib/process.js](#)

The `process` object provides information about, and control over, the current Node.js process. While it is available as a global, it is recommended to explicitly access it via `require` or `import`:

```
import process from 'process';const process = require('process');
```

Process events

The `process` object is an instance of `EventEmitter`.

Event: 'beforeExit'

The `'beforeExit'` event is emitted when Node.js empties its event loop and has no additional work to schedule. Normally, the Node.js process will exit when there is no work scheduled, but a listener registered on the `'beforeExit'` event can make asynchronous calls, and thereby cause the Node.js process to continue.

The listener callback function is invoked with the value of `process.exitCode` passed as the only argument.

The `'beforeExit'` event is *not* emitted for conditions causing explicit termination, such as calling `process.exit()` or uncaught exceptions.

The `'beforeExit'` should *not* be used as an alternative to the `'exit'` event unless the intention is to schedule additional work.

```
import process from 'process';

process.on('beforeExit', (code) => {
  console.log('Process beforeExit event with code: ', code);
});

process.on('exit', (code) => {
  console.log('Process exit event with code: ', code);
});

console.log('This message is displayed first.');
```

// Prints:
// This message is displayed first.
// Process beforeExit event with code: 0
// Process exit event with code: 0

```
const process = require('process');

process.on('beforeExit', (code) => {
  console.log('Process beforeExit event with code: ', code);
});

process.on('exit', (code) => {
  console.log('Process exit event with code: ', code);
});

console.log('This message is displayed first.');
```

// Prints:

```
// This message is displayed first.
// Process beforeExit event with code: 0
// Process exit event with code: 0
```

Event: 'disconnect'

If the Node.js process is spawned with an IPC channel (see the [Child Process](#) and [Cluster](#) documentation), the `'disconnect'` event will be emitted when the IPC channel is closed.

Event: 'exit'

- `code` `<integer>`

The `'exit'` event is emitted when the Node.js process is about to exit as a result of either:

- The `process.exit()` method being called explicitly;
- The Node.js event loop no longer having any additional work to perform.

There is no way to prevent the exiting of the event loop at this point, and once all `'exit'` listeners have finished running the Node.js process will terminate.

The listener callback function is invoked with the exit code specified either by the `process.exitCode` property, or the `exitCode` argument passed to the `process.exit()` method.

```
import process from 'process';

process.on('exit', (code) => {
  console.log(`About to exit with code: ${code}`);
});const process = require('process');

process.on('exit', (code) => {
  console.log(`About to exit with code: ${code}`);
});
```

Listener functions **must** only perform **synchronous** operations. The Node.js process will exit immediately after calling the `'exit'` event listeners causing any additional work still queued in the event loop to be abandoned. In the following example, for instance, the timeout will never occur:

```
import process from 'process';

process.on('exit', (code) => {
  setTimeout(() => {
    console.log('This will not run');
  }, 0);
});const process = require('process');

process.on('exit', (code) => {
  setTimeout(() => {
    console.log('This will not run');
  }, 0);
});
```


Event: 'message'

- `message` `<Object>` | `<boolean>` | `<number>` | `<string>` | `<null>` a parsed JSON object or a serializable primitive value.
- `sendHandle` `<net.Server>` | `<net.Socket>` a `net.Server` or `net.Socket` object, or undefined.

If the Node.js process is spawned with an IPC channel (see the [Child Process](#) and [Cluster](#) documentation), the `'message'` event is emitted whenever a message sent by a parent process using `childprocess.send()` is received by the child process.

The message goes through serialization and parsing. The resulting message might not be the same as what is originally sent.

If the `serialization` option was set to `advanced` used when spawning the process, the `message` argument can contain data that JSON is not able to represent. See [Advanced serialization for child_process](#) for more details.

Event: 'multipleResolves'

- `type` `<string>` The resolution type. One of `'resolve'` or `'reject'`.
- `promise` `<Promise>` The promise that resolved or rejected more than once.
- `value` `<any>` The value with which the promise was either resolved or rejected after the original resolve.

The `'multipleResolves'` event is emitted whenever a `Promise` has been either:

- Resolved more than once.
- Rejected more than once.
- Rejected after resolve.
- Resolved after reject.

This is useful for tracking potential errors in an application while using the `Promise` constructor, as multiple resolutions are silently swallowed. However, the occurrence of this event does not necessarily indicate an error. For example, `Promise.race()` can trigger a `'multipleResolves'` event.

```
import process from 'process';

process.on('multipleResolves', (type, promise, reason) => {
  console.error(type, promise, reason);
  setImmediate(() => process.exit(1));
});

async function main() {
  try {
    return await new Promise((resolve, reject) => {
      resolve('First call');
      resolve('Swallowed resolve');
      reject(new Error('Swallowed reject'));
    });
  } catch {
    throw new Error('Failed');
  }
}

main().then(console.log);
// resolve: Promise { 'First call' } 'Swallowed resolve'
// reject: Promise { 'First call' } Error: Swallowed reject
//   at Promise (*)
//   at new Promise (<anonymous>)
```

```
//      at main (*)
// First callconst process = require('process');

process.on('multipleResolves', (type, promise, reason) => {
  console.error(type, promise, reason);
  setImmediate(() => process.exit(1));
});

async function main() {
  try {
    return await new Promise((resolve, reject) => {
      resolve('First call');
      resolve('Swallowed resolve');
      reject(new Error('Swallowed reject'));
    });
  } catch {
    throw new Error('Failed');
  }
}

main().then(console.log);
// resolve: Promise { 'First call' } 'Swallowed resolve'
// reject: Promise { 'First call' } Error: Swallowed reject
//      at Promise (*)
//      at new Promise (<anonymous>)
//      at main (*)
// First call
```

Event: 'rejectionHandled'

- `promise` <Promise> The late handled promise.

The 'rejectionHandled' event is emitted whenever a `Promise` has been rejected and an error handler was attached to it (using `promise.catch()`, for example) later than one turn of the Node.js event loop.

The `Promise` object would have previously been emitted in an 'unhandledRejection' event, but during the course of processing gained a rejection handler.

There is no notion of a top level for a `Promise` chain at which rejections can always be handled. Being inherently asynchronous in nature, a `Promise` rejection can be handled at a future point in time, possibly much later than the event loop turn it takes for the 'unhandledRejection' event to be emitted.

Another way of stating this is that, unlike in synchronous code where there is an ever-growing list of unhandled exceptions, with Promises there can be a growing-and-shrinking list of unhandled rejections.

In synchronous code, the 'uncaughtException' event is emitted when the list of unhandled exceptions grows.

In asynchronous code, the 'unhandledRejection' event is emitted when the list of unhandled rejections grows, and the 'rejectionHandled' event is emitted when the list of unhandled rejections shrinks.

```
import process from 'process';

const unhandledRejections = new Map();
```

```

process.on('unhandledRejection', (reason, promise) => {
  unhandledRejections.set(promise, reason);
});
process.on('rejectionHandled', (promise) => {
  unhandledRejections.delete(promise);
});const process = require('process');

const unhandledRejections = new Map();
process.on('unhandledRejection', (reason, promise) => {
  unhandledRejections.set(promise, reason);
});
process.on('rejectionHandled', (promise) => {
  unhandledRejections.delete(promise);
});

```

In this example, the `unhandledRejections` `Map` will grow and shrink over time, reflecting rejections that start unhandled and then become handled. It is possible to record such errors in an error log, either periodically (which is likely best for long-running application) or upon process exit (which is likely most convenient for scripts).

Event: 'uncaughtException'

- `err` `<Error>` The uncaught exception.
- `origin` `<string>` Indicates if the exception originates from an unhandled rejection or from an synchronous error. Can either be `'uncaughtException'` or `'unhandledRejection'`. The latter is only used in conjunction with the `--unhandled-rejections` flag set to `strict` or `throw` and an unhandled rejection.

The `'uncaughtException'` event is emitted when an uncaught JavaScript exception bubbles all the way back to the event loop. By default, Node.js handles such exceptions by printing the stack trace to `stderr` and exiting with code 1, overriding any previously set `process.exitCode`. Adding a handler for the `'uncaughtException'` event overrides this default behavior. Alternatively, change the `process.exitCode` in the `'uncaughtException'` handler which will result in the process exiting with the provided exit code. Otherwise, in the presence of such handler the process will exit with 0.

```

import process from 'process';

process.on('uncaughtException', (err, origin) => {
  fs.writeFileSync(
    process.stderr.fd,
    `Caught exception: ${err}\n` +
    `Exception origin: ${origin}`
  );
});

setTimeout(() => {
  console.log('This will still run.');
```

```

}, 500);

// Intentionally cause an exception, but don't catch it.
nonexistentFunc();
console.log('This will not run.');
```

```

const process = require('process');

process.on('uncaughtException', (err, origin) => {
  fs.writeFileSync(

```

```

    process.stderr.fd,
    `Caught exception: ${err}\n` +
    `Exception origin: ${origin}`
  );
});

setTimeout(() => {
  console.log('This will still run.');
```

```

}, 500);

// Intentionally cause an exception, but don't catch it.
nonexistentFunc();
console.log('This will not run.');
```

It is possible to monitor `'uncaughtException'` events without overriding the default behavior to exit the process by installing a `'uncaughtExceptionMonitor'` listener.

Warning: Using `'uncaughtException'` correctly

`'uncaughtException'` is a crude mechanism for exception handling intended to be used only as a last resort. The event *should not* be used as an equivalent to `On Error Resume Next`. Unhandled exceptions inherently mean that an application is in an undefined state. Attempting to resume application code without properly recovering from the exception can cause additional unforeseen and unpredictable issues.

Exceptions thrown from within the event handler will not be caught. Instead the process will exit with a non-zero exit code and the stack trace will be printed. This is to avoid infinite recursion.

Attempting to resume normally after an uncaught exception can be similar to pulling out the power cord when upgrading a computer. Nine out of ten times, nothing happens. But the tenth time, the system becomes corrupted.

The correct use of `'uncaughtException'` is to perform synchronous cleanup of allocated resources (e.g. file descriptors, handles, etc) before shutting down the process. **It is not safe to resume normal operation after `'uncaughtException'`.**

To restart a crashed application in a more reliable way, whether `'uncaughtException'` is emitted or not, an external monitor should be employed in a separate process to detect application failures and recover or restart as needed.

Event: `'uncaughtExceptionMonitor'`

- `err` `<Error>` The uncaught exception.
- `origin` `<string>` Indicates if the exception originates from an unhandled rejection or from synchronous errors. Can either be `'uncaughtException'` or `'unhandledRejection'`. The latter is only used in conjunction with the `--unhandled-rejections` flag set to `strict` or `throw` and an unhandled rejection.

The `'uncaughtExceptionMonitor'` event is emitted before an `'uncaughtException'` event is emitted or a hook installed via `process.setUncaughtExceptionCaptureCallback()` is called.

Installing an `'uncaughtExceptionMonitor'` listener does not change the behavior once an `'uncaughtException'` event is emitted. The process will still crash if no `'uncaughtException'` listener is installed.

```

import process from 'process';

process.on('uncaughtExceptionMonitor', (err, origin) => {
  MyMonitoringTool.logSync(err, origin);
});
```

```
// Intentionally cause an exception, but don't catch it.
nonexistentFunc();
// Still crashes Node.jsconst process = require('process');

process.on('uncaughtExceptionMonitor', (err, origin) => {
  MyMonitoringTool.logSync(err, origin);
});

// Intentionally cause an exception, but don't catch it.
nonexistentFunc();
// Still crashes Node.js
```

Event: 'unhandledRejection'

- `reason` `<Error>` | `<any>` The object with which the promise was rejected (typically an `Error` object).
- `promise` `<Promise>` The rejected promise.

The `'unhandledRejection'` event is emitted whenever a `Promise` is rejected and no error handler is attached to the promise within a turn of the event loop. When programming with Promises, exceptions are encapsulated as "rejected promises". Rejections can be caught and handled using `promise.catch()` and are propagated through a `Promise` chain. The `'unhandledRejection'` event is useful for detecting and keeping track of promises that were rejected whose rejections have not yet been handled.

```
import process from 'process';

process.on('unhandledRejection', (reason, promise) => {
  console.log('Unhandled Rejection at:', promise, 'reason:', reason);
  // Application specific logging, throwing an error, or other logic here
});

somePromise.then((res) => {
  return reportToUser(JSON.pasre(res)); // Note the typo (`pasre`)
}); // No `.catch()` or `.then()`const process = require('process');

process.on('unhandledRejection', (reason, promise) => {
  console.log('Unhandled Rejection at:', promise, 'reason:', reason);
  // Application specific logging, throwing an error, or other logic here
});

somePromise.then((res) => {
  return reportToUser(JSON.pasre(res)); // Note the typo (`pasre`)
}); // No `.catch()` or `.then()`
```

The following will also trigger the `'unhandledRejection'` event to be emitted:

```
import process from 'process';

function SomeResource() {
  // Initially set the loaded status to a rejected promise
  this.loaded = Promise.reject(new Error('Resource not yet loaded!'));
}
```

```
const resource = new SomeResource();
// no .catch or .then on resource.loaded for at least a turnconst process = require('process');

function SomeResource() {
  // Initially set the loaded status to a rejected promise
  this.loaded = Promise.reject(new Error('Resource not yet loaded!'));
}

const resource = new SomeResource();
// no .catch or .then on resource.loaded for at least a turn
```

In this example case, it is possible to track the rejection as a developer error as would typically be the case for other `'unhandledRejection'` events. To address such failures, a non-operational `.catch(() => { })` handler may be attached to `resource.loaded`, which would prevent the `'unhandledRejection'` event from being emitted.

Event: 'warning'

- `warning` `<Error>` Key properties of the warning are:
 - `name` `<string>` The name of the warning. Default: `'warning'`.
 - `message` `<string>` A system-provided description of the warning.
 - `stack` `<string>` A stack trace to the location in the code where the warning was issued.

The `'warning'` event is emitted whenever Node.js emits a process warning.

A process warning is similar to an error in that it describes exceptional conditions that are being brought to the user's attention. However, warnings are not part of the normal Node.js and JavaScript error handling flow. Node.js can emit warnings whenever it detects bad coding practices that could lead to sub-optimal application performance, bugs, or security vulnerabilities.

```
import process from 'process';

process.on('warning', (warning) => {
  console.warn(warning.name);    // Print the warning name
  console.warn(warning.message); // Print the warning message
  console.warn(warning.stack);   // Print the stack trace
});const process = require('process');

process.on('warning', (warning) => {
  console.warn(warning.name);    // Print the warning name
  console.warn(warning.message); // Print the warning message
  console.warn(warning.stack);   // Print the stack trace
});
```

By default, Node.js will print process warnings to `stderr`. The `--no-warnings` command-line option can be used to suppress the default console output but the `'warning'` event will still be emitted by the `process` object.

The following example illustrates the warning that is printed to `stderr` when too many listeners have been added to an event:

```
$ node
> events.defaultMaxListeners = 1;
> process.on('foo', () => {});
```

```
> process.on('foo', () => {});  
> (node:38638) MaxListenersExceededWarning: Possible EventEmitter memory leak  
detected. 2 foo listeners added. Use emitter.setMaxListeners() to increase limit
```

In contrast, the following example turns off the default warning output and adds a custom handler to the `'warning'` event:

```
$ node --no-warnings  
> const p = process.on('warning', (warning) => console.warn('Do not do that!'));  
> events.defaultMaxListeners = 1;  
> process.on('foo', () => {});  
> process.on('foo', () => {});  
> Do not do that!
```

The `--trace-warnings` command-line option can be used to have the default console output for warnings include the full stack trace of the warning.

Launching Node.js using the `--throw-deprecation` command-line flag will cause custom deprecation warnings to be thrown as exceptions.

Using the `--trace-deprecation` command-line flag will cause the custom deprecation to be printed to `stderr` along with the stack trace.

Using the `--no-deprecation` command-line flag will suppress all reporting of the custom deprecation.

The `*-deprecation` command-line flags only affect warnings that use the name `'DeprecationWarning'`.

Event: `'worker'`

- `worker` `<Worker>` The `<Worker>` that was created.

The `'worker'` event is emitted after a new `<Worker>` thread has been created.

Emitting custom warnings

See the `process.emitWarning()` method for issuing custom or application-specific warnings.

Node.js warning names

There are no strict guidelines for warning types (as identified by the `name` property) emitted by Node.js. New types of warnings can be added at any time. A few of the warning types that are most common include:

- `'DeprecationWarning'` - Indicates use of a deprecated Node.js API or feature. Such warnings must include a `'code'` property identifying the `deprecation code`.
- `'ExperimentalWarning'` - Indicates use of an experimental Node.js API or feature. Such features must be used with caution as they may change at any time and are not subject to the same strict semantic-versioning and long-term support policies as supported features.
- `'MaxListenersExceededWarning'` - Indicates that too many listeners for a given event have been registered on either an `EventEmitter` or `EventTarget`. This is often an indication of a memory leak.
- `'TimeoutOverflowWarning'` - Indicates that a numeric value that cannot fit within a 32-bit signed integer has been provided to either the `setTimeout()` or `setInterval()` functions.
- `'UnsupportedWarning'` - Indicates use of an unsupported option or feature that will be ignored rather than treated as an error. One example is use of the HTTP response status message when using the HTTP/2 compatibility API.

Signal events

Signal events will be emitted when the Node.js process receives a signal. Please refer to `signal(7)` for a listing of standard POSIX signal names such as `'SIGINT'`, `'SIGHUP'`, etc.

Signals are not available on `Worker` threads.

The signal handler will receive the signal's name (`'SIGINT'` , `'SIGTERM'` , etc.) as the first argument.

The name of each event will be the uppercase common name for the signal (e.g. `'SIGINT'` for `SIGINT` signals).

```
import process from 'process';

// Begin reading from stdin so the process does not exit.
process.stdin.resume();

process.on('SIGINT', () => {
  console.log('Received SIGINT. Press Control-D to exit.');
```



```
// Using a single function to handle multiple signals
function handle(signal) {
  console.log(`Received ${signal}`);
}

process.on('SIGINT', handle);
process.on('SIGTERM', handle);const process = require('process');
```



```
// Begin reading from stdin so the process does not exit.
process.stdin.resume();

process.on('SIGINT', () => {
  console.log('Received SIGINT. Press Control-D to exit.');
```



```
// Using a single function to handle multiple signals
function handle(signal) {
  console.log(`Received ${signal}`);
}

process.on('SIGINT', handle);
process.on('SIGTERM', handle);
```

- `'SIGUSR1'` is reserved by Node.js to start the `debugger` . It's possible to install a listener but doing so might interfere with the debugger.
- `'SIGTERM'` and `'SIGINT'` have default handlers on non-Windows platforms that reset the terminal mode before exiting with code `128 + signal number` . If one of these signals has a listener installed, its default behavior will be removed (Node.js will no longer exit).
- `'SIGPIPE'` is ignored by default. It can have a listener installed.
- `'SIGHUP'` is generated on Windows when the console window is closed, and on other platforms under various similar conditions. See `signal(7)` . It can have a listener installed, however Node.js will be unconditionally terminated by Windows about 10 seconds later. On non-Windows platforms, the default behavior of `SIGHUP` is to terminate Node.js, but once a listener has been installed its default behavior will be removed.
- `'SIGTERM'` is not supported on Windows, it can be listened on.
- `'SIGINT'` from the terminal is supported on all platforms, and can usually be generated with `Ctrl + C` (though this may be configurable). It is not generated when `terminal raw mode` is enabled and `Ctrl + C` is used.

- `'SIGBREAK'` is delivered on Windows when `Ctrl + Break` is pressed. On non-Windows platforms, it can be listened on, but there is no way to send or generate it.
- `'SIGWINCH'` is delivered when the console has been resized. On Windows, this will only happen on write to the console when the cursor is being moved, or when a readable tty is used in raw mode.
- `'SIGKILL'` cannot have a listener installed, it will unconditionally terminate Node.js on all platforms.
- `'SIGSTOP'` cannot have a listener installed.
- `'SIGBUS'`, `'SIGFPE'`, `'SIGSEGV'` and `'SIGILL'`, when not raised artificially using `kill(2)`, inherently leave the process in a state from which it is not safe to call JS listeners. Doing so might cause the process to stop responding.
- `0` can be sent to test for the existence of a process, it has no effect if the process exists, but will throw an error if the process does not exist.

Windows does not support signals so has no equivalent to termination by signal, but Node.js offers some emulation with `process.kill()`, and `subprocess.kill()`:

- Sending `SIGINT`, `SIGTERM`, and `SIGKILL` will cause the unconditional termination of the target process, and afterwards, subprocess will report that the process was terminated by signal.
- Sending signal `0` can be used as a platform independent way to test for the existence of a process.

process.abort()

The `process.abort()` method causes the Node.js process to exit immediately and generate a core file.

This feature is not available in `worker` threads.

process.allowedNodeEnvironmentFlags

- `<Set>`

The `process.allowedNodeEnvironmentFlags` property is a special, read-only `Set` of flags allowable within the `NODE_OPTIONS` environment variable.

`process.allowedNodeEnvironmentFlags` extends `Set`, but overrides `Set.prototype.has` to recognize several different possible flag representations. `process.allowedNodeEnvironmentFlags.has()` will return `true` in the following cases:

- Flags may omit leading single (`-`) or double (`--`) dashes; e.g., `inspect-brk` for `--inspect-brk`, or `r` for `-r`.
- Flags passed through to V8 (as listed in `--v8-options`) may replace one or more *non-leading* dashes for an underscore, or vice-versa; e.g., `--perf_basic_prof`, `--perf-basic-prof`, `--perf_basic-prof`, etc.
- Flags may contain one or more equals (`=`) characters; all characters after and including the first equals will be ignored; e.g., `--stack-trace-limit=100`.
- Flags *must* be allowable within `NODE_OPTIONS`.

When iterating over `process.allowedNodeEnvironmentFlags`, flags will appear only *once*; each will begin with one or more dashes. Flags passed through to V8 will contain underscores instead of non-leading dashes:

```
import { allowedNodeEnvironmentFlags } from 'process';

allowedNodeEnvironmentFlags.forEach((flag) => {
  // -r
  // --inspect-brk
  // --abort_on_uncaught_exception
  // ...
});const { allowedNodeEnvironmentFlags } = require('process');
```

```

allowedNodeEnvironmentFlags.forEach((flag) => {
  // -r
  // --inspect-brk
  // --abort_on_uncaught_exception
  // ...
});

```

The methods `add()`, `clear()`, and `delete()` of `process.allowedNodeEnvironmentFlags` do nothing, and will fail silently.

If Node.js was compiled *without* `NODE_OPTIONS` support (shown in `process.config`), `process.allowedNodeEnvironmentFlags` will contain what *would have* been allowable.

process.arch

- `<string>`

The operating system CPU architecture for which the Node.js binary was compiled. Possible values are: `'arm'`, `'arm64'`, `'ia32'`, `'mips'`, `'mipsel'`, `'ppc'`, `'ppc64'`, `'s390'`, `'s390x'`, `'x32'`, and `'x64'`.

```

import { arch } from 'process';

console.log(`This processor architecture is ${arch}`);
const { arch } = require('process');

console.log(`This processor architecture is ${process.arch}`);

```

process.argv

- `<string[]>`

The `process.argv` property returns an array containing the command-line arguments passed when the Node.js process was launched. The first element will be `process.execPath`. See `process.argv0` if access to the original value of `argv[0]` is needed. The second element will be the path to the JavaScript file being executed. The remaining elements will be any additional command-line arguments.

For example, assuming the following script for `process-args.js`:

```

import { argv } from 'process';

// print process.argv
argv.forEach((val, index) => {
  console.log(`${index}: ${val}`);
});
const { argv } = require('process');

// print process.argv
argv.forEach((val, index) => {
  console.log(`${index}: ${val}`);
});

```

Launching the Node.js process as:

```
$ node process-args.js one two=three four
```

Would generate the output:

```
0: /usr/local/bin/node
1: /Users/mjr/work/node/process-args.js
2: one
3: two=three
4: four
```

process.argv0

- `<string>`

The `process.argv0` property stores a read-only copy of the original value of `argv[0]` passed when Node.js starts.

```
$ bash -c 'exec -a customArgv0 ./node'
> process.argv[0]
'/Volumes/code/external/node/out/Release/node'
> process.argv0
'customArgv0'
```

process.channel

- `<Object>`

If the Node.js process was spawned with an IPC channel (see the [Child Process](#) documentation), the `process.channel` property is a reference to the IPC channel. If no IPC channel exists, this property is `undefined`.

process.channel.ref()

This method makes the IPC channel keep the event loop of the process running if `.unref()` has been called before.

Typically, this is managed through the number of `'disconnect'` and `'message'` listeners on the `process` object. However, this method can be used to explicitly request a specific behavior.

process.channel.unref()

This method makes the IPC channel not keep the event loop of the process running, and lets it finish even while the channel is open.

Typically, this is managed through the number of `'disconnect'` and `'message'` listeners on the `process` object. However, this method can be used to explicitly request a specific behavior.

process.chdir(directory)

- `directory` `<string>`

The `process.chdir()` method changes the current working directory of the Node.js process or throws an exception if doing so fails (for instance, if the specified `directory` does not exist).

```
import { chdir, cwd } from 'process';

console.log(`Starting directory: ${cwd()}`);
try {
  chdir('/tmp');
  console.log(`New directory: ${cwd()}`);
} catch (err) {
  console.error(`chdir: ${err}`);
}const { chdir, cwd } = require('process');

console.log(`Starting directory: ${cwd()}`);
try {
  chdir('/tmp');
  console.log(`New directory: ${cwd()}`);
} catch (err) {
  console.error(`chdir: ${err}`);
}
```

This feature is not available in `Worker` threads.

process.config

- `<Object>`

The `process.config` property returns an `Object` containing the JavaScript representation of the configure options used to compile the current Node.js executable. This is the same as the `config.gypi` file that was produced when running the `./configure` script.

An example of the possible output looks like:

```
{
  target_defaults:
    { cflags: [],
      default_configuration: 'Release',
      defines: [],
      include_dirs: [],
      libraries: [] },
  variables:
    {
      host_arch: 'x64',
      napi_build_version: 5,
      node_install_npm: 'true',
      node_prefix: '',
      node_shared_cares: 'false',
      node_shared_http_parser: 'false',
      node_shared_libuv: 'false',
      node_shared_zlib: 'false',
      node_use_dtrace: 'false',
      node_use_openssl: 'true',
      node_shared_openssl: 'false',
      strict_aliasing: 'true',
```

```
target_arch: 'x64',
v8_use_snapshot: 1
}
}
```

The `process.config` property is **not** read-only and there are existing modules in the ecosystem that are known to extend, modify, or entirely replace the value of `process.config`.

Modifying the `process.config` property, or any child-property of the `process.config` object has been deprecated. The `process.config` will be made read-only in a future release.

process.connected

- `<boolean>`

If the Node.js process is spawned with an IPC channel (see the [Child Process](#) and [Cluster](#) documentation), the `process.connected` property will return `true` so long as the IPC channel is connected and will return `false` after `process.disconnect()` is called.

Once `process.connected` is `false`, it is no longer possible to send messages over the IPC channel using `process.send()`.

process.cpuUsage([previousValue])

- `previousValue` `<Object>` A previous return value from calling `process.cpuUsage()`
- Returns: `<Object>`
 - `user` `<integer>`
 - `system` `<integer>`

The `process.cpuUsage()` method returns the user and system CPU time usage of the current process, in an object with properties `user` and `system`, whose values are microsecond values (millionth of a second). These values measure time spent in user and system code respectively, and may end up being greater than actual elapsed time if multiple CPU cores are performing work for this process.

The result of a previous call to `process.cpuUsage()` can be passed as the argument to the function, to get a diff reading.

```
import { cpuUsage } from 'process';

const startUsage = cpuUsage();
// { user: 38579, system: 6986 }

// spin the CPU for 500 milliseconds
const now = Date.now();
while (Date.now() - now < 500);

console.log(cpuUsage(startUsage));
// { user: 514883, system: 11226 }const { cpuUsage } = require('process');

const startUsage = cpuUsage();
// { user: 38579, system: 6986 }

// spin the CPU for 500 milliseconds
const now = Date.now();
while (Date.now() - now < 500);
```

```
console.log(cpuUsage(startUsage));  
// { user: 514883, system: 11226 }
```

process.cwd()

- Returns: `<string>`

The `process.cwd()` method returns the current working directory of the Node.js process.

```
import { cwd } from 'process';  
  
console.log(`Current directory: ${cwd()}`);const { cwd } = require('process');  
  
console.log(`Current directory: ${cwd()}`);
```

process.debugPort

- `<number>`

The port used by the Node.js debugger when enabled.

```
import process from 'process';  
  
process.debugPort = 5858;const process = require('process');  
  
process.debugPort = 5858;
```

process.disconnect()

If the Node.js process is spawned with an IPC channel (see the [Child Process](#) and [Cluster](#) documentation), the `process.disconnect()` method will close the IPC channel to the parent process, allowing the child process to exit gracefully once there are no other connections keeping it alive.

The effect of calling `process.disconnect()` is the same as calling `ChildProcess.disconnect()` from the parent process.

If the Node.js process was not spawned with an IPC channel, `process.disconnect()` will be `undefined`.

process.dlopen(module, filename[, flags])

- `module` `<Object>`
- `filename` `<string>`
- `flags` `<os.constants.dlopen>` **Default:** `os.constants.dlopen.RTLD_LAZY`

The `process.dlopen()` method allows dynamically loading shared objects. It is primarily used by `require()` to load C++ Addons, and should not be used directly, except in special cases. In other words, `require()` should be preferred over `process.dlopen()` unless there are specific reasons such as custom dlopen flags or loading from ES modules.

The `flags` argument is an integer that allows to specify dlopen behavior. See the [os.constants.dlopen](#) documentation for details.

An important requirement when calling `process.dlopen()` is that the `module` instance must be passed. Functions exported by the C++ Addon are then accessible via `module.exports`.

The example below shows how to load a C++ Addon, named `local.node`, that exports a `foo` function. All the symbols are loaded before the call returns, by passing the `RTLD_NOW` constant. In this example the constant is assumed to be available.

```
import { dlopen } from 'process';
import { constants } from 'os';
import { fileURLToPath } from 'url';

const module = { exports: {} };
dlopen(module, fileURLToPath(new URL('local.node', import.meta.url)),
  constants.dlopen.RTLD_NOW);
module.exports.foo();const { dlopen } = require('process');
const { constants } = require('os');
const { join } = require('path');

const module = { exports: {} };
dlopen(module, join(__dirname, 'local.node'), constants.dlopen.RTLD_NOW);
module.exports.foo();
```

process.emitWarning(warning[, options])

- `warning` `<string> | <Error>` The warning to emit.
- `options` `<Object>`
 - `type` `<string>` When `warning` is a `String`, `type` is the name to use for the *type* of warning being emitted. **Default:** `'Warning'`.
 - `code` `<string>` A unique identifier for the warning instance being emitted.
 - `ctor` `<Function>` When `warning` is a `String`, `ctor` is an optional function used to limit the generated stack trace. **Default:** `process.emitWarning`.
 - `detail` `<string>` Additional text to include with the error.

The `process.emitWarning()` method can be used to emit custom or application specific process warnings. These can be listened for by adding a handler to the `'warning'` event.

```
import { emitWarning } from 'process';

// Emit a warning with a code and additional detail.
emitWarning('Something happened!', {
  code: 'MY_WARNING',
  detail: 'This is some additional information'
});
// Emits:
// (node:56338) [MY_WARNING] Warning: Something happened!
// This is some additional informationconst { emitWarning } = require('process');

// Emit a warning with a code and additional detail.
emitWarning('Something happened!', {
  code: 'MY_WARNING',
  detail: 'This is some additional information'
```

```
});
// Emits:
// (node:56338) [MY_WARNING] Warning: Something happened!
// This is some additional information
```

In this example, an `Error` object is generated internally by `process.emitWarning()` and passed through to the `'warning'` handler.

```
import process from 'process';

process.on('warning', (warning) => {
  console.warn(warning.name);    // 'Warning'
  console.warn(warning.message); // 'Something happened!'
  console.warn(warning.code);    // 'MY_WARNING'
  console.warn(warning.stack);   // Stack trace
  console.warn(warning.detail);  // 'This is some additional information'
});const process = require('process');

process.on('warning', (warning) => {
  console.warn(warning.name);    // 'Warning'
  console.warn(warning.message); // 'Something happened!'
  console.warn(warning.code);    // 'MY_WARNING'
  console.warn(warning.stack);   // Stack trace
  console.warn(warning.detail);  // 'This is some additional information'
});
```

If `warning` is passed as an `Error` object, the `options` argument is ignored.

process.emitWarning(warning[, type[, code]][, ctor])

- `warning` `<string>` | `<Error>` The warning to emit.
- `type` `<string>` When `warning` is a `String`, `type` is the name to use for the `type` of warning being emitted. **Default:** `'Warning'`.
- `code` `<string>` A unique identifier for the warning instance being emitted.
- `ctor` `<Function>` When `warning` is a `String`, `ctor` is an optional function used to limit the generated stack trace. **Default:** `process.emitWarning`.

The `process.emitWarning()` method can be used to emit custom or application specific process warnings. These can be listened for by adding a handler to the `'warning'` event.

```
import { emitWarning } from 'process';

// Emit a warning using a string.
emitWarning('Something happened!');
// Emits: (node: 56338) Warning: Something happened!const { emitWarning } = require('process');

// Emit a warning using a string.
emitWarning('Something happened!');
// Emits: (node: 56338) Warning: Something happened!
```



```
import { emitWarning } from 'process';

// Emit a warning using a string and a type.
emitWarning('Something Happened!', 'CustomWarning');
// Emts: (node:56338) CustomWarning: Something Happened!const { emitWarning } = require('process');

// Emit a warning using a string and a type.
emitWarning('Something Happened!', 'CustomWarning');
// Emts: (node:56338) CustomWarning: Something Happened!
```

```
import { emitWarning } from 'process';

emitWarning('Something happened!', 'CustomWarning', 'WARN001');
// Emts: (node:56338) [WARN001] CustomWarning: Something happened!const { emitWarning } = require('process');

process.emitWarning('Something happened!', 'CustomWarning', 'WARN001');
// Emts: (node:56338) [WARN001] CustomWarning: Something happened!
```

In each of the previous examples, an `Error` object is generated internally by `process.emitWarning()` and passed through to the `'warning'` handler.

```
import process from 'process';

process.on('warning', (warning) => {
  console.warn(warning.name);
  console.warn(warning.message);
  console.warn(warning.code);
  console.warn(warning.stack);
});const process = require('process');

process.on('warning', (warning) => {
  console.warn(warning.name);
  console.warn(warning.message);
  console.warn(warning.code);
  console.warn(warning.stack);
});
```

If `warning` is passed as an `Error` object, it will be passed through to the `'warning'` event handler unmodified (and the optional `type`, `code` and `ctor` arguments will be ignored):

```
import { emitWarning } from 'process';

// Emit a warning using an Error object.
const myWarning = new Error('Something happened!');
// Use the Error name property to specify the type name
myWarning.name = 'CustomWarning';
myWarning.code = 'WARN001';
```

```

emitWarning(myWarning);
// Emits: (node:56338) [WARN001] CustomWarning: Something happened!const { emitWarning } = require('process');

// Emit a warning using an Error object.
const myWarning = new Error('Something happened!');
// Use the Error name property to specify the type name
myWarning.name = 'CustomWarning';
myWarning.code = 'WARN001';

emitWarning(myWarning);
// Emits: (node:56338) [WARN001] CustomWarning: Something happened!

```

A `TypeError` is thrown if `warning` is anything other than a string or `Error` object.

While process warnings use `Error` objects, the process warning mechanism is **not** a replacement for normal error handling mechanisms.

The following additional handling is implemented if the warning `type` is `'DeprecationWarning'` :

- If the `--throw-deprecation` command-line flag is used, the deprecation warning is thrown as an exception rather than being emitted as an event.
- If the `--no-deprecation` command-line flag is used, the deprecation warning is suppressed.
- If the `--trace-deprecation` command-line flag is used, the deprecation warning is printed to `stderr` along with the full stack trace.

Avoiding duplicate warnings

As a best practice, warnings should be emitted only once per process. To do so, it is recommended to place the `emitWarning()` behind a simple boolean flag as illustrated in the example below:

```

import { emitWarning } from 'process';

function emitMyWarning() {
  if (!emitMyWarning.warned) {
    emitMyWarning.warned = true;
    emitWarning('Only warn once!');
  }
}

emitMyWarning();
// Emits: (node: 56339) Warning: Only warn once!
emitMyWarning();
// Emits nothingconst { emitWarning } = require('process');

function emitMyWarning() {
  if (!emitMyWarning.warned) {
    emitMyWarning.warned = true;
    emitWarning('Only warn once!');
  }
}

emitMyWarning();
// Emits: (node: 56339) Warning: Only warn once!
emitMyWarning();
// Emits nothing

```

process.env

- `<Object>`

The `process.env` property returns an object containing the user environment. See `environ(7)`.

An example of this object looks like:

```
{
  TERM: 'xterm-256color',
  SHELL: '/usr/local/bin/bash',
  USER: 'maciej',
  PATH: '~/.bin:/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin',
  PWD: '/Users/maciej',
  EDITOR: 'vim',
  SHLVL: '1',
  HOME: '/Users/maciej',
  LOGNAME: 'maciej',
  _: '/usr/local/bin/node'
}
```

It is possible to modify this object, but such modifications will not be reflected outside the Node.js process, or (unless explicitly requested) to other `Worker` threads. In other words, the following example would not work:

```
$ node -e 'process.env.foo = "bar"' && echo $foo
```

While the following will:

```
import { env } from 'process';

env.foo = 'bar';
console.log(env.foo);const { env } = require('process');

env.foo = 'bar';
console.log(env.foo);
```

Assigning a property on `process.env` will implicitly convert the value to a string. **This behavior is deprecated.** Future versions of Node.js may throw an error when the value is not a string, number, or boolean.

```
import { env } from 'process';

env.test = null;
console.log(env.test);
// => 'null'
env.test = undefined;
console.log(env.test);
// => 'undefined'const { env } = require('process');

env.test = null;
```

```
console.log(env.test);
// => 'null'
env.test = undefined;
console.log(env.test);
// => 'undefined'
```

Use `delete` to delete a property from `process.env`.

```
import { env } from 'process';

env.TEST = 1;
delete env.TEST;
console.log(env.TEST);
// => undefined

const { env } = require('process');

env.TEST = 1;
delete env.TEST;
console.log(env.TEST);
// => undefined
```

On Windows operating systems, environment variables are case-insensitive.

```
import { env } from 'process';

env.TEST = 1;
console.log(env.test);
// => 1

const { env } = require('process');

env.TEST = 1;
console.log(env.test);
// => 1
```

Unless explicitly specified when creating a `Worker` instance, each `Worker` thread has its own copy of `process.env`, based on its parent thread's `process.env`, or whatever was specified as the `env` option to the `Worker` constructor. Changes to `process.env` will not be visible across `Worker` threads, and only the main thread can make changes that are visible to the operating system or to native add-ons.

process.execArgv

- `<string[]>`

The `process.execArgv` property returns the set of Node.js-specific command-line options passed when the Node.js process was launched. These options do not appear in the array returned by the `process.argv` property, and do not include the Node.js executable, the name of the script, or any options following the script name. These options are useful in order to spawn child processes with the same execution environment as the parent.

```
$ node --harmony script.js --version
```

Results in `process.execArgv`:

```
[ '--harmony' ]
```

And `process.argv`:

```
[ '/usr/local/bin/node', 'script.js', '--version' ]
```

Refer to [Worker constructor](#) for the detailed behavior of worker threads with this property.

process.execPath

- `<string>`

The `process.execPath` property returns the absolute pathname of the executable that started the Node.js process. Symbolic links, if any, are resolved.

```
 '/usr/local/bin/node'
```

process.exit([code])

- `code` `<integer>` The exit code. **Default:** `0`.

The `process.exit()` method instructs Node.js to terminate the process synchronously with an exit status of `code`. If `code` is omitted, exit uses either the 'success' code `0` or the value of `process.exitCode` if it has been set. Node.js will not terminate until all the 'exit' event listeners are called.

To exit with a 'failure' code:

```
import { exit } from 'process';

exit(1);const { exit } = require('process');

exit(1);
```

The shell that executed Node.js should see the exit code as `1`.

Calling `process.exit()` will force the process to exit as quickly as possible even if there are still asynchronous operations pending that have not yet completed fully, including I/O operations to `process.stdout` and `process.stderr`.

In most situations, it is not actually necessary to call `process.exit()` explicitly. The Node.js process will exit on its own *if there is no additional work pending* in the event loop. The `process.exitCode` property can be set to tell the process which exit code to use when the process exits gracefully.

For instance, the following example illustrates a *misuse* of the `process.exit()` method that could lead to data printed to stdout being truncated and lost:

```
import { exit } from 'process';

// This is an example of what *not* to do:
if (someConditionNotMet()) {
  printUsageToStdout();
```

```

    exit(1);
}const { exit } = require('process');

// This is an example of what *not* to do:
if (someConditionNotMet()) {
    printUsageToStdout();
    exit(1);
}

```

The reason this is problematic is because writes to `process.stdout` in Node.js are sometimes *asynchronous* and may occur over multiple ticks of the Node.js event loop. Calling `process.exit()`, however, forces the process to exit *before* those additional writes to `stdout` can be performed.

Rather than calling `process.exit()` directly, the code *should* set the `process.exitCode` and allow the process to exit naturally by avoiding scheduling any additional work for the event loop:

```

import process from 'process';

// How to properly set the exit code while letting
// the process exit gracefully.
if (someConditionNotMet()) {
    printUsageToStdout();
    process.exitCode = 1;
}const process = require('process');

// How to properly set the exit code while letting
// the process exit gracefully.
if (someConditionNotMet()) {
    printUsageToStdout();
    process.exitCode = 1;
}

```

If it is necessary to terminate the Node.js process due to an error condition, throwing an *uncaught* error and allowing the process to terminate accordingly is safer than calling `process.exit()`.

In **worker** threads, this function stops the current thread rather than the current process.

process.exitCode

- `<integer>`

A number which will be the process exit code, when the process either exits gracefully, or is exited via `process.exit()` without specifying a code.

Specifying a code to `process.exit(code)` will override any previous setting of `process.exitCode`.

process.getegid()

The `process.getegid()` method returns the numerical effective group identity of the Node.js process. (See `getegid(2)`.)

```
import process from 'process';

if (process.getegid) {
  console.log(`Current gid: ${process.getegid()}`);
}const process = require('process');

if (process.getegid) {
  console.log(`Current gid: ${process.getegid()}`);
}
```

This function is only available on POSIX platforms (i.e. not Windows or Android).

process.geteuid()

- Returns: `<Object>`

The `process.geteuid()` method returns the numerical effective user identity of the process. (See [geteuid\(2\)](#) .)

```
import process from 'process';

if (process.geteuid) {
  console.log(`Current uid: ${process.geteuid()}`);
}const process = require('process');

if (process.geteuid) {
  console.log(`Current uid: ${process.geteuid()}`);
}
```

This function is only available on POSIX platforms (i.e. not Windows or Android).

process.getgid()

- Returns: `<Object>`

The `process.getgid()` method returns the numerical group identity of the process. (See [getgid\(2\)](#) .)

```
import process from 'process';

if (process.getgid) {
  console.log(`Current gid: ${process.getgid()}`);
}const process = require('process');

if (process.getgid) {
  console.log(`Current gid: ${process.getgid()}`);
}
```

This function is only available on POSIX platforms (i.e. not Windows or Android).

process.getgroups()

- Returns: `<integer[]>`

The `process.getgroups()` method returns an array with the supplementary group IDs. POSIX leaves it unspecified if the effective group ID is included but Node.js ensures it always is.

```
import process from 'process';

if (process.getgroups) {
  console.log(process.getgroups()); // [ 16, 21, 297 ]
}const process = require('process');

if (process.getgroups) {
  console.log(process.getgroups()); // [ 16, 21, 297 ]
}
```

This function is only available on POSIX platforms (i.e. not Windows or Android).

process.getuid()

- Returns: `<integer>`

The `process.getuid()` method returns the numeric user identity of the process. (See `getuid(2)`.)

```
import process from 'process';

if (process.getuid) {
  console.log(`Current uid: ${process.getuid()}`);
}const process = require('process');

if (process.getuid) {
  console.log(`Current uid: ${process.getuid()}`);
}
```

This function is only available on POSIX platforms (i.e. not Windows or Android).

process.hasUncaughtExceptionCaptureCallback()

- Returns: `<boolean>`

Indicates whether a callback has been set using `process.setUncaughtExceptionCaptureCallback()`.

process.hrtime([time])

Stability: 3 - Legacy. Use `process.hrtime.bigint()` instead.

- `time` `<integer[]>` The result of a previous call to `process.hrtime()`
- Returns: `<integer[]>`

This is the legacy version of `process.hrtime.bigint()` before `bigint` was introduced in JavaScript.

The `process.hrtime()` method returns the current high-resolution real time in a `[seconds, nanoseconds]` tuple Array, where `nanoseconds` is the remaining part of the real time that can't be represented in second precision.

`time` is an optional parameter that must be the result of a previous `process.hrtime()` call to diff with the current time. If the parameter passed in is not a tuple Array, a `TypeError` will be thrown. Passing in a user-defined array instead of the result of a previous call to `process.hrtime()` will lead to undefined behavior.

These times are relative to an arbitrary time in the past, and not related to the time of day and therefore not subject to clock drift. The primary use is for measuring performance between intervals:

```
import { hrtime } from 'process';

const NS_PER_SEC = 1e9;
const time = hrtime();
// [ 1800216, 25 ]

setTimeout(() => {
  const diff = hrtime(time);
  // [ 1, 552 ]

  console.log(`Benchmark took ${diff[0] * NS_PER_SEC + diff[1]} nanoseconds`);
  // Benchmark took 1000000552 nanoseconds
}, 1000);const { hrtime } = require('process');

const NS_PER_SEC = 1e9;
const time = hrtime();
// [ 1800216, 25 ]

setTimeout(() => {
  const diff = hrtime(time);
  // [ 1, 552 ]

  console.log(`Benchmark took ${diff[0] * NS_PER_SEC + diff[1]} nanoseconds`);
  // Benchmark took 1000000552 nanoseconds
}, 1000);
```

process.hrtime.bigint()

- Returns: `<bigint>`

The `bigint` version of the `process.hrtime()` method returning the current high-resolution real time in nanoseconds as a `bigint`.

Unlike `process.hrtime()`, it does not support an additional `time` argument since the difference can just be computed directly by subtraction of the two `bigint`s.

```
import { hrtime } from 'process';

const start = hrtime.bigint();
// 191051479007711n

setTimeout(() => {
```

```

const end = hrtime.bigint();
// 191052633396993n

console.log(`Benchmark took ${end - start} nanoseconds`);
// Benchmark took 1154389282 nanoseconds
}, 1000);const { hrtime } = require('process');

const start = hrtime.bigint();
// 191051479007711n

setTimeout(() => {
  const end = hrtime.bigint();
  // 191052633396993n

  console.log(`Benchmark took ${end - start} nanoseconds`);
  // Benchmark took 1154389282 nanoseconds
}, 1000);

```

process.initgroups(user, extraGroup)

- `user` `<string>` | `<number>` The user name or numeric identifier.
- `extraGroup` `<string>` | `<number>` A group name or numeric identifier.

The `process.initgroups()` method reads the `/etc/group` file and initializes the group access list, using all groups of which the user is a member. This is a privileged operation that requires that the Node.js process either have `root` access or the `CAP_SETGID` capability.

Use care when dropping privileges:

```

import { getgroups, initgroups, setgid } from 'process';

console.log(getgroups());           // [ 0 ]
initgroups('nodeuser', 1000);      // switch user
console.log(getgroups());           // [ 27, 30, 46, 1000, 0 ]
setgid(1000);                       // drop root gid
console.log(getgroups());           // [ 27, 30, 46, 1000 ]const { getgroups, initgroups, setgid } = require('process');

console.log(getgroups());           // [ 0 ]
initgroups('nodeuser', 1000);      // switch user
console.log(getgroups());           // [ 27, 30, 46, 1000, 0 ]
setgid(1000);                       // drop root gid
console.log(getgroups());           // [ 27, 30, 46, 1000 ]

```

This function is only available on POSIX platforms (i.e. not Windows or Android). This feature is not available in `worker` threads.

process.kill(pid[, signal])

- `pid` `<number>` A process ID
- `signal` `<string>` | `<number>` The signal to send, either as a string or number. **Default:** `'SIGTERM'`.

The `process.kill()` method sends the `signal` to the process identified by `pid`.

Signal names are strings such as `'SIGINT'` or `'SIGHUP'`. See [Signal Events](#) and `kill(2)` for more information.

This method will throw an error if the target `pid` does not exist. As a special case, a signal of `0` can be used to test for the existence of a process. Windows platforms will throw an error if the `pid` is used to kill a process group.

Even though the name of this function is `process.kill()`, it is really just a signal sender, like the `kill` system call. The signal sent may do something other than kill the target process.

```
import process, { kill } from 'process';

process.on('SIGHUP', () => {
  console.log('Got SIGHUP signal.');
```



```
});

setTimeout(() => {
  console.log('Exiting.');
```



```
process.exit(0);
}, 100);

kill(process.pid, 'SIGHUP');const process = require('process');

process.on('SIGHUP', () => {
  console.log('Got SIGHUP signal.');
```



```
});

setTimeout(() => {
  console.log('Exiting.');
```



```
process.exit(0);
}, 100);

process.kill(process.pid, 'SIGHUP');
```

When `SIGUSR1` is received by a Node.js process, Node.js will start the debugger. See [Signal Events](#).

process.mainModule

Stability: 0 - Deprecated: Use `require.main` instead.

- `<Object>`

The `process.mainModule` property provides an alternative way of retrieving `require.main`. The difference is that if the main module changes at runtime, `require.main` may still refer to the original main module in modules that were required before the change occurred. Generally, it's safe to assume that the two refer to the same module.

As with `require.main`, `process.mainModule` will be `undefined` if there is no entry script.

process.memoryUsage()

- Returns: `<Object>`
 - `rss` `<integer>`

- `heapTotal` `<integer>`
- `heapUsed` `<integer>`
- `external` `<integer>`
- `arrayBuffers` `<integer>`

Returns an object describing the memory usage of the Node.js process measured in bytes.

```
import { memoryUsage } from 'process';

console.log(memoryUsage());
// Prints:
// {
//   rss: 4935680,
//   heapTotal: 1826816,
//   heapUsed: 650472,
//   external: 49879,
//   arrayBuffers: 9386
// }const { memoryUsage } = require('process');

console.log(memoryUsage());
// Prints:
// {
//   rss: 4935680,
//   heapTotal: 1826816,
//   heapUsed: 650472,
//   external: 49879,
//   arrayBuffers: 9386
// }
```

- `heapTotal` and `heapUsed` refer to V8's memory usage.
- `external` refers to the memory usage of C++ objects bound to JavaScript objects managed by V8.
- `rss`, Resident Set Size, is the amount of space occupied in the main memory device (that is a subset of the total allocated memory) for the process, including all C++ and JavaScript objects and code.
- `arrayBuffers` refers to memory allocated for `ArrayBuffer`s and `SharedArrayBuffer`s, including all Node.js `Buffer`s. This is also included in the `external` value. When Node.js is used as an embedded library, this value may be `0` because allocations for `ArrayBuffer`s may not be tracked in that case.

When using `Worker` threads, `rss` will be a value that is valid for the entire process, while the other fields will only refer to the current thread.

The `process.memoryUsage()` method iterates over each page to gather information about memory usage which might be slow depending on the program memory allocations.

process.memoryUsage.rss()

- Returns: `<integer>`

The `process.memoryUsage.rss()` method returns an integer representing the Resident Set Size (RSS) in bytes.

The Resident Set Size, is the amount of space occupied in the main memory device (that is a subset of the total allocated memory) for the process, including all C++ and JavaScript objects and code.

This is the same value as the `rss` property provided by `process.memoryUsage()` but `process.memoryUsage.rss()` is faster.

```
import { memoryUsage } from 'process';

console.log(memoryUsage.rss());
// 35655680
const { rss } = require('process');

console.log(memoryUsage.rss());
// 35655680
```

process.nextTick(callback[, ...args])

- `callback` `<Function>`
- `...args` `<any>` Additional arguments to pass when invoking the `callback`

`process.nextTick()` adds `callback` to the "next tick queue". This queue is fully drained after the current operation on the JavaScript stack runs to completion and before the event loop is allowed to continue. It's possible to create an infinite loop if one were to recursively call `process.nextTick()`. See the [Event Loop](#) guide for more background.

```
import { nextTick } from 'process';

console.log('start');
nextTick(() => {
  console.log('nextTick callback');
});
console.log('scheduled');
// Output:
// start
// scheduled
// nextTick callback
const { nextTick } = require('process');

console.log('start');
nextTick(() => {
  console.log('nextTick callback');
});
console.log('scheduled');
// Output:
// start
// scheduled
// nextTick callback
```

This is important when developing APIs in order to give users the opportunity to assign event handlers *after* an object has been constructed but before any I/O has occurred:

```
import { nextTick } from 'process';

function MyThing(options) {
  this.setupOptions(options);

  nextTick(() => {
    this.startDoingStuff();
  });
}
```

```

});
}

const thing = new MyThing();
thing.getReadyForStuff();

// thing.startDoingStuff() gets called now, not before.
const { nextTick } = require('process');

function MyThing(options) {
  this.setupOptions(options);

  nextTick(() => {
    this.startDoingStuff();
  });
}

const thing = new MyThing();
thing.getReadyForStuff();

// thing.startDoingStuff() gets called now, not before.

```

It is very important for APIs to be either 100% synchronous or 100% asynchronous. Consider this example:

```

// WARNING! DO NOT USE! BAD UNSAFE HAZARD!
function maybeSync(arg, cb) {
  if (arg) {
    cb();
    return;
  }

  fs.stat('file', cb);
}

```

This API is hazardous because in the following case:

```

const maybeTrue = Math.random() > 0.5;

maybeSync(maybeTrue, () => {
  foo();
});

bar();

```

It is not clear whether `foo()` or `bar()` will be called first.

The following approach is much better:

```

import { nextTick } from 'process';

function definitelyAsync(arg, cb) {

```

```

    if (arg) {
      nextTick(cb);
      return;
    }

    fs.stat('file', cb);
  }const { nextTick } = require('process');

function definitelyAsync(arg, cb) {
  if (arg) {
    nextTick(cb);
    return;
  }

  fs.stat('file', cb);
}

```

When to use `queueMicrotask()` vs. `process.nextTick()`

The `queueMicrotask()` API is an alternative to `process.nextTick()` that also defers execution of a function using the same microtask queue used to execute the then, catch, and finally handlers of resolved promises. Within Node.js, every time the "next tick queue" is drained, the microtask queue is drained immediately after.

```

import { nextTick } from 'process';

Promise.resolve().then(() => console.log(2));
queueMicrotask(() => console.log(3));
nextTick(() => console.log(1));
// Output:
// 1
// 2
// 3const { nextTick } = require('process');

Promise.resolve().then(() => console.log(2));
queueMicrotask(() => console.log(3));
nextTick(() => console.log(1));
// Output:
// 1
// 2
// 3

```

For *most* userland use cases, the `queueMicrotask()` API provides a portable and reliable mechanism for deferring execution that works across multiple JavaScript platform environments and should be favored over `process.nextTick()`. In simple scenarios, `queueMicrotask()` can be a drop-in replacement for `process.nextTick()`.

```

console.log('start');
queueMicrotask(() => {
  console.log('microtask callback');
});
console.log('scheduled');

```

```
// Output:  
// start  
// scheduled  
// microtask callback
```

One note-worthy difference between the two APIs is that `process.nextTick()` allows specifying additional values that will be passed as arguments to the deferred function when it is called. Achieving the same result with `queueMicrotask()` requires using either a closure or a bound function:

```
function deferred(a, b) {  
  console.log('microtask', a + b);  
}  
  
console.log('start');  
queueMicrotask(deferred.bind(undefined, 1, 2));  
console.log('scheduled');  
// Output:  
// start  
// scheduled  
// microtask 3
```

There are minor differences in the way errors raised from within the next tick queue and microtask queue are handled. Errors thrown within a queued microtask callback should be handled within the queued callback when possible. If they are not, the `process.on('uncaughtException')` event handler can be used to capture and handle the errors.

When in doubt, unless the specific capabilities of `process.nextTick()` are needed, use `queueMicrotask()`.

process.noDeprecation

- `<boolean>`

The `process.noDeprecation` property indicates whether the `--no-deprecation` flag is set on the current Node.js process. See the documentation for the `'warning' event` and the `emitWarning()` method for more information about this flag's behavior.

process.pid

- `<integer>`

The `process.pid` property returns the PID of the process.

```
import { pid } from 'process';  
  
console.log(`This process is pid ${pid}`);const { pid } = require('process');  
  
console.log(`This process is pid ${pid}`);
```

process.platform

- `<string>`

The `process.platform` property returns a string identifying the operating system platform on which the Node.js process is running.

Currently possible values are:

- `'aix'`
- `'darwin'`
- `'freebsd'`
- `'linux'`
- `'openbsd'`
- `'sunos'`
- `'win32'`

```
import { platform } from 'process';

console.log(`This platform is ${platform}`);const { platform } = require('process');

console.log(`This platform is ${platform}`);
```

The value `'android'` may also be returned if the Node.js is built on the Android operating system. However, Android support in Node.js is *experimental*.

process.ppid

- `<integer>`

The `process.ppid` property returns the PID of the parent of the current process.

```
import { ppid } from 'process';

console.log(`The parent process is pid ${ppid}`);const { ppid } = require('process');

console.log(`The parent process is pid ${ppid}`);
```

process.release

- `<Object>`

The `process.release` property returns an `Object` containing metadata related to the current release, including URLs for the source tarball and headers-only tarball.

`process.release` contains the following properties:

- `name` `<string>` A value that will always be `'node'`.
- `sourceUrl` `<string>` an absolute URL pointing to a `.tar.gz` file containing the source code of the current release.
- `headersUrl` `<string>` an absolute URL pointing to a `.tar.gz` file containing only the source header files for the current release. This file is significantly smaller than the full source file and can be used for compiling Node.js native add-ons.
- `libUrl` `<string>` an absolute URL pointing to a `node.lib` file matching the architecture and version of the current release. This file is used for compiling Node.js native add-ons. *This property is only present on Windows builds of Node.js and will be missing on all other platforms.*
- `lts` `<string>` a string label identifying the LTS label for this release. This property only exists for LTS releases and is `undefined` for all other release types, including *Current* releases. Valid values include the LTS Release code names (including those that are no longer

supported).

- 'Dubnium' for the 10.x LTS line beginning with 10.13.0.
- 'Erbium' for the 12.x LTS line beginning with 12.13.0.

For other LTS Release code names, see [Node.js Changelog Archive](#)

```
{
  name: 'node',
  lts: 'Erbium',
  sourceUrl: 'https://nodejs.org/download/release/v12.18.1/node-v12.18.1.tar.gz',
  headersUrl: 'https://nodejs.org/download/release/v12.18.1/node-v12.18.1-headers.tar.gz',
  libUrl: 'https://nodejs.org/download/release/v12.18.1/win-x64/node.lib'
}
```

In custom builds from non-release versions of the source tree, only the `name` property may be present. The additional properties should not be relied upon to exist.

process.report

- `<Object>`

`process.report` is an object whose methods are used to generate diagnostic reports for the current process. Additional documentation is available in the [report documentation](#).

process.report.compact

- `<boolean>`

Write reports in a compact format, single-line JSON, more easily consumable by log processing systems than the default multi-line format designed for human consumption.

```
import { report } from 'process';

console.log(`Reports are compact? ${report.compact}`);const { report } = require('process');

console.log(`Reports are compact? ${report.compact}`);
```

process.report.directory

- `<string>`

Directory where the report is written. The default value is the empty string, indicating that reports are written to the current working directory of the Node.js process.

```
import { report } from 'process';

console.log(`Report directory is ${report.directory}`);const { report } = require('process');

console.log(`Report directory is ${report.directory}`);
```

process.report.filename

- `<string>`

Filename where the report is written. If set to the empty string, the output filename will be comprised of a timestamp, PID, and sequence number. The default value is the empty string.

```
import { report } from 'process';

console.log(`Report filename is ${report.filename}`);const { report } = require('process');

console.log(`Report filename is ${report.filename}`);
```

process.report.getReport([err])

- `err` `<Error>` A custom error used for reporting the JavaScript stack.
- Returns: `<Object>`

Returns a JavaScript Object representation of a diagnostic report for the running process. The report's JavaScript stack trace is taken from `err`, if present.

```
import { report } from 'process';

const data = report.getReport();
console.log(data.header.nodejsVersion);

// Similar to process.report.writeReport()
import fs from 'fs';
fs.writeFileSync('my-report.log', util.inspect(data), 'utf8');const { report } = require('process');

const data = report.getReport();
console.log(data.header.nodejsVersion);

// Similar to process.report.writeReport()
const fs = require('fs');
fs.writeFileSync('my-report.log', util.inspect(data), 'utf8');
```

Additional documentation is available in the [report documentation](#).

process.report.reportOnFatalError

- `<boolean>`

If `true`, a diagnostic report is generated on fatal errors, such as out of memory errors or failed C++ assertions.

```
import { report } from 'process';

console.log(`Report on fatal error: ${report.reportOnFatalError}`);const { report } = require('process');

console.log(`Report on fatal error: ${report.reportOnFatalError}`);
```

process.report.reportOnSignal

- `<boolean>`

If `true`, a diagnostic report is generated when the process receives the signal specified by `process.report.signal`.

```
import { report } from 'process';

console.log(`Report on signal: ${report.reportOnSignal}`);const { report } = require('process');

console.log(`Report on signal: ${report.reportOnSignal}`);
```

process.report.reportOnUncaughtException

- `<boolean>`

If `true`, a diagnostic report is generated on uncaught exception.

```
import { report } from 'process';

console.log(`Report on exception: ${report.reportOnUncaughtException}`);const { report } = require('process');

console.log(`Report on exception: ${report.reportOnUncaughtException}`);
```

process.report.signal

- `<string>`

The signal used to trigger the creation of a diagnostic report. Defaults to `'SIGUSR2'`.

```
import { report } from 'process';

console.log(`Report signal: ${report.signal}`);const { report } = require('process');

console.log(`Report signal: ${report.signal}`);
```

process.report.writeReport([filename], [err])

- `filename` `<string>` Name of the file where the report is written. This should be a relative path, that will be appended to the directory specified in `process.report.directory`, or the current working directory of the Node.js process, if unspecified.
- `err` `<Error>` A custom error used for reporting the JavaScript stack.
- Returns: `<string>` Returns the filename of the generated report.

Writes a diagnostic report to a file. If `filename` is not provided, the default filename includes the date, time, PID, and a sequence number. The report's JavaScript stack trace is taken from `err`, if present.

```
import { report } from 'process';

report.writeReport();const { report } = require('process');

report.writeReport();
```

Additional documentation is available in the [report documentation](#).

process.resourceUsage()

- Returns: `<Object>` the resource usage for the current process. All of these values come from the `uv_getrusage` call which returns a `uv_rusage_t` struct.
 - `userCPUTime <integer>` maps to `ru_utime` computed in microseconds. It is the same value as `process.cpuUsage().user`.
 - `systemCPUTime <integer>` maps to `ru_stime` computed in microseconds. It is the same value as `process.cpuUsage().system`.
 - `maxRSS <integer>` maps to `ru_maxrss` which is the maximum resident set size used in kilobytes.
 - `sharedMemorySize <integer>` maps to `ru_ixrss` but is not supported by any platform.
 - `unsharedDataSize <integer>` maps to `ru_idrss` but is not supported by any platform.
 - `unsharedStackSize <integer>` maps to `ru_isrss` but is not supported by any platform.
 - `minorPageFault <integer>` maps to `ru_minflt` which is the number of minor page faults for the process, see [this article for more details](#).
 - `majorPageFault <integer>` maps to `ru_majflt` which is the number of major page faults for the process, see [this article for more details](#). This field is not supported on Windows.
 - `swappedOut <integer>` maps to `ru_nswap` but is not supported by any platform.
 - `fsRead <integer>` maps to `ru_inblock` which is the number of times the file system had to perform input.
 - `fsWrite <integer>` maps to `ru_oublock` which is the number of times the file system had to perform output.
 - `ipcSent <integer>` maps to `ru_msgsnd` but is not supported by any platform.
 - `ipcReceived <integer>` maps to `ru_msgrcv` but is not supported by any platform.
 - `signalsCount <integer>` maps to `ru_nsignals` but is not supported by any platform.
 - `voluntaryContextSwitches <integer>` maps to `ru_nvcsw` which is the number of times a CPU context switch resulted due to a process voluntarily giving up the processor before its time slice was completed (usually to await availability of a resource). This field is not supported on Windows.
 - `involuntaryContextSwitches <integer>` maps to `ru_nivcsw` which is the number of times a CPU context switch resulted due to a higher priority process becoming runnable or because the current process exceeded its time slice. This field is not supported on Windows.

```
import { resourceUsage } from 'process';
```

```
console.log(resourceUsage());
```

```
/*
```

```
Will output:
```

```
{
  userCPUTime: 82872,
  systemCPUTime: 4143,
  maxRSS: 33164,
  sharedMemorySize: 0,
  unsharedDataSize: 0,
  unsharedStackSize: 0,
  minorPageFault: 2469,
  majorPageFault: 0,
  swappedOut: 0,
  fsRead: 0,
  fsWrite: 8,
  ipcSent: 0,
  ipcReceived: 0,
  signalsCount: 0,
  voluntaryContextSwitches: 79,
```

```

    involuntaryContextSwitches: 1
  }
  */const { resourceUsage } = require('process');

console.log(resourceUsage());
/*
Will output:
{
  userCPUTime: 82872,
  systemCPUTime: 4143,
  maxRSS: 33164,
  sharedMemorySize: 0,
  unsharedDataSize: 0,
  unsharedStackSize: 0,
  minorPageFault: 2469,
  majorPageFault: 0,
  swappedOut: 0,
  fsRead: 0,
  fsWrite: 8,
  ipcSent: 0,
  ipcReceived: 0,
  signalsCount: 0,
  voluntaryContextSwitches: 79,
  involuntaryContextSwitches: 1
}
*/

```

process.send(message[, sendHandle[, options]][, callback])

- `message` `<Object>`
- `sendHandle` `<net.Server> | <net.Socket>`
- `options` `<Object>` used to parameterize the sending of certain types of handles. `options` supports the following properties:
 - `keepOpen` `<boolean>` A value that can be used when passing instances of `net.Socket`. When `true`, the socket is kept open in the sending process. **Default:** `false`.
- `callback` `<Function>`
- Returns: `<boolean>`

If Node.js is spawned with an IPC channel, the `process.send()` method can be used to send messages to the parent process. Messages will be received as a `'message'` event on the parent's `ChildProcess` object.

If Node.js was not spawned with an IPC channel, `process.send` will be `undefined`.

The message goes through serialization and parsing. The resulting message might not be the same as what is originally sent.

process.setegid(id)

- `id` `<string> | <number>` A group name or ID

The `process.setegid()` method sets the effective group identity of the process. (See `setegid(2)`.) The `id` can be passed as either a numeric ID or a group name string. If a group name is specified, this method blocks while resolving the associated a numeric ID.

```
import process from 'process';

if (process.getegid && process.setegid) {
  console.log(`Current gid: ${process.getegid()}`);
  try {
    process.setegid(501);
    console.log(`New gid: ${process.getegid()}`);
  } catch (err) {
    console.log(`Failed to set gid: ${err}`);
  }
}

const process = require('process');

if (process.getegid && process.setegid) {
  console.log(`Current gid: ${process.getegid()}`);
  try {
    process.setegid(501);
    console.log(`New gid: ${process.getegid()}`);
  } catch (err) {
    console.log(`Failed to set gid: ${err}`);
  }
}
```

This function is only available on POSIX platforms (i.e. not Windows or Android). This feature is not available in `Worker` threads.

process.seteuid(id)

- `id` `<string>` | `<number>` A user name or ID

The `process.seteuid()` method sets the effective user identity of the process. (See `seteuid(2)`.) The `id` can be passed as either a numeric ID or a username string. If a username is specified, the method blocks while resolving the associated numeric ID.

```
import process from 'process';

if (process.geteuid && process.seteuid) {
  console.log(`Current uid: ${process.geteuid()}`);
  try {
    process.seteuid(501);
    console.log(`New uid: ${process.geteuid()}`);
  } catch (err) {
    console.log(`Failed to set uid: ${err}`);
  }
}

const process = require('process');

if (process.geteuid && process.seteuid) {
  console.log(`Current uid: ${process.geteuid()}`);
  try {
    process.seteuid(501);
    console.log(`New uid: ${process.geteuid()}`);
  } catch (err) {
    console.log(`Failed to set uid: ${err}`);
  }
}
```

```
}  
}
```

This function is only available on POSIX platforms (i.e. not Windows or Android). This feature is not available in `worker` threads.

process.setgid(id)

- `id` `<string> | <number>` The group name or ID

The `process.setgid()` method sets the group identity of the process. (See `setgid(2)`.) The `id` can be passed as either a numeric ID or a group name string. If a group name is specified, this method blocks while resolving the associated numeric ID.

```
import process from 'process';  
  
if (process.getgid && process.setgid) {  
  console.log(`Current gid: ${process.getgid()}`);  
  try {  
    process.setgid(501);  
    console.log(`New gid: ${process.getgid()}`);  
  } catch (err) {  
    console.log(`Failed to set gid: ${err}`);  
  }  
}  
const process = require('process');  
  
if (process.getgid && process.setgid) {  
  console.log(`Current gid: ${process.getgid()}`);  
  try {  
    process.setgid(501);  
    console.log(`New gid: ${process.getgid()}`);  
  } catch (err) {  
    console.log(`Failed to set gid: ${err}`);  
  }  
}
```

This function is only available on POSIX platforms (i.e. not Windows or Android). This feature is not available in `worker` threads.

process.setgroups(groups)

- `groups` `<integer[]>`

The `process.setgroups()` method sets the supplementary group IDs for the Node.js process. This is a privileged operation that requires the Node.js process to have `root` or the `CAP_SETGID` capability.

The `groups` array can contain numeric group IDs, group names, or both.

```
import process from 'process';  
  
if (process.getgroups && process.setgroups) {  
  try {  
    process.setgroups([501]);  
    console.log(process.getgroups()); // new groups
```



```

    } catch (err) {
      console.log(`Failed to set groups: ${err}`);
    }
  }
}

const process = require('process');

if (process.getgroups && process.setgroups) {
  try {
    process.setgroups([501]);
    console.log(process.getgroups()); // new groups
  } catch (err) {
    console.log(`Failed to set groups: ${err}`);
  }
}

```

This function is only available on POSIX platforms (i.e. not Windows or Android). This feature is not available in `worker` threads.

process.setuid(id)

- `id` `<integer> | <string>`

The `process.setuid(id)` method sets the user identity of the process. (See `setuid(2)` .) The `id` can be passed as either a numeric ID or a username string. If a username is specified, the method blocks while resolving the associated numeric ID.

```

import process from 'process';

if (process.getuid && process.setuid) {
  console.log(`Current uid: ${process.getuid()}`);
  try {
    process.setuid(501);
    console.log(`New uid: ${process.getuid()}`);
  } catch (err) {
    console.log(`Failed to set uid: ${err}`);
  }
}

const process = require('process');

if (process.getuid && process.setuid) {
  console.log(`Current uid: ${process.getuid()}`);
  try {
    process.setuid(501);
    console.log(`New uid: ${process.getuid()}`);
  } catch (err) {
    console.log(`Failed to set uid: ${err}`);
  }
}

```

This function is only available on POSIX platforms (i.e. not Windows or Android). This feature is not available in `worker` threads.

process.setSourceMapsEnabled(val)

- `val` `<boolean>`

This function enables or disables the `Source Map v3` support for stack traces.

It provides same features as launching Node.js process with commandline options `--enable-source-maps`.

Only source maps in JavaScript files that are loaded after source maps has been enabled will be parsed and loaded.

`process.setUncaughtExceptionCaptureCallback(fn)`

- `fn` `<Function>` | `<null>`

The `process.setUncaughtExceptionCaptureCallback()` function sets a function that will be invoked when an uncaught exception occurs, which will receive the exception value itself as its first argument.

If such a function is set, the `'uncaughtException'` event will not be emitted. If `--abort-on-uncaught-exception` was passed from the command line or set through `v8.setFlagsFromString()`, the process will not abort. Actions configured to take place on exceptions such as report generations will be affected too

To unset the capture function, `process.setUncaughtExceptionCaptureCallback(null)` may be used. Calling this method with a non-`null` argument while another capture function is set will throw an error.

Using this function is mutually exclusive with using the deprecated `domain` built-in module.

`process.stderr`

- `<Stream>`

The `process.stderr` property returns a stream connected to `stderr` (fd `2`). It is a `net.Socket` (which is a `Duplex` stream) unless fd `2` refers to a file, in which case it is a `Writable` stream.

`process.stderr` differs from other Node.js streams in important ways. See [note on process I/O](#) for more information.

`process.stderr.fd`

- `<number>`

This property refers to the value of underlying file descriptor of `process.stderr`. The value is fixed at `2`. In `worker` threads, this field does not exist.

`process.stdin`

- `<Stream>`

The `process.stdin` property returns a stream connected to `stdin` (fd `0`). It is a `net.Socket` (which is a `Duplex` stream) unless fd `0` refers to a file, in which case it is a `Readable` stream.

For details of how to read from `stdin` see `readable.read()`.

As a `Duplex` stream, `process.stdin` can also be used in "old" mode that is compatible with scripts written for Node.js prior to v0.10. For more information see [Stream compatibility](#).

In "old" streams mode the `stdin` stream is paused by default, so one must call `process.stdin.resume()` to read from it. Note also that calling `process.stdin.resume()` itself would switch stream to "old" mode.

`process.stdin.fd`

- `<number>`

This property refers to the value of underlying file descriptor of `process.stdin`. The value is fixed at `0`. In `Worker` threads, this field does not exist.

`process.stdout`

- `<Stream>`

The `process.stdout` property returns a stream connected to `stdout` (fd `1`). It is a `net.Socket` (which is a `Duplex` stream) unless fd `1` refers to a file, in which case it is a `Writable` stream.

For example, to copy `process.stdin` to `process.stdout` :

```
import { stdin, stdout } from 'process';

stdin.pipe(stdout);const { stdin, stdout } = require('process');

stdin.pipe(stdout);
```

`process.stdout` differs from other Node.js streams in important ways. See [note on process I/O](#) for more information.

`process.stdout.fd`

- `<number>`

This property refers to the value of underlying file descriptor of `process.stdout`. The value is fixed at `1`. In `Worker` threads, this field does not exist.

A note on process I/O

`process.stdout` and `process.stderr` differ from other Node.js streams in important ways:

1. They are used internally by `console.log()` and `console.error()`, respectively.
2. Writes may be synchronous depending on what the stream is connected to and whether the system is Windows or POSIX:
 - Files: *synchronous* on Windows and POSIX
 - TTYs (Terminals): *asynchronous* on Windows, *synchronous* on POSIX
 - Pipes (and sockets): *synchronous* on Windows, *asynchronous* on POSIX

These behaviors are partly for historical reasons, as changing them would create backward incompatibility, but they are also expected by some users.

Synchronous writes avoid problems such as output written with `console.log()` or `console.error()` being unexpectedly interleaved, or not written at all if `process.exit()` is called before an asynchronous write completes. See [process.exit\(\)](#) for more information.

Warning: Synchronous writes block the event loop until the write has completed. This can be near instantaneous in the case of output to a file, but under high system load, pipes that are not being read at the receiving end, or with slow terminals or file systems, its possible for the event loop to be blocked often enough and long enough to have severe negative performance impacts. This may not be a problem when writing to an interactive terminal session, but consider this particularly careful when doing production logging to the process output streams.

To check if a stream is connected to a `TTY` context, check the `isTTY` property.

For instance:

```
$ node -p "Boolean(process.stdin.isTTY)"
true
$ echo "foo" | node -p "Boolean(process.stdin.isTTY)"
false
$ node -p "Boolean(process.stdout.isTTY)"
true
$ node -p "Boolean(process.stdout.isTTY)" | cat
false
```

See the [TTY](#) documentation for more information.

process.throwDeprecation

- `<boolean>`

The initial value of `process.throwDeprecation` indicates whether the `--throw-deprecation` flag is set on the current Node.js process. `process.throwDeprecation` is mutable, so whether or not deprecation warnings result in errors may be altered at runtime. See the documentation for the `'warning'` event and the `emitWarning()` method for more information.

```
$ node --throw-deprecation -p "process.throwDeprecation"
true
$ node -p "process.throwDeprecation"
undefined
$ node
> process.emitWarning('test', 'DeprecationWarning');
undefined
> (node:26598) DeprecationWarning: test
> process.throwDeprecation = true;
true
> process.emitWarning('test', 'DeprecationWarning');
Thrown:
[DeprecationWarning: test] { name: 'DeprecationWarning' }
```

process.title

- `<string>`

The `process.title` property returns the current process title (i.e. returns the current value of `ps`). Assigning a new value to `process.title` modifies the current value of `ps`.

When a new value is assigned, different platforms will impose different maximum length restrictions on the title. Usually such restrictions are quite limited. For instance, on Linux and macOS, `process.title` is limited to the size of the binary name plus the length of the command-line arguments because setting the `process.title` overwrites the `argv` memory of the process. Node.js v0.8 allowed for longer process title strings by also overwriting the `environ` memory but that was potentially insecure and confusing in some (rather obscure) cases.

Assigning a value to `process.title` might not result in an accurate label within process manager applications such as macOS Activity Monitor or Windows Services Manager.

process.traceDeprecation

- `<boolean>`

The `process.traceDeprecation` property indicates whether the `--trace-deprecation` flag is set on the current Node.js process. See the documentation for the `'warning' event` and the `emitWarning()` method for more information about this flag's behavior.

process.umask()

Stability: 0 - Deprecated. Calling `process.umask()` with no argument causes the process-wide umask to be written twice. This introduces a race condition between threads, and is a potential security vulnerability. There is no safe, cross-platform alternative API.

`process.umask()` returns the Node.js process's file mode creation mask. Child processes inherit the mask from the parent process.

process.umask(mask)

- `mask <string> | <integer>`

`process.umask(mask)` sets the Node.js process's file mode creation mask. Child processes inherit the mask from the parent process. Returns the previous mask.

```
import { umask } from 'process';

const newmask = 0o022;
const oldmask = umask(newmask);
console.log(
  `Changed umask from ${oldmask.toString(8)} to ${newmask.toString(8)}`
);const { umask } = require('process');

const newmask = 0o022;
const oldmask = umask(newmask);
console.log(
  `Changed umask from ${oldmask.toString(8)} to ${newmask.toString(8)}`
);
```

In `worker` threads, `process.umask(mask)` will throw an exception.

process.uptime()

- Returns: `<number>`

The `process.uptime()` method returns the number of seconds the current Node.js process has been running.

The return value includes fractions of a second. Use `Math.floor()` to get whole seconds.

process.version

- `<string>`

The `process.version` property contains the Node.js version string.

```
import { version } from 'process';

console.log(`Version: ${version}`);
// Version: v14.8.0const { version } = require('process');

console.log(`Version: ${version}`);
// Version: v14.8.0
```

To get the version string without the prepended v, use `process.versions.node`.

process.versions

- `<Object>`

The `process.versions` property returns an object listing the version strings of Node.js and its dependencies. `process.versions.modules` indicates the current ABI version, which is increased whenever a C++ API changes. Node.js will refuse to load modules that were compiled against a different module ABI version.

```
import { versions } from 'process';

console.log(versions);const { versions } = require('process');

console.log(versions);
```

Will generate an object similar to:

```
{ node: '11.13.0',
  v8: '7.0.276.38-node.18',
  uv: '1.27.0',
  zlib: '1.2.11',
  brotli: '1.0.7',
  ares: '1.15.0',
  modules: '67',
  nghttp2: '1.34.0',
  napi: '4',
  llhttp: '1.1.1',
  openssl: '1.1.1b',
  cldr: '34.0',
  icu: '63.1',
  tz: '2018e',
  unicode: '11.0' }
```

Exit codes

Node.js will normally exit with a `0` status code when no more async operations are pending. The following status codes are used in other cases:

- **1 Uncaught Fatal Exception:** There was an uncaught exception, and it was not handled by a domain or an `'uncaughtException'` event handler.

- **2** : Unused (reserved by Bash for builtin misuse)
- **3 Internal JavaScript Parse Error**: The JavaScript source code internal in the Node.js bootstrapping process caused a parse error. This is extremely rare, and generally can only happen during development of Node.js itself.
- **4 Internal JavaScript Evaluation Failure**: The JavaScript source code internal in the Node.js bootstrapping process failed to return a function value when evaluated. This is extremely rare, and generally can only happen during development of Node.js itself.
- **5 Fatal Error**: There was a fatal unrecoverable error in V8. Typically a message will be printed to stderr with the prefix `FATAL ERROR`.
- **6 Non-function Internal Exception Handler**: There was an uncaught exception, but the internal fatal exception handler function was somehow set to a non-function, and could not be called.
- **7 Internal Exception Handler Run-Time Failure**: There was an uncaught exception, and the internal fatal exception handler function itself threw an error while attempting to handle it. This can happen, for example, if an `'uncaughtException'` or `domain.on('error')` handler throws an error.
- **8** : Unused. In previous versions of Node.js, exit code 8 sometimes indicated an uncaught exception.
- **9 Invalid Argument**: Either an unknown option was specified, or an option requiring a value was provided without a value.
- **10 Internal JavaScript Run-Time Failure**: The JavaScript source code internal in the Node.js bootstrapping process threw an error when the bootstrapping function was called. This is extremely rare, and generally can only happen during development of Node.js itself.
- **12 Invalid Debug Argument**: The `--inspect` and/or `--inspect-brk` options were set, but the port number chosen was invalid or unavailable.
- **13 Unfinished Top-Level Await**: `await` was used outside of a function in the top-level code, but the passed `Promise` never resolved.
- **>128 Signal Exits**: If Node.js receives a fatal signal such as `SIGKILL` or `SIGHUP`, then its exit code will be `128` plus the value of the signal code. This is a standard POSIX practice, since exit codes are defined to be 7-bit integers, and signal exits set the high-order bit, and then contain the value of the signal code. For example, signal `SIGABRT` has value `6`, so the expected exit code will be `128 + 6`, or `134`.

Punycode

Stability: 0 - Deprecated

Source Code: [lib/punycode.js](#)

The version of the `punycode` module bundled in Node.js is being deprecated. In a future major version of Node.js this module will be removed. Users currently depending on the `punycode` module should switch to using the userland-provided `Punycode.js` module instead. For punycode-based URL encoding, see `url.domainToASCII` or, more generally, the [WHATWG URL API](#).

The `punycode` module is a bundled version of the `Punycode.js` module. It can be accessed using:

```
const punycode = require('punycode');
```

`Punycode` is a character encoding scheme defined by RFC 3492 that is primarily intended for use in Internationalized Domain Names. Because host names in URLs are limited to ASCII characters only, Domain Names that contain non-ASCII characters must be converted into ASCII using the Punycode scheme. For instance, the Japanese character that translates into the English word, `'example'` is `'例'`. The Internationalized Domain Name, `'例.com'` (equivalent to `'example.com'`) is represented by Punycode as the ASCII string `'xn--fsq.com'`.

The `punycode` module provides a simple implementation of the Punycode standard.

The `punycode` module is a third-party dependency used by Node.js and made available to developers as a convenience. Fixes or other modifications to the module must be directed to the [Punycode.js](#) project.

`punycode.decode(string)`

- `string` `<string>`

The `punycode.decode()` method converts a **Punycode** string of ASCII-only characters to the equivalent string of Unicode codepoints.

```
punycode.decode('maana-pta'); // 'mañana'
punycode.decode('--dqq34k'); // '👤-👤'
```

punycode.encode(string)

- `string` `<string>`

The `punycode.encode()` method converts a string of Unicode codepoints to a **Punycode** string of ASCII-only characters.

```
punycode.encode('mañana'); // 'maana-pta'
punycode.encode('👤-👤'); // '--dqq34k'
```

punycode.toASCII(domain)

- `domain` `<string>`

The `punycode.toASCII()` method converts a Unicode string representing an Internationalized Domain Name to **Punycode**. Only the non-ASCII parts of the domain name will be converted. Calling `punycode.toASCII()` on a string that already only contains ASCII characters will have no effect.

```
// encode domain names
punycode.toASCII('mañana.com'); // 'xn--maana-pta.com'
punycode.toASCII('👤-👤.com'); // 'xn---dqq34k.com'
punycode.toASCII('example.com'); // 'example.com'
```

punycode.toUnicode(domain)

- `domain` `<string>`

The `punycode.toUnicode()` method converts a string representing a domain name containing **Punycode** encoded characters into Unicode. Only the **Punycode** encoded parts of the domain name are be converted.

```
// decode domain names
punycode.toUnicode('xn--maana-pta.com'); // 'mañana.com'
punycode.toUnicode('xn---dqq34k.com'); // '👤-👤.com'
punycode.toUnicode('example.com'); // 'example.com'
```

punycode.ucs2

punycode.ucs2.decode(string)

- `string` `<string>`

The `punycode.ucs2.decode()` method returns an array containing the numeric codepoint values of each Unicode symbol in the string.


```
punycode.ucs2.decode('abc'); // [0x61, 0x62, 0x63]
// surrogate pair for U+1D306 tetragram for centre:
punycode.ucs2.decode('\uD834\uDF06'); // [0x1D306]
```

punycode.ucs2.encode(codePoints)

- `codePoints` `<integer[]>`

The `punycode.ucs2.encode()` method returns a string based on an array of numeric code point values.

```
punycode.ucs2.encode([0x61, 0x62, 0x63]); // 'abc'
punycode.ucs2.encode([0x1D306]); // '\uD834\uDF06'
```

punycode.version

- `<string>`

Returns a string identifying the current `Punycode.js` version number.

Query string

Stability: 3 - Legacy

Source Code: [lib/querystring.js](#)

The `querystring` module provides utilities for parsing and formatting URL query strings. It can be accessed using:

```
const querystring = require('querystring');
```

The `querystring` API is considered Legacy. While it is still maintained, new code should use the `<URLSearchParams>` API instead.

querystring.decode()

The `querystring.decode()` function is an alias for `querystring.parse()`.

querystring.encode()

The `querystring.encode()` function is an alias for `querystring.stringify()`.

querystring.escape(str)

- `str` `<string>`

The `querystring.escape()` method performs URL percent-encoding on the given `str` in a manner that is optimized for the specific requirements of URL query strings.

The `querystring.escape()` method is used by `querystring.stringify()` and is generally not expected to be used directly. It is exported primarily to allow application code to provide a replacement percent-encoding implementation if necessary by assigning

`querystring.escape` to an alternative function.

`querystring.parse(str[, sep[, eq[, options]]])`

- `str` `<string>` The URL query string to parse
- `sep` `<string>` The substring used to delimit key and value pairs in the query string. **Default:** `'&'`.
- `eq` `<string>` . The substring used to delimit keys and values in the query string. **Default:** `'='`.
- `options` `<Object>`
 - `decodeURIComponent` `<Function>` The function to use when decoding percent-encoded characters in the query string. **Default:** `querystring.unescape()`.
 - `maxKeys` `<number>` Specifies the maximum number of keys to parse. Specify `0` to remove key counting limitations. **Default:** `1000`.

The `querystring.parse()` method parses a URL query string (`str`) into a collection of key and value pairs.

For example, the query string `'foo=bar&abc=xyz&abc=123'` is parsed into:

```
{
  foo: 'bar',
  abc: ['xyz', '123']
}
```

The object returned by the `querystring.parse()` method *does not* prototypically inherit from the JavaScript `Object`. This means that typical `Object` methods such as `obj.toString()`, `obj.hasOwnProperty()`, and others are not defined and *will not work*.

By default, percent-encoded characters within the query string will be assumed to use UTF-8 encoding. If an alternative character encoding is used, then an alternative `decodeURIComponent` option will need to be specified:

```
// Assuming gbkDecodeURIComponent function already exists...

querystring.parse('w=%D6%D0%CE%C4&foo=bar', null, null,
  { decodeURIComponent: gbkDecodeURIComponent });
```

`querystring.stringify(obj[, sep[, eq[, options]]])`

- `obj` `<Object>` The object to serialize into a URL query string
- `sep` `<string>` The substring used to delimit key and value pairs in the query string. **Default:** `'&'`.
- `eq` `<string>` . The substring used to delimit keys and values in the query string. **Default:** `'='`.
- `options`
 - `encodeURIComponent` `<Function>` The function to use when converting URL-unsafe characters to percent-encoding in the query string. **Default:** `querystring.escape()`.

The `querystring.stringify()` method produces a URL query string from a given `obj` by iterating through the object's "own properties".

It serializes the following types of values passed in `obj`: `<string>` | `<number>` | `<bigint>` | `<boolean>` | `<string[]>` | `<number[]>` | `<bigint[]>` | `<boolean[]>` The numeric values must be finite. Any other input values will be coerced to empty strings.

```
querystring.stringify({ foo: 'bar', baz: ['qux', 'quux'], corge: '' });
// Returns 'foo=bar&baz=qux&baz=quux&corge='
```

```
querystring.stringify({ foo: 'bar', baz: 'qux' }, ';', ':');  
// Returns 'foo:bar;baz:qux'
```

By default, characters requiring percent-encoding within the query string will be encoded as UTF-8. If an alternative encoding is required, then an alternative `encodeURIComponent` option will need to be specified:

```
// Assuming gbkEncodeURIComponent function already exists,  
  
querystring.stringify({ w: '中文', foo: 'bar' }, null, null,  
    { encodeURIComponent: gbkEncodeURIComponent }));
```

querystring.unescape(str)

- `str` `<string>`

The `querystring.unescape()` method performs decoding of URL percent-encoded characters on the given `str`.

The `querystring.unescape()` method is used by `querystring.parse()` and is generally not expected to be used directly. It is exported primarily to allow application code to provide a replacement decoding implementation if necessary by assigning `querystring.unescape` to an alternative function.

By default, the `querystring.unescape()` method will attempt to use the JavaScript built-in `decodeURIComponent()` method to decode. If that fails, a safer equivalent that does not throw on malformed URLs will be used.

Readline

Stability: 2 - Stable

Source Code: [lib/readline.js](#)

The `readline` module provides an interface for reading data from a `Readable` stream (such as `process.stdin`) one line at a time. It can be accessed using:

```
const readline = require('readline');
```

The following simple example illustrates the basic use of the `readline` module.

```
const readline = require('readline');  
  
const rl = readline.createInterface({  
    input: process.stdin,  
    output: process.stdout  
});  
  
rl.question('What do you think of Node.js? ', (answer) => {  
    // TODO: Log the answer in a database  
    console.log(`Thank you for your valuable feedback: ${answer}`);  
});
```

```
rl.close();
});
```

Once this code is invoked, the Node.js application will not terminate until the `readline.Interface` is closed because the interface waits for data to be received on the `input` stream.

Class: Interface

- Extends: `<EventEmitter>`

Instances of the `readline.Interface` class are constructed using the `readline.createInterface()` method. Every instance is associated with a single `input` `Readable` stream and a single `output` `Writable` stream. The `output` stream is used to print prompts for user input that arrives on, and is read from, the `input` stream.

Event: 'close'

The `'close'` event is emitted when one of the following occur:

- The `rl.close()` method is called and the `readline.Interface` instance has relinquished control over the `input` and `output` streams;
- The `input` stream receives its `'end'` event;
- The `input` stream receives `Ctrl + D` to signal end-of-transmission (EOT);
- The `input` stream receives `Ctrl + C` to signal `SIGINT` and there is no `'SIGINT'` event listener registered on the `readline.Interface` instance.

The listener function is called without passing any arguments.

The `readline.Interface` instance is finished once the `'close'` event is emitted.

Event: 'line'

The `'line'` event is emitted whenever the `input` stream receives an end-of-line input (`\n`, `\r`, or `\r\n`). This usually occurs when the user presses `Enter` or `Return`.

The listener function is called with a string containing the single line of received input.

```
rl.on('line', (input) => {
  console.log(`Received: ${input}`);
});
```

Event: 'history'

The `'history'` event is emitted whenever the history array has changed.

The listener function is called with an array containing the history array. It will reflect all changes, added lines and removed lines due to `historySize` and `removeHistoryDuplicates`.

The primary purpose is to allow a listener to persist the history. It is also possible for the listener to change the history object. This could be useful to prevent certain lines to be added to the history, like a password.

```
rl.on('history', (history) => {
  console.log(`Received: ${history}`);
});
```

Event: 'pause'

The 'pause' event is emitted when one of the following occur:

- The `input` stream is paused.
- The `input` stream is not paused and receives the 'SIGCONT' event. (See events 'SIGTSTP' and 'SIGCONT' .)

The listener function is called without passing any arguments.

```
rl.on('pause', () => {  
  console.log('Readline paused.');
```

```
});
```

Event: 'resume'

The 'resume' event is emitted whenever the `input` stream is resumed.

The listener function is called without passing any arguments.

```
rl.on('resume', () => {  
  console.log('Readline resumed.');
```

```
});
```

Event: 'SIGCONT'

The 'SIGCONT' event is emitted when a Node.js process previously moved into the background using `Ctrl + Z` (i.e. `SIGTSTP`) is then brought back to the foreground using `fg(1p)` .

If the `input` stream was paused *before* the `SIGTSTP` request, this event will not be emitted.

The listener function is invoked without passing any arguments.

```
rl.on('SIGCONT', () => {  
  // `prompt` will automatically resume the stream  
  rl.prompt();  
});
```

The 'SIGCONT' event is *not* supported on Windows.

Event: 'SIGINT'

The 'SIGINT' event is emitted whenever the `input` stream receives a `ctrl+c` input, known typically as `SIGINT` . If there are no 'SIGINT' event listeners registered when the `input` stream receives a `SIGINT` , the 'pause' event will be emitted.

The listener function is invoked without passing any arguments.

```
rl.on('SIGINT', () => {  
  rl.question('Are you sure you want to exit? ', (answer) => {  
    if (answer.match(/^y(es)?$/i)) rl.pause();  
  });  
});
```

Event: 'SIGTSTP'

The 'SIGTSTP' event is emitted when the `input` stream receives a `ctrl+z` input, typically known as `SIGTSTP`. If there are no 'SIGTSTP' event listeners registered when the `input` stream receives a `SIGTSTP`, the Node.js process will be sent to the background.

When the program is resumed using `fg(1p)`, the 'pause' and 'SIGCONT' events will be emitted. These can be used to resume the `input` stream.

The 'pause' and 'SIGCONT' events will not be emitted if the `input` was paused before the process was sent to the background.

The listener function is invoked without passing any arguments.

```
rl.on('SIGTSTP', () => {
  // This will override SIGTSTP and prevent the program from going to the
  // background.
  console.log('Caught SIGTSTP.');
```

The 'SIGTSTP' event is *not* supported on Windows.

rl.close()

The `rl.close()` method closes the `readline.Interface` instance and relinquishes control over the `input` and `output` streams. When called, the 'close' event will be emitted.

Calling `rl.close()` does not immediately stop other events (including 'line') from being emitted by the `readline.Interface` instance.

rl.pause()

The `rl.pause()` method pauses the `input` stream, allowing it to be resumed later if necessary.

Calling `rl.pause()` does not immediately pause other events (including 'line') from being emitted by the `readline.Interface` instance.

rl.prompt([preserveCursor])

- `preserveCursor` `<boolean>` If `true`, prevents the cursor placement from being reset to 0.

The `rl.prompt()` method writes the `readline.Interface` instances configured `prompt` to a new line in `output` in order to provide a user with a new location at which to provide input.

When called, `rl.prompt()` will resume the `input` stream if it has been paused.

If the `readline.Interface` was created with `output` set to `null` or `undefined` the prompt is not written.

rl.question(query[, options], callback)

- `query` `<string>` A statement or query to write to `output`, prepended to the prompt.
- `options` `<Object>`
 - `signal` `<AbortSignal>` Optionally allows the `question()` to be canceled using an `AbortController`.
- `callback` `<Function>` A callback function that is invoked with the user's input in response to the `query`.

The `rl.question()` method displays the `query` by writing it to the `output`, waits for user input to be provided on `input`, then invokes the `callback` function passing the provided input as the first argument.

When called, `rl.question()` will resume the `input` stream if it has been paused.

If the `readline.Interface` was created with `output` set to `null` or `undefined` the query is not written.

The `callback` function passed to `rl.question()` does not follow the typical pattern of accepting an `Error` object or `null` as the first argument. The `callback` is called with the provided answer as the only argument.

Example usage:

```
rl.question('What is your favorite food? ', (answer) => {
  console.log(`Oh, so your favorite food is ${answer}`);
});
```

Using an `AbortController` to cancel a question.

```
const ac = new AbortController();
const signal = ac.signal;

rl.question('What is your favorite food? ', { signal }, (answer) => {
  console.log(`Oh, so your favorite food is ${answer}`);
});

signal.addEventListener('abort', () => {
  console.log('The food question timed out');
}, { once: true });

setTimeout(() => ac.abort(), 10000);
```

If this method is invoked as it's `util.promisify()`ed version, it returns a `Promise` that fulfills with the answer. If the question is canceled using an `AbortController` it will reject with an `AbortError`.

```
const util = require('util');
const question = util.promisify(rl.question).bind(rl);

async function questionExample() {
  try {
    const answer = await question('What is your favorite food? ');
    console.log(`Oh, so your favorite food is ${answer}`);
  } catch (err) {
    console.error('Question rejected', err);
  }
}

questionExample();
```

`rl.resume()`

The `rl.resume()` method resumes the `input` stream if it has been paused.

`rl.setPrompt(prompt)`

- `prompt` `<string>`

The `rl.setPrompt()` method sets the prompt that will be written to `output` whenever `rl.prompt()` is called.

`rl.getPrompt()`

- Returns: `<string>` the current prompt string

The `rl.getPrompt()` method returns the current prompt used by `rl.prompt()`.

`rl.write(data[, key])`

- `data` `<string>`
- `key` `<Object>`
 - `ctrl` `<boolean>` `true` to indicate the `Ctrl` key.
 - `meta` `<boolean>` `true` to indicate the `Meta` key.
 - `shift` `<boolean>` `true` to indicate the `Shift` key.
 - `name` `<string>` The name of the a key.

The `rl.write()` method will write either `data` or a key sequence identified by `key` to the `output`. The `key` argument is supported only if `output` is a TTY text terminal. See [TTY keybindings](#) for a list of key combinations.

If `key` is specified, `data` is ignored.

When called, `rl.write()` will resume the `input` stream if it has been paused.

If the `readline.Interface` was created with `output` set to `null` or `undefined` the `data` and `key` are not written.

```
rl.write('Delete this!');
// Simulate Ctrl+U to delete the line written previously
rl.write(null, { ctrl: true, name: 'u' });
```

The `rl.write()` method will write the data to the `readline` Interface's `input` as if it were provided by the user.

`rl[Symbol.asyncIterator]()`

- Returns: `<AsyncIterator>`

Create an `AsyncIterator` object that iterates through each line in the input stream as a string. This method allows asynchronous iteration of `readline.Interface` objects through `for await...of` loops.

Errors in the input stream are not forwarded.

If the loop is terminated with `break`, `throw`, or `return`, `rl.close()` will be called. In other words, iterating over a `readline.Interface` will always consume the input stream fully.

Performance is not on par with the traditional `'line'` event API. Use `'line'` instead for performance-sensitive applications.

```
async function processLineByLine() {
  const rl = readline.createInterface({
    // ...
  });

  for await (const line of rl) {
    // Each line in the readline input will be successively available here as
    // `line`.
  }
}
```



```
}  
}
```

`readline.createInterface()` will start to consume the input stream once invoked. Having asynchronous operations between interface creation and asynchronous iteration may result in missed lines.

`rl.line`

- `<string>`

The current input data being processed by node.

This can be used when collecting input from a TTY stream to retrieve the current value that has been processed thus far, prior to the `line` event being emitted. Once the `line` event has been emitted, this property will be an empty string.

Be aware that modifying the value during the instance runtime may have unintended consequences if `rl.cursor` is not also controlled.

If not using a TTY stream for input, use the `'line'` event.

One possible use case would be as follows:

```
const values = ['lorem ipsum', 'dolor sit amet'];  
const rl = readline.createInterface(process.stdin);  
const showResults = debounce(() => {  
  console.log(  
    '\n',  
    values.filter((val) => val.startsWith(rl.line)).join(' '),  
  );  
}, 300);  
process.stdin.on('keypress', (c, k) => {  
  showResults();  
});
```

`rl.cursor`

- `<number>` | `<undefined>`

The cursor position relative to `rl.line`.

This will track where the current cursor lands in the input string, when reading input from a TTY stream. The position of cursor determines the portion of the input string that will be modified as input is processed, as well as the column where the terminal caret will be rendered.

`rl.getCursorPos()`

- Returns: `<Object>`
 - `rows` `<number>` the row of the prompt the cursor currently lands on
 - `cols` `<number>` the screen column the cursor currently lands on

Returns the real position of the cursor in relation to the input prompt + string. Long input (wrapping) strings, as well as multiple line prompts are included in the calculations.

`readline.clearLine(stream, dir[, callback])`

- `stream` `<stream.Writable>`

- `dir` `<number>`
 - `-1`: to the left from cursor
 - `1`: to the right from cursor
 - `0`: the entire line
- `callback` `<Function>` Invoked once the operation completes.
- Returns: `<boolean>` `false` if `stream` wishes for the calling code to wait for the `'drain'` event to be emitted before continuing to write additional data; otherwise `true`.

The `readline.clearLine()` method clears current line of given `TTY` stream in a specified direction identified by `dir`.

readline.clearScreenDown(stream[, callback])

- `stream` `<stream.Writable>`
- `callback` `<Function>` Invoked once the operation completes.
- Returns: `<boolean>` `false` if `stream` wishes for the calling code to wait for the `'drain'` event to be emitted before continuing to write additional data; otherwise `true`.

The `readline.clearScreenDown()` method clears the given `TTY` stream from the current position of the cursor down.

readline.createInterface(options)

- `options` `<Object>`
 - `input` `<stream.Readable>` The `Readable` stream to listen to. This option is *required*.
 - `output` `<stream.Writable>` The `Writable` stream to write readline data to.
 - `completer` `<Function>` An optional function used for Tab autocompletion.
 - `terminal` `<boolean>` `true` if the `input` and `output` streams should be treated like a TTY, and have ANSI/VT100 escape codes written to it. **Default:** checking `isTTY` on the `output` stream upon instantiation.
 - `history` `<string[]>` Initial list of history lines. This option makes sense only if `terminal` is set to `true` by the user or by an internal `output` check, otherwise the history caching mechanism is not initialized at all. **Default:** `[]`.
 - `historySize` `<number>` Maximum number of history lines retained. To disable the history set this value to `0`. This option makes sense only if `terminal` is set to `true` by the user or by an internal `output` check, otherwise the history caching mechanism is not initialized at all. **Default:** `30`.
 - `removeHistoryDuplicates` `<boolean>` If `true`, when a new input line added to the history list duplicates an older one, this removes the older line from the list. **Default:** `false`.
 - `prompt` `<string>` The prompt string to use. **Default:** `'> '`.
 - `crlfDelay` `<number>` If the delay between `\r` and `\n` exceeds `crlfDelay` milliseconds, both `\r` and `\n` will be treated as separate end-of-line input. `crlfDelay` will be coerced to a number no less than `100`. It can be set to `Infinity`, in which case `\r` followed by `\n` will always be considered a single newline (which may be reasonable for `reading files` with `\r\n` line delimiter). **Default:** `100`.
 - `escapeCodeTimeout` `<number>` The duration `readline` will wait for a character (when reading an ambiguous key sequence in milliseconds one that can both form a complete key sequence using the input read so far and can take additional input to complete a longer key sequence). **Default:** `500`.
 - `tabSize` `<integer>` The number of spaces a tab is equal to (minimum 1). **Default:** `8`.
 - `signal` `<AbortSignal>` Allows closing the interface using an `AbortSignal`. Aborting the signal will internally call `close` on the interface.
- Returns: `<readline.Interface>`

The `readline.createInterface()` method creates a new `readline.Interface` instance.

```
const readline = require('readline');
const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});
```

Once the `readline.Interface` instance is created, the most common case is to listen for the `'line'` event:

```
rl.on('line', (line) => {
  console.log(`Received: ${line}`);
});
```

If `terminal` is `true` for this instance then the `output` stream will get the best compatibility if it defines an `output.columns` property and emits a `'resize'` event on the `output` if or when the columns ever change (`process.stdout` does this automatically when it is a TTY).

When creating a `readline.Interface` using `stdin` as input, the program will not terminate until it receives EOF (`Ctrl + D` on Linux/macOS, `Ctrl + Z` followed by `Return` on Windows). If you want your application to exit without waiting for user input, you can `unref()` the standard input stream:

```
process.stdin.unref();
```

Use of the `completer` function

The `completer` function takes the current line entered by the user as an argument, and returns an `Array` with 2 entries:

- An `Array` with matching entries for the completion.
- The substring that was used for the matching.

For instance: `[[substr1, substr2, ...], originalsubstring]`.

```
function completer(line) {
  const completions = '.help .error .exit .quit .q'.split(' ');
  const hits = completions.filter((c) => c.startsWith(line));
  // Show all completions if none found
  return [hits.length ? hits : completions, line];
}
```

The `completer` function can be called asynchronously if it accepts two arguments:

```
function completer(linePartial, callback) {
  callback(null, [['123'], linePartial]);
}
```

`readline.cursorTo(stream, x[, y][, callback])`

- `stream` `<stream.Writable>`
- `x` `<number>`
- `y` `<number>`

- `callback` `<Function>` Invoked once the operation completes.
- Returns: `<boolean>` `false` if `stream` wishes for the calling code to wait for the `'drain'` event to be emitted before continuing to write additional data; otherwise `true`.

The `readline.cursorTo()` method moves cursor to the specified position in a given `TTY` `stream`.

readline.emitKeypressEvents(stream[, interface])

- `stream` `<stream.Readable>`
- `interface` `<readline.Interface>`

The `readline.emitKeypressEvents()` method causes the given `Readable` stream to begin emitting `'keypress'` events corresponding to received input.

Optionally, `interface` specifies a `readline.Interface` instance for which autocompletion is disabled when copy-pasted input is detected.

If the `stream` is a `TTY`, then it must be in raw mode.

This is automatically called by any readline instance on its `input` if the `input` is a terminal. Closing the `readline` instance does not stop the `input` from emitting `'keypress'` events.

```
readline.emitKeypressEvents(process.stdin);
if (process.stdin.isTTY)
  process.stdin.setRawMode(true);
```

readline.moveCursor(stream, dx, dy[, callback])

- `stream` `<stream.Writable>`
- `dx` `<number>`
- `dy` `<number>`
- `callback` `<Function>` Invoked once the operation completes.
- Returns: `<boolean>` `false` if `stream` wishes for the calling code to wait for the `'drain'` event to be emitted before continuing to write additional data; otherwise `true`.

The `readline.moveCursor()` method moves the cursor *relative* to its current position in a given `TTY` `stream`.

Example: Tiny CLI

The following example illustrates the use of `readline.Interface` class to implement a small command-line interface:

```
const readline = require('readline');
const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout,
  prompt: 'OHAI> '
});

rl.prompt();

rl.on('line', (line) => {
```

```

switch (line.trim()) {
  case 'hello':
    console.log('world!');
    break;
  default:
    console.log(`Say what? I might have heard '${line.trim()}'`);
    break;
}
rl.prompt();
}).on('close', () => {
  console.log('Have a great day!');
  process.exit(0);
});

```

Example: Read file stream line-by-Line

A common use case for `readline` is to consume an input file one line at a time. The easiest way to do so is leveraging the `fs.ReadStream` API as well as a `for await...of` loop:

```

const fs = require('fs');
const readline = require('readline');

async function processLineByLine() {
  const fileStream = fs.createReadStream('input.txt');

  const rl = readline.createInterface({
    input: fileStream,
    crlfDelay: Infinity
  });
  // Note: we use the crlfDelay option to recognize all instances of CR LF
  // ('\r\n') in input.txt as a single line break.

  for await (const line of rl) {
    // Each line in input.txt will be successively available here as `line`.
    console.log(`Line from file: ${line}`);
  }
}

processLineByLine();

```

Alternatively, one could use the `'line'` event:

```

const fs = require('fs');
const readline = require('readline');

const rl = readline.createInterface({
  input: fs.createReadStream('sample.txt'),
  crlfDelay: Infinity
});

```

```
rl.on('line', (line) => {
  console.log(`Line from file: ${line}`);
});
```

Currently, `for await...of` loop can be a bit slower. If `async` / `await` flow and speed are both essential, a mixed approach can be applied:

```
const { once } = require('events');
const { createReadStream } = require('fs');
const { createInterface } = require('readline');

(async function processLineByLine() {
  try {
    const rl = createInterface({
      input: createReadStream('big-file.txt'),
      crlfDelay: Infinity
    });

    rl.on('line', (line) => {
      // Process the line.
    });

    await once(rl, 'close');

    console.log('File processed.');
  } catch (err) {
    console.error(err);
  }
})();
```

TTY keybindings

Keybindings	Description	Notes
Ctrl + Shift + Backspace	Delete line left	Doesn't work on Linux, Mac and Windows
Ctrl + Shift + Delete	Delete line right	Doesn't work on Mac
Ctrl + C	Emit <code>SIGINT</code> or close the readline instance	
Ctrl + H	Delete left	
Ctrl + D	Delete right or close the readline instance in case the current line is empty / EOF	Doesn't work on Windows
Ctrl + U	Delete from the current position to the line start	
Ctrl + K	Delete from the current position to the end of line	
Ctrl + A	Go to start of line	
Ctrl + E	Go to to end of line	

Ctrl + B	Back one character	
Ctrl + F	Forward one character	
Ctrl + L	Clear screen	
Ctrl + N	Next history item	
Ctrl + P	Previous history item	
Ctrl + Z	Moves running process into background. Type <code>fg</code> and press Enter to return.	Doesn't work on Windows
Ctrl + W or Ctrl + Backspace	Delete backward to a word boundary	Ctrl + Backspace Doesn't work on Linux, Mac and Windows
Ctrl + Delete	Delete forward to a word boundary	Doesn't work on Mac
Ctrl + Left arrow or Meta + B	Word left	Ctrl + Left arrow Doesn't work on Mac
Ctrl + Right arrow or Meta + F	Word right	Ctrl + Right arrow Doesn't work on Mac
Meta + D or Meta + Delete	Delete word right	Meta + Delete Doesn't work on windows
Meta + Backspace	Delete word left	Doesn't work on Mac

REPL

Stability: 2 - Stable

Source Code: [lib/repl.js](#)

The `repl` module provides a Read-Eval-Print-Loop (REPL) implementation that is available both as a standalone program or includible in other applications. It can be accessed using:

```
const repl = require('repl');
```

Design and features

The `repl` module exports the `repl.REPLServer` class. While running, instances of `repl.REPLServer` will accept individual lines of user input, evaluate those according to a user-defined evaluation function, then output the result. Input and output may be from `stdin` and `stdout`, respectively, or may be connected to any Node.js `stream`.

Instances of `repl.REPLServer` support automatic completion of inputs, completion preview, simplistic Emacs-style line editing, multi-line inputs, `ZSH`-like reverse-i-search, `ZSH`-like substring-based history search, ANSI-styled output, saving and restoring current REPL session state, error recovery, and customizable evaluation functions. Terminals that do not support ANSI styles and Emacs-style line editing automatically fall back to a limited feature set.

Commands and special keys

The following special commands are supported by all REPL instances:

- `.break` : When in the process of inputting a multi-line expression, enter the `.break` command (or press `Ctrl + C`) to abort further input or processing of that expression.
- `.clear` : Resets the REPL `context` to an empty object and clears any multi-line expression being input.
- `.exit` : Close the I/O stream, causing the REPL to exit.
- `.help` : Show this list of special commands.
- `.save` : Save the current REPL session to a file: `> .save ./file/to/save.js`
- `.load` : Load a file into the current REPL session. `> .load ./file/to/load.js`
- `.editor` : Enter editor mode (`Ctrl + D` to finish, `Ctrl + C` to cancel).

```
> .editor
// Entering editor mode (^D to finish, ^C to cancel)
function welcome(name) {
  return `Hello ${name}!`;
}

welcome('Node.js User');

// ^D
'Hello Node.js User!'
>
```

The following key combinations in the REPL have these special effects:

- `Ctrl + C` : When pressed once, has the same effect as the `.break` command. When pressed twice on a blank line, has the same effect as the `.exit` command.
- `Ctrl + D` : Has the same effect as the `.exit` command.
- `Tab` : When pressed on a blank line, displays global and local (scope) variables. When pressed while entering other input, displays relevant autocompletion options.

For key bindings related to the reverse-i-search, see [reverse-i-search](#) . For all other key bindings, see [TTY keybindings](#) .

Default evaluation

By default, all instances of `repl.REPLServer` use an evaluation function that evaluates JavaScript expressions and provides access to Node.js built-in modules. This default behavior can be overridden by passing in an alternative evaluation function when the `repl.REPLServer` instance is created.

JavaScript expressions

The default evaluator supports direct evaluation of JavaScript expressions:

```
> 1 + 1
2
> const m = 2
undefined
> m + 1
3
```


Unless otherwise scoped within blocks or functions, variables declared either implicitly or using the `const`, `let`, or `var` keywords are declared at the global scope.

Global and local scope

The default evaluator provides access to any variables that exist in the global scope. It is possible to expose a variable to the REPL explicitly by assigning it to the `context` object associated with each `REPLServer`:

```
const repl = require('repl');
const msg = 'message';

repl.start('> ').context.m = msg;
```

Properties in the `context` object appear as local within the REPL:

```
$ node repl_test.js
> m
'message'
```

Context properties are not read-only by default. To specify read-only globals, context properties must be defined using `Object.defineProperty()`:

```
const repl = require('repl');
const msg = 'message';

const r = repl.start('> ');
Object.defineProperty(r.context, 'm', {
  configurable: false,
  enumerable: true,
  value: msg
});
```

Accessing core Node.js modules

The default evaluator will automatically load Node.js core modules into the REPL environment when used. For instance, unless otherwise declared as a global or scoped variable, the input `fs` will be evaluated on-demand as `global.fs = require('fs')`.

```
> fs.createReadStream('./some/file');
```

Global uncaught exceptions

The REPL uses the `domain` module to catch all uncaught exceptions for that REPL session.

This use of the `domain` module in the REPL has these side effects:

- Uncaught exceptions only emit the `'uncaughtException'` event in the standalone REPL. Adding a listener for this event in a REPL within another Node.js program results in `ERR_INVALID_REPL_INPUT`.

```
const r = repl.start();
```

```

r.write('process.on("uncaughtException", () => console.log("Foobar"));\\n');
// Output stream includes:
//   TypeError [ERR_INVALID_REPL_INPUT]: Listeners for `uncaughtException`
//   cannot be used in the REPL

r.close();

```

- Trying to use `process.setUncaughtExceptionCaptureCallback()` throws an `ERR_DOMAIN_CANNOT_SET_UNCAUGHT_EXCEPTION_CAPTURE` error.

Assignment of the `_` (underscore) variable

The default evaluator will, by default, assign the result of the most recently evaluated expression to the special variable `_` (underscore). Explicitly setting `_` to a value will disable this behavior.

```

> [ 'a', 'b', 'c' ]
[ 'a', 'b', 'c' ]
> _.length
3
> _ += 1
Expression assignment to _ now disabled.
4
> 1 + 1
2
> _
4

```

Similarly, `_error` will refer to the last seen error, if there was any. Explicitly setting `_error` to a value will disable this behavior.

```

> throw new Error('foo');
Error: foo
> _error.message
'foo'

```

await keyword

Support for the `await` keyword is enabled at the top level.

```

> await Promise.resolve(123)
123
> await Promise.reject(new Error('REPL await'))
Error: REPL await
    at repl:1:45
> const timeout = util.promisify(setTimeout);
undefined
> const old = Date.now(); await timeout(1000); console.log(Date.now() - old);
1002
undefined

```

One known limitation of using the `await` keyword in the REPL is that it will invalidate the lexical scoping of the `const` and `let` keywords.

For example:

```
> const m = await Promise.resolve(123)
undefined
> m
123
> const m = await Promise.resolve(234)
undefined
> m
234
```

`--no-experimental-repl-await` shall disable top-level await in REPL.

Reverse-i-search

The REPL supports bi-directional reverse-i-search similar to [ZSH](#). It is triggered with `Ctrl + R` to search backward and `Ctrl + S` to search forwards.

Duplicated history entries will be skipped.

Entries are accepted as soon as any key is pressed that doesn't correspond with the reverse search. Cancelling is possible by pressing `Esc` or `Ctrl + C`.

Changing the direction immediately searches for the next entry in the expected direction from the current position on.

Custom evaluation functions

When a new `repl.REPLServer` is created, a custom evaluation function may be provided. This can be used, for instance, to implement fully customized REPL applications.

The following illustrates a hypothetical example of a REPL that performs translation of text from one language to another:

```
const repl = require('repl');
const { Translator } = require('translator');

const myTranslator = new Translator('en', 'fr');

function myEval(cmd, context, filename, callback) {
  callback(null, myTranslator.translate(cmd));
}

repl.start({ prompt: '> ', eval: myEval });
```

Recoverable errors

At the REPL prompt, pressing `Enter` sends the current line of input to the `eval` function. In order to support multi-line input, the `eval` function can return an instance of `repl.Recoverable` to the provided callback function:

```
function myEval(cmd, context, filename, callback) {
  let result;
  try {
    result = vm.runInThisContext(cmd);
  } catch (err) {
    // ...
  }
  callback(err, result);
}
```

```

    } catch (e) {
      if (isRecoverableError(e)) {
        return callback(new repl.Recoverable(e));
      }
    }
    callback(null, result);
  }

function isRecoverableError(error) {
  if (error.name === 'SyntaxError') {
    return /^(Unexpected end of input|Unexpected token)/.test(error.message);
  }
  return false;
}

```

Customizing REPL output

By default, `repl.REPLServer` instances format output using the `util.inspect()` method before writing the output to the provided `Writable` stream (`process.stdout` by default). The `showProxy` inspection option is set to true by default and the `colors` option is set to true depending on the REPL's `useColors` option.

The `useColors` boolean option can be specified at construction to instruct the default writer to use ANSI style codes to colorize the output from the `util.inspect()` method.

If the REPL is run as standalone program, it is also possible to change the REPL's `inspect defaults` from inside the REPL by using the `inspect.replDefaults` property which mirrors the `defaultOptions` from `util.inspect()`.

```

> util.inspect.replDefaults.compact = false;
false
> [1]
[
  1
]
>

```

To fully customize the output of a `repl.REPLServer` instance pass in a new function for the `writer` option on construction. The following example, for instance, simply converts any input text to upper case:

```

const repl = require('repl');

const r = repl.start({ prompt: '> ', eval: myEval, writer: myWriter });

function myEval(cmd, context, filename, callback) {
  callback(null, cmd);
}

function myWriter(output) {
  return output.toUpperCase();
}

```

Class: REPLServer

- `options` `<Object>` | `<string>` See `repl.start()`
- Extends: `<readline.Interface>`

Instances of `repl.REPLServer` are created using the `repl.start()` method or directly using the JavaScript `new` keyword.

```
const repl = require('repl');

const options = { useColors: true };

const firstInstance = repl.start(options);
const secondInstance = new repl.REPLServer(options);
```

Event: 'exit'

The `'exit'` event is emitted when the REPL is exited either by receiving the `.exit` command as input, the user pressing `ctrl + c` twice to signal `SIGINT`, or by pressing `ctrl + d` to signal `'end'` on the input stream. The listener callback is invoked without any arguments.

```
replServer.on('exit', () => {
  console.log('Received "exit" event from repl!');
  process.exit();
});
```

Event: 'reset'

The `'reset'` event is emitted when the REPL's context is reset. This occurs whenever the `.clear` command is received as input *unless* the REPL is using the default evaluator and the `repl.REPLServer` instance was created with the `useGlobal` option set to `true`. The listener callback will be called with a reference to the `context` object as the only argument.

This can be used primarily to re-initialize REPL context to some pre-defined state:

```
const repl = require('repl');

function initializeContext(context) {
  context.m = 'test';
}

const r = repl.start({ prompt: '> ' });
initializeContext(r.context);

r.on('reset', initializeContext);
```

When this code is executed, the global `'m'` variable can be modified but then reset to its initial value using the `.clear` command:

```
$ ./node example.js
> m
'test'
> m = 1
1
```

```
> m
1
> .clear
Clearing context...
> m
'test'
>
```

replServer.defineCommand(keyword, cmd)

- **keyword** `<string>` The command keyword (*without* a leading `.` character).
- **cmd** `<Object> | <Function>` The function to invoke when the command is processed.

The `replServer.defineCommand()` method is used to add new `.`-prefixed commands to the REPL instance. Such commands are invoked by typing a `.` followed by the `keyword`. The `cmd` is either a `Function` or an `Object` with the following properties:

- **help** `<string>` Help text to be displayed when `.help` is entered (Optional).
- **action** `<Function>` The function to execute, optionally accepting a single string argument.

The following example shows two new commands added to the REPL instance:

```
const repl = require('repl');

const replServer = repl.start({ prompt: '> ' });
replServer.defineCommand('sayhello', {
  help: 'Say hello',
  action(name) {
    this.clearBufferedCommand();
    console.log(`Hello, ${name}!`);
    this.displayPrompt();
  }
});
replServer.defineCommand('saybye', function saybye() {
  console.log('Goodbye!');
  this.close();
});
```

The new commands can then be used from within the REPL instance:

```
> .sayhello Node.js User
Hello, Node.js User!
> .saybye
Goodbye!
```

replServer.displayPrompt([preserveCursor])

- **preserveCursor** `<boolean>`

The `replServer.displayPrompt()` method readies the REPL instance for input from the user, printing the configured `prompt` to a new line in the `output` and resuming the `input` to accept new input.

When multi-line input is being entered, an ellipsis is printed rather than the 'prompt'.

When `preserveCursor` is `true`, the cursor placement will not be reset to `0`.

The `replServer.displayPrompt` method is primarily intended to be called from within the action function for commands registered using the `replServer.defineCommand()` method.

`replServer.clearBufferedCommand()`

The `replServer.clearBufferedCommand()` method clears any command that has been buffered but not yet executed. This method is primarily intended to be called from within the action function for commands registered using the `replServer.defineCommand()` method.

`replServer.parseREPLKeyword(keyword[, rest])`

Stability: 0 - Deprecated.

- `keyword` `<string>` the potential keyword to parse and execute
- `rest` `<any>` any parameters to the keyword command
- Returns: `<boolean>`

An internal method used to parse and execute `REPLServer` keywords. Returns `true` if `keyword` is a valid keyword, otherwise `false`.

`replServer.setupHistory(historyPath, callback)`

- `historyPath` `<string>` the path to the history file
- `callback` `<Function>` called when history writes are ready or upon error
 - `err` `<Error>`
 - `repl` `<repl.REPLServer>`

Initializes a history log file for the REPL instance. When executing the Node.js binary and using the command-line REPL, a history file is initialized by default. However, this is not the case when creating a REPL programmatically. Use this method to initialize a history log file when working with REPL instances programmatically.

`repl.builtinModules`

- `<string[]>`

A list of the names of all Node.js modules, e.g., `'http'`.

`repl.start([options])`

- `options` `<Object> | <string>`
 - `prompt` `<string>` The input prompt to display. **Default:** `'> '` (with a trailing space).
 - `input` `<stream.Readable>` The `Readable` stream from which REPL input will be read. **Default:** `process.stdin`.
 - `output` `<stream.Writable>` The `Writable` stream to which REPL output will be written. **Default:** `process.stdout`.
 - `terminal` `<boolean>` If `true`, specifies that the `output` should be treated as a TTY terminal. **Default:** checking the value of the `isTTY` property on the `output` stream upon instantiation.
 - `eval` `<Function>` The function to be used when evaluating each given line of input. **Default:** an async wrapper for the JavaScript `eval()` function. An `eval` function can error with `repl.Recoverable` to indicate the input was incomplete and prompt for additional lines.
 - `useColors` `<boolean>` If `true`, specifies that the default `writer` function should include ANSI color styling to REPL output. If a custom `writer` function is provided then this has no effect. **Default:** checking color support on the `output` stream if the REPL

instance's `terminal` value is `true`.

- `useGlobal` `<boolean>` If `true`, specifies that the default evaluation function will use the JavaScript `global` as the context as opposed to creating a new separate context for the REPL instance. The node CLI REPL sets this value to `true`. **Default:** `false`.
 - `ignoreUndefined` `<boolean>` If `true`, specifies that the default writer will not output the return value of a command if it evaluates to `undefined`. **Default:** `false`.
 - `writer` `<Function>` The function to invoke to format the output of each command before writing to `output`. **Default:** `util.inspect()`.
 - `completer` `<Function>` An optional function used for custom Tab auto completion. See `readline.InterfaceCompleter` for an example.
 - `replMode` `<symbol>` A flag that specifies whether the default evaluator executes all JavaScript commands in strict mode or default (sloppy) mode. Acceptable values are:
 - `repl.REPL_MODE_SLOPPY` to evaluate expressions in sloppy mode.
 - `repl.REPL_MODE_STRICT` to evaluate expressions in strict mode. This is equivalent to prefacing every repl statement with `'use strict'`.
 - `breakEvalOnSigint` `<boolean>` Stop evaluating the current piece of code when `SIGINT` is received, such as when `Ctrl + C` is pressed. This cannot be used together with a custom `eval` function. **Default:** `false`.
 - `preview` `<boolean>` Defines if the repl prints autocomplete and output previews or not. **Default:** `true` with the default eval function and `false` in case a custom eval function is used. If `terminal` is falsy, then there are no previews and the value of `preview` has no effect.
- Returns: `<repl.REPLServer>`

The `repl.start()` method creates and starts a `repl.REPLServer` instance.

If `options` is a string, then it specifies the input prompt:

```
const repl = require('repl');

// a Unix style prompt
repl.start('$ ');
```

The Node.js REPL

Node.js itself uses the `repl` module to provide its own interactive interface for executing JavaScript. This can be used by executing the Node.js binary without passing any arguments (or by passing the `-i` argument):

```
$ node
> const a = [1, 2, 3];
undefined
> a
[ 1, 2, 3 ]
> a.forEach((v) => {
...   console.log(v);
... });
1
2
3
```

Environment variable options

Various behaviors of the Node.js REPL can be customized using the following environment variables:

- `NODE_REPL_HISTORY` : When a valid path is given, persistent REPL history will be saved to the specified file rather than `.node_repl_history` in the user's home directory. Setting this value to `''` (an empty string) will disable persistent REPL history. Whitespace will be trimmed from the value. On Windows platforms environment variables with empty values are invalid so set this variable to one or more spaces to disable persistent REPL history.
- `NODE_REPL_HISTORY_SIZE` : Controls how many lines of history will be persisted if history is available. Must be a positive number. **Default:** `1000`.
- `NODE_REPL_MODE` : May be either `'sloppy'` or `'strict'`. **Default:** `'sloppy'`, which will allow non-strict mode code to be run.

Persistent history

By default, the Node.js REPL will persist history between `node` REPL sessions by saving inputs to a `.node_repl_history` file located in the user's home directory. This can be disabled by setting the environment variable `NODE_REPL_HISTORY=''`.

Using the Node.js REPL with advanced line-editors

For advanced line-editors, start Node.js with the environment variable `NODE_NO_READLINE=1`. This will start the main and debugger REPL in canonical terminal settings, which will allow use with `rlwrap`.

For example, the following can be added to a `.bashrc` file:

```
alias node="env NODE_NO_READLINE=1 rlwrap node"
```

Starting multiple REPL instances against a single running instance

It is possible to create and run multiple REPL instances against a single running instance of Node.js that share a single `global` object but have separate I/O interfaces.

The following example, for instance, provides separate REPLs on `stdin`, a Unix socket, and a TCP socket:

```
const net = require('net');
const repl = require('repl');
let connections = 0;

repl.start({
  prompt: 'Node.js via stdin> ',
  input: process.stdin,
  output: process.stdout
});

net.createServer((socket) => {
  connections += 1;
  repl.start({
    prompt: 'Node.js via Unix socket> ',
    input: socket,
    output: socket
  }).on('exit', () => {
    socket.end();
  });
}).listen('/tmp/node-repl-sock');
```

```
net.createServer((socket) => {
  connections += 1;
  repl.start({
    prompt: 'Node.js via TCP socket> ',
    input: socket,
    output: socket
  }).on('exit', () => {
    socket.end();
  });
}).listen(5001);
```

Running this application from the command line will start a REPL on stdin. Other REPL clients may connect through the Unix socket or TCP socket. `telnet`, for instance, is useful for connecting to TCP sockets, while `socat` can be used to connect to both Unix and TCP sockets.

By starting a REPL from a Unix socket-based server instead of stdin, it is possible to connect to a long-running Node.js process without restarting it.

For an example of running a "full-featured" (`terminal`) REPL over a `net.Server` and `net.Socket` instance, see: <https://gist.github.com/TooTallNate/2209310>.

For an example of running a REPL instance over `curl(1)`, see: <https://gist.github.com/TooTallNate/2053342>.

Diagnostic report

Stability: 2 - Stable

Delivers a JSON-formatted diagnostic summary, written to a file.

The report is intended for development, test and production use, to capture and preserve information for problem determination. It includes JavaScript and native stack traces, heap statistics, platform information, resource usage etc. With the report option enabled, diagnostic reports can be triggered on unhandled exceptions, fatal errors and user signals, in addition to triggering programmatically through API calls.

A complete example report that was generated on an uncaught exception is provided below for reference.

```
{
  "header": {
    "reportVersion": 1,
    "event": "exception",
    "trigger": "Exception",
    "filename": "report.20181221.005011.8974.0.001.json",
    "dumpEventTime": "2018-12-21T00:50:11Z",
    "dumpEventTimeStamp": "1545371411331",
    "processId": 8974,
    "cwd": "/home/nodeuser/project/node",
    "commandLine": [
      "/home/nodeuser/project/node/out/Release/node",
      "--report-uncaught-exception",
      "/home/nodeuser/project/node/test/report/test-exception.js",
      "child"
    ],
    "nodeVersion": "v6.9.0"
  },
  "exception": {
    "type": "UncaughtException",
    "message": "Uncaught Exception",
    "stack": "Error: Uncaught Exception\n    at Object.<anonymous> (/home/nodeuser/project/node/test/report/test-exception.js:1:1)\n    at Module._compile (module.js:413:26)\n    at Object.Module._extensions..js (module.js:426:10)\n    at Module.load (module.js:357:32)\n    at Function.Module._load (module.js:314:12)\n    at Function.Module.runMain (module.js:247:10)\n    at startup (bootstrap_node.js:151:16)\n    at bootstrap_node.js:586:3"
  },
  "platform": {
    "platform": "linux",
    "arch": "x64",
    "release": "v6.9.0",
    "nodeVersion": "v6.9.0",
    "osVersion": "Ubuntu 16.04.2 LTS",
    "cpu": {
      "vendor": "Intel",
      "brand": "Intel(R) Xeon(R) CPU @ 2.60GHz",
      "model": 60,
      "stepping": 2
    },
    "memory": {
      "total": 16777216,
      "free": 12582912,
      "used": 4194304
    },
    "disk": {
      "total": 107374080,
      "free": 94273280,
      "used": 13100800
    }
  },
  "report": {
    "reportVersion": 1,
    "event": "exception",
    "trigger": "Exception",
    "filename": "report.20181221.005011.8974.0.001.json",
    "dumpEventTime": "2018-12-21T00:50:11Z",
    "dumpEventTimeStamp": "1545371411331",
    "processId": 8974,
    "cwd": "/home/nodeuser/project/node",
    "commandLine": [
      "/home/nodeuser/project/node/out/Release/node",
      "--report-uncaught-exception",
      "/home/nodeuser/project/node/test/report/test-exception.js",
      "child"
    ],
    "nodeVersion": "v6.9.0"
  }
}
```

```
"nodejsVersion": "v12.0.0-pre",
"glibcVersionRuntime": "2.17",
"glibcVersionCompiler": "2.17",
"wordSize": "64 bit",
"arch": "x64",
"platform": "linux",
"componentVersions": {
  "node": "12.0.0-pre",
  "v8": "7.1.302.28-node.5",
  "uv": "1.24.1",
  "zlib": "1.2.11",
  "ares": "1.15.0",
  "modules": "68",
  "nghttp2": "1.34.0",
  "napi": "3",
  "llhttp": "1.0.1",
  "openssl": "1.1.0j"
},
"release": {
  "name": "node"
},
"osName": "Linux",
"osRelease": "3.10.0-862.el7.x86_64",
"osVersion": "#1 SMP Wed Mar 21 18:14:51 EDT 2018",
"osMachine": "x86_64",
"cpus": [
  {
    "model": "Intel(R) Core(TM) i7-6820HQ CPU @ 2.70GHz",
    "speed": 2700,
    "user": 88902660,
    "nice": 0,
    "sys": 50902570,
    "idle": 241732220,
    "irq": 0
  },
  {
    "model": "Intel(R) Core(TM) i7-6820HQ CPU @ 2.70GHz",
    "speed": 2700,
    "user": 88902660,
    "nice": 0,
    "sys": 50902570,
    "idle": 241732220,
    "irq": 0
  }
],
"networkInterfaces": [
  {
    "name": "en0",
    "internal": false,
    "mac": "13:10:de:ad:be:ef",
    "address": "10.0.0.37",
    "netmask": "255.255.255.0"
```

```

    "netmask": "255.255.255.0",
    "family": "IPv4"
  },
],
"host": "test_machine"
},
"javascriptStack": {
  "message": "Error: *** test-exception.js: throwing uncaught Error",
  "stack": [
    "at myException (/home/nodeuser/project/node/test/report/test-exception.js:9:11)",
    "at Object.<anonymous> (/home/nodeuser/project/node/test/report/test-exception.js:12:3)",
    "at Module._compile (internal/modules/cjs/loader.js:718:30)",
    "at Object.Module._extensions..js (internal/modules/cjs/loader.js:729:10)",
    "at Module.load (internal/modules/cjs/loader.js:617:32)",
    "at tryModuleLoad (internal/modules/cjs/loader.js:560:12)",
    "at Function.Module._load (internal/modules/cjs/loader.js:552:3)",
    "at Function.Module.runMain (internal/modules/cjs/loader.js:771:12)",
    "at executeUserCode (internal/bootstrap/node.js:332:15)"
  ]
},
"nativeStack": [
  {
    "pc": "0x000055b57f07a9ef",
    "symbol": "report::GetNodeReport(v8::Isolate*, node::Environment*, char const*, char const*, v8::Local<v8::String>,"
  },
  {
    "pc": "0x000055b57f07cf03",
    "symbol": "report::GetReport(v8::FunctionCallbackInfo<v8::Value> const&) [./node]"
  },
  {
    "pc": "0x000055b57f1bccfd",
    "symbol": " [./node]"
  },
  {
    "pc": "0x000055b57f1be048",
    "symbol": "v8::internal::Builtin_HandleApiCall(int, v8::internal::Object**, v8::internal::Isolate*) [./node]"
  },
  {
    "pc": "0x000055b57feeda0e",
    "symbol": " [./node]"
  }
],
"javascriptHeap": {
  "totalMemory": 6127616,
  "totalCommittedMemory": 4357352,
  "usedMemory": 3221136,
  "availableMemory": 1521370240,
  "memoryLimit": 1526909922,
  "heapSpaces": {
    "read_only_space": {
      "memorySize": 524288,
      "committedMemory": 00000
    }
  }
}

```

```
    "committedMemory": 39208,  
    "capacity": 515584,  
    "used": 30504,  
    "available": 485080  
  },  
  "new_space": {  
    "memorySize": 2097152,  
    "committedMemory": 2019312,  
    "capacity": 1031168,  
    "used": 985496,  
    "available": 45672  
  },  
  "old_space": {  
    "memorySize": 2273280,  
    "committedMemory": 1769008,  
    "capacity": 1974640,  
    "used": 1725488,  
    "available": 249152  
  },  
  "code_space": {  
    "memorySize": 696320,  
    "committedMemory": 184896,  
    "capacity": 152128,  
    "used": 152128,  
    "available": 0  
  },  
  "map_space": {  
    "memorySize": 536576,  
    "committedMemory": 344928,  
    "capacity": 327520,  
    "used": 327520,  
    "available": 0  
  },  
  "large_object_space": {  
    "memorySize": 0,  
    "committedMemory": 0,  
    "capacity": 1520590336,  
    "used": 0,  
    "available": 1520590336  
  },  
  "new_large_object_space": {  
    "memorySize": 0,  
    "committedMemory": 0,  
    "capacity": 0,  
    "used": 0,  
    "available": 0  
  }  
}  
},  
"resourceUsage": {  
  "userCpuSeconds": 0.069595,  
  "kernelCpuSeconds": 0.040100
```

```

    "kernelCpuSeconds": 0.019163,
    "cpuConsumptionPercent": 0.000000,
    "maxRss": 18079744,
    "pageFaults": {
      "IORequired": 0,
      "IONotRequired": 4610
    },
    "fsActivity": {
      "reads": 0,
      "writes": 0
    }
  },
  "uvthreadResourceUsage": {
    "userCpuSeconds": 0.068457,
    "kernelCpuSeconds": 0.019127,
    "cpuConsumptionPercent": 0.000000,
    "fsActivity": {
      "reads": 0,
      "writes": 0
    }
  },
  "libuv": [
    {
      "type": "async",
      "is_active": true,
      "is_referenced": false,
      "address": "0x0000000102910900",
      "details": ""
    },
    {
      "type": "timer",
      "is_active": false,
      "is_referenced": false,
      "address": "0x00007fff5fbfeab0",
      "repeat": 0,
      "firesInMsFromNow": 94403548320796,
      "expired": true
    },
    {
      "type": "check",
      "is_active": true,
      "is_referenced": false,
      "address": "0x00007fff5fbfeb48"
    },
    {
      "type": "idle",
      "is_active": false,
      "is_referenced": true,
      "address": "0x00007fff5fbfeb00"
    },
    {
      "type": "process"
    }
  ]
}

```

```

    "type": "prepare",
    "is_active": false,
    "is_referenced": false,
    "address": "0x00007fff5fbfec38"
},
{
    "type": "check",
    "is_active": false,
    "is_referenced": false,
    "address": "0x00007fff5fbfecb0"
},
{
    "type": "async",
    "is_active": true,
    "is_referenced": false,
    "address": "0x000000010188f2e0"
},
{
    "type": "tty",
    "is_active": false,
    "is_referenced": true,
    "address": "0x000055b581db0e18",
    "width": 204,
    "height": 55,
    "fd": 17,
    "writeQueueSize": 0,
    "readable": true,
    "writable": true
},
{
    "type": "signal",
    "is_active": true,
    "is_referenced": false,
    "address": "0x000055b581d80010",
    "signum": 28,
    "signal": "SIGWINCH"
},
{
    "type": "tty",
    "is_active": true,
    "is_referenced": true,
    "address": "0x000055b581df59f8",
    "width": 204,
    "height": 55,
    "fd": 19,
    "writeQueueSize": 0,
    "readable": true,
    "writable": true
},
{
    "type": "loop",
    "is_active": true

```

```
    "is_active": true,
    "address": "0x000055fc7b2cb180",
    "loopIdleTimeSeconds": 22644.8
  }
],
"workers": [],
"environmentVariables": {
  "REMOTEHOST": "REMOVED",
  "MANPATH": "/opt/rh/devtoolset-3/root/usr/share/man:",
  "XDG_SESSION_ID": "66126",
  "HOSTNAME": "test_machine",
  "HOST": "test_machine",
  "TERM": "xterm-256color",
  "SHELL": "/bin/csh",
  "SSH_CLIENT": "REMOVED",
  "PERL5LIB": "/opt/rh/devtoolset-3/root//usr/lib64/perl5/vendor_perl:/opt/rh/devtoolset-3/root/usr/lib/perl5:/opt/rh/d",
  "OLDPWD": "/home/nodeuser/project/node/src",
  "JAVACONFDIRS": "/opt/rh/devtoolset-3/root/etc/java:/etc/java",
  "SSH_TTY": "/dev/pts/0",
  "PCP_DIR": "/opt/rh/devtoolset-3/root",
  "GROUP": "normaluser",
  "USER": "nodeuser",
  "LD_LIBRARY_PATH": "/opt/rh/devtoolset-3/root/usr/lib64:/opt/rh/devtoolset-3/root/usr/lib",
  "HOSTTYPE": "x86_64-linux",
  "XDG_CONFIG_DIRS": "/opt/rh/devtoolset-3/root/etc/xdg:/etc/xdg",
  "MAIL": "/var/spool/mail/nodeuser",
  "PATH": "/home/nodeuser/project/node:/opt/rh/devtoolset-3/root/usr/bin:/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/s",
  "PWD": "/home/nodeuser/project/node",
  "LANG": "en_US.UTF-8",
  "PS1": "\\u@\\h : \\[\\e[31m\\]\\w\\[\\e[m\\] > ",
  "SHLVL": "2",
  "HOME": "/home/nodeuser",
  "OSTYPE": "linux",
  "VENDOR": "unknown",
  "PYTHONPATH": "/opt/rh/devtoolset-3/root/usr/lib64/python2.7/site-packages:/opt/rh/devtoolset-3/root/usr/lib/python2.",
  "MACHTYPE": "x86_64",
  "LOGNAME": "nodeuser",
  "XDG_DATA_DIRS": "/opt/rh/devtoolset-3/root/usr/share:/usr/local/share:/usr/share",

  "LESSOPEN": "|/usr/bin/lesspipe.sh %s",
  "INFOPATH": "/opt/rh/devtoolset-3/root/usr/share/info",
  "XDG_RUNTIME_DIR": "/run/user/50141",
  "_": "./node"
},
"userLimits": {
  "core_file_size_blocks": {
    "soft": "",
    "hard": "unlimited"
  },
  "data_seg_size_kbytes": {
    "soft": "unlimited",
    "hard": "unlimited"
  },

```



```

},
"file_size_blocks": {
  "soft": "unlimited",
  "hard": "unlimited"
},
"max_locked_memory_bytes": {
  "soft": "unlimited",
  "hard": 65536
},
"max_memory_size_kbytes": {
  "soft": "unlimited",
  "hard": "unlimited"
},
"open_files": {
  "soft": "unlimited",
  "hard": 4096
},
"stack_size_bytes": {
  "soft": "unlimited",
  "hard": "unlimited"
},
"cpu_time_seconds": {
  "soft": "unlimited",
  "hard": "unlimited"
},
"max_user_processes": {
  "soft": "unlimited",
  "hard": 4127290
},
"virtual_memory_kbytes": {
  "soft": "unlimited",
  "hard": "unlimited"
}
},
"sharedObjects": [
  "/lib64/libdl.so.2",
  "/lib64/librt.so.1",
  "/lib64/libstdc++.so.6",

  "/lib64/libm.so.6",
  "/lib64/libgcc_s.so.1",
  "/lib64/libpthread.so.0",
  "/lib64/libc.so.6",
  "/lib64/ld-linux-x86-64.so.2"
]
}

```

Usage

```
node --report-uncaught-exception --report-on-signal \  
--report-on-fatalerror app.js
```

- `--report-uncaught-exception` Enables report to be generated on un-caught exceptions. Useful when inspecting JavaScript stack in conjunction with native stack and other runtime environment data.
- `--report-on-signal` Enables report to be generated upon receiving the specified (or predefined) signal to the running Node.js process. (See below on how to modify the signal that triggers the report.) Default signal is `SIGUSR2`. Useful when a report needs to be triggered from another program. Application monitors may leverage this feature to collect report at regular intervals and plot rich set of internal runtime data to their views.

Signal based report generation is not supported in Windows.

Under normal circumstances, there is no need to modify the report triggering signal. However, if `SIGUSR2` is already used for other purposes, then this flag helps to change the signal for report generation and preserve the original meaning of `SIGUSR2` for the said purposes.

- `--report-on-fatalerror` Enables the report to be triggered on fatal errors (internal errors within the Node.js runtime, such as out of memory) that leads to termination of the application. Useful to inspect various diagnostic data elements such as heap, stack, event loop state, resource consumption etc. to reason about the fatal error.
- `--report-compact` Write reports in a compact format, single-line JSON, more easily consumable by log processing systems than the default multi-line format designed for human consumption.
- `--report-directory` Location at which the report will be generated.
- `--report-filename` Name of the file to which the report will be written.
- `--report-signal` Sets or resets the signal for report generation (not supported on Windows). Default signal is `SIGUSR2`.

A report can also be triggered via an API call from a JavaScript application:

```
process.report.writeReport();
```

This function takes an optional additional argument `filename`, which is the name of a file into which the report is written.

```
process.report.writeReport('./foo.json');
```

This function takes an optional additional argument `err` which is an `Error` object that will be used as the context for the JavaScript stack printed in the report. When using report to handle errors in a callback or an exception handler, this allows the report to include the location of the original error as well as where it was handled.

```
try {  
  process.chdir('/non-existent-path');  
} catch (err) {  
  process.report.writeReport(err);  
}  
  
// Any other code
```

If both filename and error object are passed to `writeReport()` the error object must be the second parameter.

```
try {
  process.chdir('/non-existent-path');
} catch (err) {
  process.report.writeReport(filename, err);
}
// Any other code
```

The content of the diagnostic report can be returned as a JavaScript Object via an API call from a JavaScript application:

```
const report = process.report.getReport();
console.log(typeof report === 'object'); // true

// Similar to process.report.writeReport() output
console.log(JSON.stringify(report, null, 2));
```

This function takes an optional additional argument `err`, which is an `Error` object that will be used as the context for the JavaScript stack printed in the report.

```
const report = process.report.getReport(new Error('custom error'));
console.log(typeof report === 'object'); // true
```

The API versions are useful when inspecting the runtime state from within the application, in expectation of self-adjusting the resource consumption, load balancing, monitoring etc.

The content of the report consists of a header section containing the event type, date, time, PID and Node.js version, sections containing JavaScript and native stack traces, a section containing V8 heap information, a section containing `libuv` handle information and an OS platform information section showing CPU and memory usage and system limits. An example report can be triggered using the Node.js REPL:

```
$ node
> process.report.writeReport();
Writing Node.js report to file: report.20181126.091102.8480.0.001.json
Node.js report completed
>
```

When a report is written, start and end messages are issued to `stderr` and the filename of the report is returned to the caller. The default filename includes the date, time, PID and a sequence number. The sequence number helps in associating the report dump with the runtime state if generated multiple times for the same Node.js process.

Configuration

Additional runtime configuration of report generation is available via the following properties of `process.report`:

`reportOnFatalError` triggers diagnostic reporting on fatal errors when `true`. Defaults to `false`.

`reportOnSignal` triggers diagnostic reporting on signal when `true`. This is not supported on Windows. Defaults to `false`.

`reportOnUncaughtException` triggers diagnostic reporting on uncaught exception when `true`. Defaults to `false`.

`signal` specifies the POSIX signal identifier that will be used to intercept external triggers for report generation. Defaults to `'SIGUSR2'`.

`filename` specifies the name of the output file in the file system. Special meaning is attached to `stdout` and `stderr`. Usage of these will result in report being written to the associated standard streams. In cases where standard streams are used, the value in `directory` is ignored. URLs are not supported. Defaults to a composite filename that contains timestamp, PID and sequence number.

`directory` specifies the filesystem directory where the report will be written. URLs are not supported. Defaults to the current working directory of the Node.js process.

```
// Trigger report only on uncaught exceptions.
process.report.reportOnFatalError = false;
process.report.reportOnSignal = false;
process.report.reportOnUncaughtException = true;

// Trigger report for both internal errors as well as external signal.
process.report.reportOnFatalError = true;
process.report.reportOnSignal = true;
process.report.reportOnUncaughtException = false;

// Change the default signal to 'SIGQUIT' and enable it.
process.report.reportOnFatalError = false;
process.report.reportOnUncaughtException = false;
process.report.reportOnSignal = true;
process.report.signal = 'SIGQUIT';
```

Configuration on module initialization is also available via environment variables:

```
NODE_OPTIONS="--report-uncaught-exception \
--report-on-fatalerror --report-on-signal \
--report-signal=SIGUSR2 --report-filename=./report.json \
--report-directory=/home/nodeuser"
```

Specific API documentation can be found under [process API documentation](#) section.

Interaction with workers

`Worker` threads can create reports in the same way that the main thread does.

Reports will include information on any Workers that are children of the current thread as part of the `workers` section, with each Worker generating a report in the standard report format.

The thread which is generating the report will wait for the reports from Worker threads to finish. However, the latency for this will usually be low, as both running JavaScript and the event loop are interrupted to generate the report.

Stream

Stability: 2 - Stable

Source Code: [lib/stream.js](#)

A stream is an abstract interface for working with streaming data in Node.js. The `stream` module provides an API for implementing the stream interface.

There are many stream objects provided by Node.js. For instance, a `request to an HTTP server` and `process.stdout` are both stream instances.

Streams can be readable, writable, or both. All streams are instances of `EventEmitter`.

To access the `stream` module:

```
const stream = require('stream');
```

The `stream` module is useful for creating new types of stream instances. It is usually not necessary to use the `stream` module to consume streams.

Organization of this document

This document contains two primary sections and a third section for notes. The first section explains how to use existing streams within an application. The second section explains how to create new types of streams.

Types of streams

There are four fundamental stream types within Node.js:

- `Writable`: streams to which data can be written (for example, `fs.createWriteStream()`).
- `Readable`: streams from which data can be read (for example, `fs.createReadStream()`).
- `Duplex`: streams that are both `Readable` and `Writable` (for example, `net.Socket`).
- `Transform`: `Duplex` streams that can modify or transform the data as it is written and read (for example, `zlib.createDeflate()`).

Additionally, this module includes the utility functions `stream.pipeline()`, `stream.finished()`, `stream.Readable.from()` and `stream.addAbortSignal()`.

Streams Promises API

The `stream/promises` API provides an alternative set of asynchronous utility functions for streams that return `Promise` objects rather than using callbacks. The API is accessible via `require('stream/promises')` or `require('stream').promises`.

Object mode

All streams created by Node.js APIs operate exclusively on strings and `Buffer` (or `Uint8Array`) objects. It is possible, however, for stream implementations to work with other types of JavaScript values (with the exception of `null`, which serves a special purpose within streams). Such streams are considered to operate in "object mode".

Stream instances are switched into object mode using the `objectMode` option when the stream is created. Attempting to switch an existing stream into object mode is not safe.

Buffering

Both `Writable` and `Readable` streams will store data in an internal buffer.

The amount of data potentially buffered depends on the `highWaterMark` option passed into the stream's constructor. For normal streams, the `highWaterMark` option specifies a `total number of bytes`. For streams operating in object mode, the `highWaterMark` specifies a total number of objects.

Data is buffered in `Readable` streams when the implementation calls `stream.push(chunk)`. If the consumer of the Stream does not call `stream.read()`, the data will sit in the internal queue until it is consumed.

Once the total size of the internal read buffer reaches the threshold specified by `highWaterMark`, the stream will temporarily stop reading data from the underlying resource until the data currently buffered can be consumed (that is, the stream will stop calling the internal `readable._read()` method that is used to fill the read buffer).

Data is buffered in `Writable` streams when the `writable.write(chunk)` method is called repeatedly. While the total size of the internal write buffer is below the threshold set by `highWaterMark`, calls to `writable.write()` will return `true`. Once the size of the internal buffer reaches or exceeds the `highWaterMark`, `false` will be returned.

A key goal of the `stream` API, particularly the `stream.pipe()` method, is to limit the buffering of data to acceptable levels such that sources and destinations of differing speeds will not overwhelm the available memory.

The `highWaterMark` option is a threshold, not a limit: it dictates the amount of data that a stream buffers before it stops asking for more data. It does not enforce a strict memory limitation in general. Specific stream implementations may choose to enforce stricter limits but doing so is optional.

Because `Duplex` and `Transform` streams are both `Readable` and `Writable`, each maintains two separate internal buffers used for reading and writing, allowing each side to operate independently of the other while maintaining an appropriate and efficient flow of data. For example, `net.Socket` instances are `Duplex` streams whose `Readable` side allows consumption of data received from the socket and whose `Writable` side allows writing data to the socket. Because data may be written to the socket at a faster or slower rate than data is received, each side should operate (and buffer) independently of the other.

The mechanics of the internal buffering are an internal implementation detail and may be changed at any time. However, for certain advanced implementations, the internal buffers can be retrieved using `writable.writableBuffer` or `readable.readableBuffer`. Use of these undocumented properties is discouraged.

API for stream consumers

Almost all Node.js applications, no matter how simple, use streams in some manner. The following is an example of using streams in a Node.js application that implements an HTTP server:

```
const http = require('http');

const server = http.createServer((req, res) => {
  // `req` is an http.IncomingMessage, which is a readable stream.
  // `res` is an http.ServerResponse, which is a writable stream.

  let body = '';
  // Get the data as utf8 strings.
  // If an encoding is not set, Buffer objects will be received.
  req.setEncoding('utf8');

  // Readable streams emit 'data' events once a listener is added.
  req.on('data', (chunk) => {
    body += chunk;
  });

  // The 'end' event indicates that the entire body has been received.
  req.on('end', () => {
    try {
      const data = JSON.parse(body);
```

```

    // Write back something interesting to the user:
    res.write(typeof data);
    res.end();
  } catch (er) {
    // uh oh! bad json!
    res.statusCode = 400;
    return res.end(`error: ${er.message}`);
  }
});

server.listen(1337);

// $ curl localhost:1337 -d "{}"
// object
// $ curl localhost:1337 -d "\"foo\""
// string
// $ curl localhost:1337 -d "not json"
// error: Unexpected token o in JSON at position 1

```

Writable streams (such as `res` in the example) expose methods such as `write()` and `end()` that are used to write data onto the stream.

Readable streams use the **EventEmitter** API for notifying application code when data is available to be read off the stream. That available data can be read from the stream in multiple ways.

Both **Writable** and **Readable** streams use the **EventEmitter** API in various ways to communicate the current state of the stream.

Duplex and **Transform** streams are both **Writable** and **Readable**.

Applications that are either writing data to or consuming data from a stream are not required to implement the stream interfaces directly and will generally have no reason to call `require('stream')`.

Developers wishing to implement new types of streams should refer to the section [API for stream implementers](#).

Writable streams

Writable streams are an abstraction for a *destination* to which data is written.

Examples of **Writable** streams include:

- HTTP requests, on the client
- HTTP responses, on the server
- fs write streams
- zlib streams
- crypto streams
- TCP sockets
- child process stdin
- `process.stdout`, `process.stderr`

Some of these examples are actually **Duplex** streams that implement the **Writable** interface.

All **Writable** streams implement the interface defined by the `stream.Writable` class.

While specific instances of `Writable` streams may differ in various ways, all `Writable` streams follow the same fundamental usage pattern as illustrated in the example below:

```
const myStream = getWritableStreamSomehow();
myStream.write('some data');
myStream.write('some more data');
myStream.end('done writing data');
```

Class: `stream.Writable`

Event: 'close'

The 'close' event is emitted when the stream and any of its underlying resources (a file descriptor, for example) have been closed. The event indicates that no more events will be emitted, and no further computation will occur.

A `Writable` stream will always emit the 'close' event if it is created with the `emitClose` option.

Event: 'drain'

If a call to `stream.write(chunk)` returns `false`, the 'drain' event will be emitted when it is appropriate to resume writing data to the stream.

```
// Write the data to the supplied writable stream one million times.
// Be attentive to back-pressure.
function writeOneMillionTimes(writer, data, encoding, callback) {
  let i = 1000000;
  write();
  function write() {
    let ok = true;
    do {
      i--;
      if (i === 0) {
        // Last time!
        writer.write(data, encoding, callback);
      } else {
        // See if we should continue, or wait.
        // Don't pass the callback, because we're not done yet.
        ok = writer.write(data, encoding);
      }
    } while (i > 0 && ok);
    if (i > 0) {
      // Had to stop early!
      // Write some more once it drains.
      writer.once('drain', write);
    }
  }
}
```

Event: 'error'

- `<Error>`

The `'error'` event is emitted if an error occurred while writing or piping data. The listener callback is passed a single `Error` argument when called.

The stream is closed when the `'error'` event is emitted unless the `autoDestroy` option was set to `false` when creating the stream.

After `'error'`, no further events other than `'close'` *should* be emitted (including `'error'` events).

Event: `'finish'`

The `'finish'` event is emitted after the `stream.end()` method has been called, and all data has been flushed to the underlying system.

```
const writer = getWritableStreamSomehow();
for (let i = 0; i < 100; i++) {
  writer.write(`hello, #${i}!\n`);
}
writer.on('finish', () => {
  console.log('All writes are now complete.');
```

```
});
writer.end('This is the end\n');
```

Event: `'pipe'`

- `src` `<stream.Readable>` source stream that is piping to this writable

The `'pipe'` event is emitted when the `stream.pipe()` method is called on a readable stream, adding this writable to its set of destinations.

```
const writer = getWritableStreamSomehow();
const reader = getReadableStreamSomehow();
writer.on('pipe', (src) => {
  console.log('Something is piping into the writer.');
```

```
  assert.equal(src, reader);
});
reader.pipe(writer);
```

Event: `'unpipe'`

- `src` `<stream.Readable>` The source stream that `unpiped` this writable

The `'unpipe'` event is emitted when the `stream.unpipe()` method is called on a `Readable` stream, removing this `Writable` from its set of destinations.

This is also emitted in case this `Writable` stream emits an error when a `Readable` stream pipes into it.

```
const writer = getWritableStreamSomehow();
const reader = getReadableStreamSomehow();
writer.on('unpipe', (src) => {
  console.log('Something has stopped piping into the writer.');
```

```
  assert.equal(src, reader);
});
reader.pipe(writer);
reader.unpipe(writer);
```

writable.cork()

The `writable.cork()` method forces all written data to be buffered in memory. The buffered data will be flushed when either the `stream.uncork()` or `stream.end()` methods are called.

The primary intent of `writable.cork()` is to accommodate a situation in which several small chunks are written to the stream in rapid succession. Instead of immediately forwarding them to the underlying destination, `writable.cork()` buffers all the chunks until `writable.uncork()` is called, which will pass them all to `writable._writev()`, if present. This prevents a head-of-line blocking situation where data is being buffered while waiting for the first small chunk to be processed. However, use of `writable.cork()` without implementing `writable._writev()` may have an adverse effect on throughput.

See also: `writable.uncork()`, `writable._writev()`.

writable.destroy([error])

- `error` `<Error>` Optional, an error to emit with `'error'` event.
- Returns: `<this>`

Destroy the stream. Optionally emit an `'error'` event, and emit a `'close'` event (unless `emitClose` is set to `false`). After this call, the writable stream has ended and subsequent calls to `write()` or `end()` will result in an `ERR_STREAM_DESTROYED` error. This is a destructive and immediate way to destroy a stream. Previous calls to `write()` may not have drained, and may trigger an `ERR_STREAM_DESTROYED` error. Use `end()` instead of `destroy` if data should flush before close, or wait for the `'drain'` event before destroying the stream.

```
const { Writable } = require('stream');

const myStream = new Writable();

const fooErr = new Error('foo error');
myStream.destroy(fooErr);
myStream.on('error', (fooErr) => console.error(fooErr.message)); // foo error
```

```
const { Writable } = require('stream');

const myStream = new Writable();

myStream.destroy();
myStream.on('error', function wontHappen() {});
```

```
const { Writable } = require('stream');

const myStream = new Writable();
myStream.destroy();

myStream.write('foo', (error) => console.error(error.code));
// ERR_STREAM_DESTROYED
```

Once `destroy()` has been called any further calls will be a no-op and no further errors except from `_destroy()` may be emitted as `'error'`.

Implementors should not override this method, but instead implement `writable._destroy()`.

writable.destroyed

- `<boolean>`

Is `true` after `writable.destroy()` has been called.

```
const { Writable } = require('stream');

const myStream = new Writable();

console.log(myStream.destroyed); // false
myStream.destroy();
console.log(myStream.destroyed); // true
```

writable.end([chunk[, encoding]][, callback])

- `chunk` `<string>` | `<Buffer>` | `<Uint8Array>` | `<any>` Optional data to write. For streams not operating in object mode, `chunk` must be a string, `Buffer` or `Uint8Array`. For object mode streams, `chunk` may be any JavaScript value other than `null`.
- `encoding` `<string>` The encoding if `chunk` is a string
- `callback` `<Function>` Callback for when the stream is finished.
- Returns: `<this>`

Calling the `writable.end()` method signals that no more data will be written to the `Writable`. The optional `chunk` and `encoding` arguments allow one final additional chunk of data to be written immediately before closing the stream.

Calling the `stream.write()` method after calling `stream.end()` will raise an error.

```
// Write 'hello, ' and then end with 'world!'.
const fs = require('fs');
const file = fs.createWriteStream('example.txt');
file.write('hello, ');
file.end('world!');
// Writing more now is not allowed!
```

writable.setDefaultEncoding(encoding)

- `encoding` `<string>` The new default encoding
- Returns: `<this>`

The `writable.setDefaultEncoding()` method sets the default `encoding` for a `Writable` stream.

writable.uncork()

The `writable.uncork()` method flushes all data buffered since `stream.cork()` was called.

When using `writable.cork()` and `writable.uncork()` to manage the buffering of writes to a stream, it is recommended that calls to `writable.uncork()` be deferred using `process.nextTick()`. Doing so allows batching of all `writable.write()` calls that occur within a given Node.js event loop phase.

```
stream.cork();
stream.write('some ');
```

```
stream.write('data ');
process.nextTick(() => stream.uncork());
```

If the `writable.cork()` method is called multiple times on a stream, the same number of calls to `writable.uncork()` must be called to flush the buffered data.

```
stream.cork();
stream.write('some ');
stream.cork();
stream.write('data ');
process.nextTick(() => {
  stream.uncork();
  // The data will not be flushed until uncork() is called a second time.
  stream.uncork();
});
```

See also: `writable.cork()`.

`writable.writable`

- `<boolean>`

Is `true` if it is safe to call `writable.write()`, which means the stream has not been destroyed, errored or ended.

`writable.writableEnded`

- `<boolean>`

Is `true` after `writable.end()` has been called. This property does not indicate whether the data has been flushed, for this use `writable.writableFinished` instead.

`writable.writableCorked`

- `<integer>`

Number of times `writable.uncork()` needs to be called in order to fully uncork the stream.

`writable.writableFinished`

- `<boolean>`

Is set to `true` immediately before the `'finish'` event is emitted.

`writable.writableHighWaterMark`

- `<number>`

Return the value of `highWaterMark` passed when creating this `Writable`.

`writable.writableLength`

- `<number>`

This property contains the number of bytes (or objects) in the queue ready to be written. The value provides introspection data regarding the status of the `highWaterMark`.

`writable.writableNeedDrain`

- `<boolean>`

Is `true` if the stream's buffer has been full and stream will emit `'drain'`.

`writable.writableObjectMode`

- `<boolean>`

Getter for the property `objectMode` of a given `Writable` stream.

`writable.write(chunk[, encoding][, callback])`

- `chunk` `<string>` | `<Buffer>` | `<Uint8Array>` | `<any>` Optional data to write. For streams not operating in object mode, `chunk` must be a string, `Buffer` or `Uint8Array`. For object mode streams, `chunk` may be any JavaScript value other than `null`.
- `encoding` `<string>` | `<null>` The encoding, if `chunk` is a string. **Default:** `'utf8'`
- `callback` `<Function>` Callback for when this chunk of data is flushed.
- Returns: `<boolean>` `false` if the stream wishes for the calling code to wait for the `'drain'` event to be emitted before continuing to write additional data; otherwise `true`.

The `writable.write()` method writes some data to the stream, and calls the supplied `callback` once the data has been fully handled. If an error occurs, the `callback` will be called with the error as its first argument. The `callback` is called asynchronously and before `'error'` is emitted.

The return value is `true` if the internal buffer is less than the `highWaterMark` configured when the stream was created after admitting `chunk`. If `false` is returned, further attempts to write data to the stream should stop until the `'drain'` event is emitted.

While a stream is not draining, calls to `write()` will buffer `chunk`, and return false. Once all currently buffered chunks are drained (accepted for delivery by the operating system), the `'drain'` event will be emitted. It is recommended that once `write()` returns false, no more chunks be written until the `'drain'` event is emitted. While calling `write()` on a stream that is not draining is allowed, Node.js will buffer all written chunks until maximum memory usage occurs, at which point it will abort unconditionally. Even before it aborts, high memory usage will cause poor garbage collector performance and high RSS (which is not typically released back to the system, even after the memory is no longer required). Since TCP sockets may never drain if the remote peer does not read the data, writing a socket that is not draining may lead to a remotely exploitable vulnerability.

Writing data while the stream is not draining is particularly problematic for a `Transform`, because the `Transform` streams are paused by default until they are piped or a `'data'` or `'readable'` event handler is added.

If the data to be written can be generated or fetched on demand, it is recommended to encapsulate the logic into a `Readable` and use `stream.pipe()`. However, if calling `write()` is preferred, it is possible to respect backpressure and avoid memory issues using the `'drain'` event:

```
function write(data, cb) {
  if (!stream.write(data)) {
    stream.once('drain', cb);
  } else {
    process.nextTick(cb);
  }
}

// Wait for cb to be called before doing any other write.
write('hello', () => {
  console.log('Write completed, do more writes now.');
```

A `Writable` stream in object mode will always ignore the `encoding` argument.

Readable streams

Readable streams are an abstraction for a *source* from which data is consumed.

Examples of `Readable` streams include:

- HTTP responses, on the client
- HTTP requests, on the server
- `fs` read streams
- `zlib` streams
- `crypto` streams
- TCP sockets
- `child process stdout and stderr`
- `process.stdin`

All `Readable` streams implement the interface defined by the `stream.Readable` class.

Two reading modes

`Readable` streams effectively operate in one of two modes: flowing and paused. These modes are separate from `object mode`. A `Readable` stream can be in object mode or not, regardless of whether it is in flowing mode or paused mode.

- In flowing mode, data is read from the underlying system automatically and provided to an application as quickly as possible using events via the `EventEmitter` interface.
- In paused mode, the `stream.read()` method must be called explicitly to read chunks of data from the stream.

All `Readable` streams begin in paused mode but can be switched to flowing mode in one of the following ways:

- Adding a `'data'` event handler.
- Calling the `stream.resume()` method.
- Calling the `stream.pipe()` method to send the data to a `Writable`.

The `Readable` can switch back to paused mode using one of the following:

- If there are no pipe destinations, by calling the `stream.pause()` method.
- If there are pipe destinations, by removing all pipe destinations. Multiple pipe destinations may be removed by calling the `stream.unpipe()` method.

The important concept to remember is that a `Readable` will not generate data until a mechanism for either consuming or ignoring that data is provided. If the consuming mechanism is disabled or taken away, the `Readable` will *attempt* to stop generating the data.

For backward compatibility reasons, removing `'data'` event handlers will **not** automatically pause the stream. Also, if there are piped destinations, then calling `stream.pause()` will not guarantee that the stream will *remain* paused once those destinations drain and ask for more data.

If a `Readable` is switched into flowing mode and there are no consumers available to handle the data, that data will be lost. This can occur, for instance, when the `readable.resume()` method is called without a listener attached to the `'data'` event, or when a `'data'` event handler is removed from the stream.

Adding a `'readable'` event handler automatically makes the stream stop flowing, and the data has to be consumed via `readable.read()`. If the `'readable'` event handler is removed, then the stream will start flowing again if there is a `'data'` event handler.

Three states

The "two modes" of operation for a `Readable` stream are a simplified abstraction for the more complicated internal state management that is happening within the `Readable` stream implementation.

Specifically, at any given point in time, every `Readable` is in one of three possible states:

- `readable.readableFlowing === null`
- `readable.readableFlowing === false`
- `readable.readableFlowing === true`

When `readable.readableFlowing` is `null`, no mechanism for consuming the stream's data is provided. Therefore, the stream will not generate data. While in this state, attaching a listener for the `'data'` event, calling the `readable.pipe()` method, or calling the `readable.resume()` method will switch `readable.readableFlowing` to `true`, causing the `Readable` to begin actively emitting events as data is generated.

Calling `readable.pause()`, `readable.unpipe()`, or receiving backpressure will cause the `readable.readableFlowing` to be set as `false`, temporarily halting the flowing of events but *not* halting the generation of data. While in this state, attaching a listener for the `'data'` event will not switch `readable.readableFlowing` to `true`.

```
const { PassThrough, Writable } = require('stream');
const pass = new PassThrough();
const writable = new Writable();

pass.pipe(writable);
pass.unpipe(writable);
// readableFlowing is now false.

pass.on('data', (chunk) => { console.log(chunk.toString()); });
pass.write('ok'); // Will not emit 'data'.
pass.resume();    // Must be called to make stream emit 'data'.
```

While `readable.readableFlowing` is `false`, data may be accumulating within the stream's internal buffer.

Choose one API style

The `Readable` stream API evolved across multiple Node.js versions and provides multiple methods of consuming stream data. In general, developers should choose *one* of the methods of consuming data and *should never* use multiple methods to consume data from a single stream. Specifically, using a combination of `on('data')`, `on('readable')`, `pipe()`, or async iterators could lead to unintuitive behavior.

Use of the `readable.pipe()` method is recommended for most users as it has been implemented to provide the easiest way of consuming stream data. Developers that require more fine-grained control over the transfer and generation of data can use the `EventEmitter` and `readable.on('readable') / readable.read()` or the `readable.pause() / readable.resume()` APIs.

Class: `stream.Readable`

Event: `'close'`

The `'close'` event is emitted when the stream and any of its underlying resources (a file descriptor, for example) have been closed. The event indicates that no more events will be emitted, and no further computation will occur.

A `Readable` stream will always emit the `'close'` event if it is created with the `emitClose` option.

Event: `'data'`

- `chunk` `<Buffer>` | `<string>` | `<any>` The chunk of data. For streams that are not operating in object mode, the chunk will be either a string or `Buffer`. For streams that are in object mode, the chunk can be any JavaScript value other than `null`.

The `'data'` event is emitted whenever the stream is relinquishing ownership of a chunk of data to a consumer. This may occur whenever the stream is switched in flowing mode by calling `readable.pipe()`, `readable.resume()`, or by attaching a listener callback to the `'data'` event. The `'data'` event will also be emitted whenever the `readable.read()` method is called and a chunk of data is available to be returned.

Attaching a `'data'` event listener to a stream that has not been explicitly paused will switch the stream into flowing mode. Data will then be passed as soon as it is available.

The listener callback will be passed the chunk of data as a string if a default encoding has been specified for the stream using the `readable.setEncoding()` method; otherwise the data will be passed as a `Buffer`.

```
const readable = getReadableStreamSomehow();
readable.on('data', (chunk) => {
  console.log(`Received ${chunk.length} bytes of data.`);
});
```

Event: `'end'`

The `'end'` event is emitted when there is no more data to be consumed from the stream.

The `'end'` event **will not be emitted** unless the data is completely consumed. This can be accomplished by switching the stream into flowing mode, or by calling `stream.read()` repeatedly until all data has been consumed.

```
const readable = getReadableStreamSomehow();
readable.on('data', (chunk) => {
  console.log(`Received ${chunk.length} bytes of data.`);
});
readable.on('end', () => {
  console.log('There will be no more data.');
```

Event: `'error'`

- `<Error>`

The `'error'` event may be emitted by a `Readable` implementation at any time. Typically, this may occur if the underlying stream is unable to generate data due to an underlying internal failure, or when a stream implementation attempts to push an invalid chunk of data.

The listener callback will be passed a single `Error` object.

Event: `'pause'`

The `'pause'` event is emitted when `stream.pause()` is called and `readableFlowing` is not `false`.

Event: `'readable'`

The `'readable'` event is emitted when there is data available to be read from the stream. In some cases, attaching a listener for the `'readable'` event will cause some amount of data to be read into an internal buffer.


```
const readable = getReadableStreamSomehow();
readable.on('readable', function() {
  // There is some data to read now.
  let data;

  while (data = this.read()) {
    console.log(data);
  }
});
```

The `'readable'` event will also be emitted once the end of the stream data has been reached but before the `'end'` event is emitted.

Effectively, the `'readable'` event indicates that the stream has new information: either new data is available or the end of the stream has been reached. In the former case, `stream.read()` will return the available data. In the latter case, `stream.read()` will return `null`. For instance, in the following example, `foo.txt` is an empty file:

```
const fs = require('fs');
const rr = fs.createReadStream('foo.txt');
rr.on('readable', () => {
  console.log(`readable: ${rr.read()}`);
});
rr.on('end', () => {
  console.log('end');
});
```

The output of running this script is:

```
$ node test.js
readable: null
end
```

In general, the `readable.pipe()` and `'data'` event mechanisms are easier to understand than the `'readable'` event. However, handling `'readable'` might result in increased throughput.

If both `'readable'` and `'data'` are used at the same time, `'readable'` takes precedence in controlling the flow, i.e. `'data'` will be emitted only when `stream.read()` is called. The `readableFlowing` property would become `false`. If there are `'data'` listeners when `'readable'` is removed, the stream will start flowing, i.e. `'data'` events will be emitted without calling `.resume()`.

Event: `'resume'`

The `'resume'` event is emitted when `stream.resume()` is called and `readableFlowing` is not `true`.

`readable.destroy([error])`

- `error` `<Error>` Error which will be passed as payload in `'error'` event
- Returns: `<this>`

Destroy the stream. Optionally emit an `'error'` event, and emit a `'close'` event (unless `emitClose` is set to `false`). After this call, the readable stream will release any internal resources and subsequent calls to `push()` will be ignored.

Once `destroy()` has been called any further calls will be a no-op and no further errors except from `_destroy()` may be emitted as `'error'`.

Implementors should not override this method, but instead implement `readable._destroy()`.

`readable.destroyed`

- `<boolean>`

Is `true` after `readable.destroy()` has been called.

`readable.isPaused()`

- Returns: `<boolean>`

The `readable.isPaused()` method returns the current operating state of the `Readable`. This is used primarily by the mechanism that underlies the `readable.pipe()` method. In most typical cases, there will be no reason to use this method directly.

```
const readable = new stream.Readable();

readable.isPaused(); // === false
readable.pause();
readable.isPaused(); // === true
readable.resume();
readable.isPaused(); // === false
```

`readable.pause()`

- Returns: `<this>`

The `readable.pause()` method will cause a stream in flowing mode to stop emitting `'data'` events, switching out of flowing mode. Any data that becomes available will remain in the internal buffer.

```
const readable = getReadableStreamSomehow();
readable.on('data', (chunk) => {
  console.log(`Received ${chunk.length} bytes of data.`);
  readable.pause();
  console.log('There will be no additional data for 1 second.');
```

```
  setTimeout(() => {
    console.log('Now data will start flowing again.');
```

```
    readable.resume();
  }, 1000);
});
```

The `readable.pause()` method has no effect if there is a `'readable'` event listener.

`readable.pipe(destination[, options])`

- `destination` `<stream.Writable>` The destination for writing data
- `options` `<Object>` Pipe options
 - `end` `<boolean>` End the writer when the reader ends. **Default:** `true`.
- Returns: `<stream.Writable>` The *destination*, allowing for a chain of pipes if it is a `Duplex` or a `Transform` stream

The `readable.pipe()` method attaches a `Writable` stream to the `readable`, causing it to switch automatically into flowing mode and push all of its data to the attached `Writable`. The flow of data will be automatically managed so that the destination `Writable` stream is not overwhelmed by a faster `Readable` stream.

The following example pipes all of the data from the `readable` into a file named `file.txt`:

```
const fs = require('fs');
const readable = getReadableStreamSomehow();
const writable = fs.createWriteStream('file.txt');
// All the data from readable goes into 'file.txt'.
readable.pipe(writable);
```

It is possible to attach multiple `Writable` streams to a single `Readable` stream.

The `readable.pipe()` method returns a reference to the *destination* stream making it possible to set up chains of piped streams:

```
const fs = require('fs');
const r = fs.createReadStream('file.txt');
const z = zlib.createGzip();
const w = fs.createWriteStream('file.txt.gz');
r.pipe(z).pipe(w);
```

By default, `stream.end()` is called on the destination `Writable` stream when the source `Readable` stream emits `'end'`, so that the destination is no longer writable. To disable this default behavior, the `end` option can be passed as `false`, causing the destination stream to remain open:

```
reader.pipe(writer, { end: false });
reader.on('end', () => {
  writer.end('Goodbye\n');
});
```

One important caveat is that if the `Readable` stream emits an error during processing, the `writable` destination *is not closed* automatically. If an error occurs, it will be necessary to *manually* close each stream in order to prevent memory leaks.

The `process.stderr` and `process.stdout` `Writable` streams are never closed until the Node.js process exits, regardless of the specified options.

`readable.read([size])`

- `size` `<number>` Optional argument to specify how much data to read.
- Returns: `<string>` | `<Buffer>` | `<null>` | `<any>`

The `readable.read()` method pulls some data out of the internal buffer and returns it. If no data available to be read, `null` is returned. By default, the data will be returned as a `Buffer` object unless an encoding has been specified using the `readable.setEncoding()` method or the stream is operating in object mode.

The optional `size` argument specifies a specific number of bytes to read. If `size` bytes are not available to be read, `null` will be returned *unless* the stream has ended, in which case all of the data remaining in the internal buffer will be returned.

If the `size` argument is not specified, all of the data contained in the internal buffer will be returned.

The `size` argument must be less than or equal to 1 GiB.

The `readable.read()` method should only be called on `Readable` streams operating in paused mode. In flowing mode, `readable.read()` is called automatically until the internal buffer is fully drained.

```
const readable = getReadableStreamSomehow();

// 'readable' may be triggered multiple times as data is buffered in
readable.on('readable', () => {
  let chunk;
  console.log('Stream is readable (new data received in buffer)');
  // Use a loop to make sure we read all currently available data
  while (null !== (chunk = readable.read())) {
    console.log(`Read ${chunk.length} bytes of data...`);
  }
});

// 'end' will be triggered once when there is no more data available
readable.on('end', () => {
  console.log('Reached end of stream.');
```

Each call to `readable.read()` returns a chunk of data, or `null`. The chunks are not concatenated. A `while` loop is necessary to consume all data currently in the buffer. When reading a large file `.read()` may return `null`, having consumed all buffered content so far, but there is still more data to come not yet buffered. In this case a new `'readable'` event will be emitted when there is more data in the buffer. Finally the `'end'` event will be emitted when there is no more data to come.

Therefore to read a file's whole contents from a `readable`, it is necessary to collect chunks across multiple `'readable'` events:

```
const chunks = [];

readable.on('readable', () => {
  let chunk;
  while (null !== (chunk = readable.read())) {
    chunks.push(chunk);
  }
});

readable.on('end', () => {
  const content = chunks.join('');
```

A `Readable` stream in object mode will always return a single item from a call to `readable.read(size)`, regardless of the value of the `size` argument.

If the `readable.read()` method returns a chunk of data, a `'data'` event will also be emitted.

Calling `stream.read([size])` after the `'end'` event has been emitted will return `null`. No runtime error will be raised.

`readable.readable`

- `<boolean>`

Is `true` if it is safe to call `readable.read()`, which means the stream has not been destroyed or emitted `'error'` or `'end'`.

`readable.readableDidRead`

- `<boolean>`

Allows determining if the stream has been or is about to be read. Returns true if `'data'`, `'end'`, `'error'` or `'close'` has been emitted.

`readable.readableEncoding`

- `<null> | <string>`

Getter for the property `encoding` of a given `Readable` stream. The `encoding` property can be set using the `readable.setEncoding()` method.

`readable.readableEnded`

- `<boolean>`

Becomes `true` when `'end'` event is emitted.

`readable.readableFlowing`

- `<boolean>`

This property reflects the current state of a `Readable` stream as described in the [Three states](#) section.

`readable.readableHighWaterMark`

- `<number>`

Returns the value of `highWaterMark` passed when creating this `Readable`.

`readable.readableLength`

- `<number>`

This property contains the number of bytes (or objects) in the queue ready to be read. The value provides introspection data regarding the status of the `highWaterMark`.

`readable.readableObjectMode`

- `<boolean>`

Getter for the property `objectMode` of a given `Readable` stream.

`readable.resume()`

- Returns: `<this>`

The `readable.resume()` method causes an explicitly paused `Readable` stream to resume emitting `'data'` events, switching the stream into flowing mode.

The `readable.resume()` method can be used to fully consume the data from a stream without actually processing any of that data:

```
getReadableStreamSomehow()
  .resume()
  .on('end', () => {
    console.log('Reached the end, but did not read anything.');
```

The `readable.resume()` method has no effect if there is a `'readable'` event listener.

`readable.setEncoding(encoding)`

- `encoding` `<string>` The encoding to use.
- Returns: `<this>`

The `readable.setEncoding()` method sets the character encoding for data read from the `Readable` stream.

By default, no encoding is assigned and stream data will be returned as `Buffer` objects. Setting an encoding causes the stream data to be returned as strings of the specified encoding rather than as `Buffer` objects. For instance, calling `readable.setEncoding('utf8')` will cause the output data to be interpreted as UTF-8 data, and passed as strings. Calling `readable.setEncoding('hex')` will cause the data to be encoded in hexadecimal string format.

The `Readable` stream will properly handle multi-byte characters delivered through the stream that would otherwise become improperly decoded if simply pulled from the stream as `Buffer` objects.

```
const readable = getReadableStreamSomehow();
readable.setEncoding('utf8');
readable.on('data', (chunk) => {
  assert.equal(typeof chunk, 'string');
  console.log('Got %d characters of string data:', chunk.length);
});
```

`readable.unpipe([destination])`

- `destination` `<stream.Writable>` Optional specific stream to unpipe
- Returns: `<this>`

The `readable.unpipe()` method detaches a `Writable` stream previously attached using the `stream.pipe()` method.

If the `destination` is not specified, then *all* pipes are detached.

If the `destination` is specified, but no pipe is set up for it, then the method does nothing.

```
const fs = require('fs');
const readable = getReadableStreamSomehow();
const writable = fs.createWriteStream('file.txt');
// All the data from readable goes into 'file.txt',
// but only for the first second.
readable.pipe(writable);
setTimeout(() => {
  console.log('Stop writing to file.txt.');
  readable.unpipe(writable);
  console.log('Manually close the file stream.');
  writable.end();
}, 1000);
```

`readable.unshift(chunk[, encoding])`

- `chunk` `<Buffer>` | `<Uint8Array>` | `<string>` | `<null>` | `<any>` Chunk of data to unshift onto the read queue. For streams not operating in object mode, `chunk` must be a string, `Buffer`, `Uint8Array` or `null`. For object mode streams, `chunk` may be any JavaScript value.
- `encoding` `<string>` Encoding of string chunks. Must be a valid `Buffer` encoding, such as `'utf8'` or `'ascii'`.

Passing `chunk` as `null` signals the end of the stream (EOF) and behaves the same as `readable.push(null)`, after which no more data can be written. The EOF signal is put at the end of the buffer and any buffered data will still be flushed.

The `readable.unshift()` method pushes a chunk of data back into the internal buffer. This is useful in certain situations where a stream is being consumed by code that needs to "un-consume" some amount of data that it has optimistically pulled out of the source, so that the data can be passed on to some other party.

The `stream.unshift(chunk)` method cannot be called after the `'end'` event has been emitted or a runtime error will be thrown.

Developers using `stream.unshift()` often should consider switching to use of a `Transform` stream instead. See the [API for stream implementers](#) section for more information.

```
// Pull off a header delimited by \n\n.
// Use unshift() if we get too much.
// Call the callback with (error, header, stream).
const { StringDecoder } = require('string_decoder');
function parseHeader(stream, callback) {
  stream.on('error', callback);
  stream.on('readable', onReadable);
  const decoder = new StringDecoder('utf8');
  let header = '';
  function onReadable() {
    let chunk;
    while (null !== (chunk = stream.read())) {
      const str = decoder.write(chunk);
      if (str.match(/\n\n/)) {
        // Found the header boundary.
        const split = str.split(/\n\n/);
        header += split.shift();
        const remaining = split.join('\n\n');
        const buf = Buffer.from(remaining, 'utf8');
        stream.removeListener('error', callback);
        // Remove the 'readable' listener before unshifting.
        stream.removeListener('readable', onReadable);
        if (buf.length)
          stream.unshift(buf);
        // Now the body of the message can be read from the stream.
        callback(null, header, stream);
      } else {
        // Still reading the header.
        header += str;
      }
    }
  }
}
```

Unlike `stream.push(chunk)`, `stream.unshift(chunk)` will not end the reading process by resetting the internal reading state of the stream. This can cause unexpected results if `readable.unshift()` is called during a read (i.e. from within a `stream._read()` implementation on a custom stream). Following the call to `readable.unshift()` with an immediate `stream.push('')` will reset the reading state appropriately, however it is best to simply avoid calling `readable.unshift()` while in the process of performing a read.

`readable.wrap(stream)`

- `stream` `<Stream>` An "old style" readable stream
- Returns: `<this>`

Prior to Node.js 0.10, streams did not implement the entire `stream` module API as it is currently defined. (See [Compatibility](#) for more information.)

When using an older Node.js library that emits `'data'` events and has a `stream.pause()` method that is advisory only, the `readable.wrap()` method can be used to create a `Readable` stream that uses the old stream as its data source.

It will rarely be necessary to use `readable.wrap()` but the method has been provided as a convenience for interacting with older Node.js applications and libraries.

```
const { OldReader } = require('./old-api-module.js');
const { Readable } = require('stream');
const oreader = new OldReader();
const myReader = new Readable().wrap(oreader);

myReader.on('readable', () => {
  myReader.read(); // etc.
});
```

`readable[Symbol.asyncIterator]()`

- Returns: `<AsyncIterator>` to fully consume the stream.

```
const fs = require('fs');

async function print(readable) {
  readable.setEncoding('utf8');
  let data = '';
  for await (const chunk of readable) {
    data += chunk;
  }
  console.log(data);
}

print(fs.createReadStream('file')).catch(console.error);
```

If the loop terminates with a `break`, `return`, or a `throw`, the stream will be destroyed. In other terms, iterating over a stream will consume the stream fully. The stream will be read in chunks of size equal to the `highWaterMark` option. In the code example above, data will be in a single chunk if the file has less than 64 KB of data because no `highWaterMark` option is provided to `fs.createReadStream()`.

`readable.iterator([options])`

Stability: 1 - Experimental

- `options` `<Object>`
 - `destroyOnReturn` `<boolean>` When set to `false`, calling `return` on the async iterator, or exiting a `for await...of` iteration using a `break`, `return`, or `throw` will not destroy the stream. **Default:** `true`.

- `destroyOnError` `<boolean>` When set to `false`, if the stream emits an error while it's being iterated, the iterator will not destroy the stream. **Default:** `true`.
- Returns: `<AsyncIterator>` to consume the stream.

The iterator created by this method gives users the option to cancel the destruction of the stream if the `for await...of` loop is exited by `return`, `break`, or `throw`, or if the iterator should destroy the stream if the stream emitted an error during iteration.

```
const { Readable } = require('stream');

async function printIterator(readable) {
  for await (const chunk of readable.iterator({ destroyOnReturn: false })) {
    console.log(chunk); // 1
    break;
  }

  console.log(readable.destroyed); // false

  for await (const chunk of readable.iterator({ destroyOnReturn: false })) {
    console.log(chunk); // Will print 2 and then 3
  }

  console.log(readable.destroyed); // True, stream was totally consumed
}

async function printSymbolAsyncIterator(readable) {
  for await (const chunk of readable) {
    console.log(chunk); // 1
    break;
  }

  console.log(readable.destroyed); // true
}

async function showBoth() {
  await printIterator(Readable.from([1, 2, 3]));
  await printSymbolAsyncIterator(Readable.from([1, 2, 3]));
}

showBoth();
```

Duplex and transform streams

Class: `stream.Duplex`

Duplex streams are streams that implement both the `Readable` and `Writable` interfaces.

Examples of `Duplex` streams include:

- `TCP` sockets
- `zlib` streams
- `crypto` streams

Class: `stream.Transform`

Transform streams are `Duplex` streams where the output is in some way related to the input. Like all `Duplex` streams, `Transform` streams implement both the `Readable` and `Writable` interfaces.

Examples of `Transform` streams include:

- `zlib` streams
- `crypto` streams

`transform.destroy([error])`

- `error` `<Error>`
- Returns: `<this>`

Destroy the stream, and optionally emit an `'error'` event. After this call, the transform stream would release any internal resources. Implementors should not override this method, but instead implement `readable._destroy()`. The default implementation of `_destroy()` for `Transform` also emit `'close'` unless `emitClose` is set in false.

Once `destroy()` has been called, any further calls will be a no-op and no further errors except from `_destroy()` may be emitted as `'error'`.

`stream.finished(stream[, options], callback)`

- `stream` `<Stream>` A readable and/or writable stream.
- `options` `<Object>`
 - `error` `<boolean>` If set to `false`, then a call to `emit('error', err)` is not treated as finished. **Default:** `true`.
 - `readable` `<boolean>` When set to `false`, the callback will be called when the stream ends even though the stream might still be readable. **Default:** `true`.
 - `writable` `<boolean>` When set to `false`, the callback will be called when the stream ends even though the stream might still be writable. **Default:** `true`.
 - `signal` `<AbortSignal>` allows aborting the wait for the stream finish. The underlying stream will *not* be aborted if the signal is aborted. The callback will get called with an `AbortError`. All registered listeners added by this function will also be removed.
- `callback` `<Function>` A callback function that takes an optional error argument.
- Returns: `<Function>` A cleanup function which removes all registered listeners.

A function to get notified when a stream is no longer readable, writable or has experienced an error or a premature close event.

```
const { finished } = require('stream');

const rs = fs.createReadStream('archive.tar');

finished(rs, (err) => {
  if (err) {
    console.error('Stream failed.', err);
  } else {
    console.log('Stream is done reading.');
```

Especially useful in error handling scenarios where a stream is destroyed prematurely (like an aborted HTTP request), and will not emit `'end'` or `'finish'`.

The `finished` API provides promise version:

```
const { finished } = require('stream/promises');

const rs = fs.createReadStream('archive.tar');

async function run() {
  await finished(rs);
  console.log('Stream is done reading.');
```



```
}

run().catch(console.error);
rs.resume(); // Drain the stream.
```

`stream.finished()` leaves dangling event listeners (in particular `'error'`, `'end'`, `'finish'` and `'close'`) after `callback` has been invoked. The reason for this is so that unexpected `'error'` events (due to incorrect stream implementations) do not cause unexpected crashes. If this is unwanted behavior then the returned cleanup function needs to be invoked in the callback:

```
const cleanup = finished(rs, (err) => {
  cleanup();
  // ...
});
```

`stream.pipeline(source[, ...transforms], destination, callback)`

`stream.pipeline(streams, callback)`

- `streams` `<Stream[]> | <Iterable[]> | <AsyncIterable[]> | <Function[]>`
- `source` `<Stream> | <Iterable> | <AsyncIterable> | <Function>`
 - Returns: `<Iterable> | <AsyncIterable>`
- `...transforms` `<Stream> | <Function>`
 - `source` `<AsyncIterable>`
 - Returns: `<AsyncIterable>`
- `destination` `<Stream> | <Function>`
 - `source` `<AsyncIterable>`
 - Returns: `<AsyncIterable> | <Promise>`
- `callback` `<Function>` Called when the pipeline is fully done.
 - `err` `<Error>`
 - `val` Resolved value of `Promise` returned by `destination`.
- Returns: `<Stream>`

A module method to pipe between streams and generators forwarding errors and properly cleaning up and provide a callback when the pipeline is complete.

```
const { pipeline } = require('stream');
const fs = require('fs');
```

```

const zlib = require('zlib');

// Use the pipeline API to easily pipe a series of streams
// together and get notified when the pipeline is fully done.

// A pipeline to gzip a potentially huge tar file efficiently:

pipeline(
  fs.createReadStream('archive.tar'),
  zlib.createGzip(),
  fs.createWriteStream('archive.tar.gz'),
  (err) => {
    if (err) {
      console.error('Pipeline failed.', err);
    } else {
      console.log('Pipeline succeeded.');
    }
  }
);

```

The `pipeline` API provides a promise version, which can also receive an options argument as the last parameter with a `signal` `<AbortSignal>` property. When the signal is aborted, `destroy` will be called on the underlying pipeline, with an `AbortError`.

```

const { pipeline } = require('stream/promises');

async function run() {
  await pipeline(
    fs.createReadStream('archive.tar'),
    zlib.createGzip(),
    fs.createWriteStream('archive.tar.gz')
  );
  console.log('Pipeline succeeded.');
}

run().catch(console.error);

```

To use an `AbortSignal`, pass it inside an options object, as the last argument:

```

const { pipeline } = require('stream/promises');

async function run() {
  const ac = new AbortController();
  const options = {
    signal: ac.signal,
  };

  setTimeout(() => ac.abort(), 1);
  await pipeline(
    fs.createReadStream('archive.tar'),
    zlib.createGzip(),

```

```

    fs.createWriteStream('archive.tar.gz'),
    options,
  );
}

run().catch(console.error); // AbortError

```

The `pipeline` API also supports async generators:

```

const { pipeline } = require('stream/promises');
const fs = require('fs');

async function run() {
  await pipeline(
    fs.createReadStream('lowercase.txt'),
    async function* (source) {
      source.setEncoding('utf8'); // Work with strings rather than `Buffer`s.
      for await (const chunk of source) {
        yield chunk.toUpperCase();
      }
    },
    fs.createWriteStream('uppercase.txt')
  );
  console.log('Pipeline succeeded.');
```

```

}

run().catch(console.error);

```

`stream.pipeline()` will call `stream.destroy(err)` on all streams except:

- `Readable` streams which have emitted `'end'` or `'close'`.
- `Writable` streams which have emitted `'finish'` or `'close'`.

`stream.pipeline()` leaves dangling event listeners on the streams after the `callback` has been invoked. In the case of reuse of streams after failure, this can cause event listener leaks and swallowed errors.

`stream.Readable.from(iterable, [options])`

- `iterable` `<Iterable>` Object implementing the `Symbol.asyncIterator` or `Symbol.iterator` iterable protocol. Emits an `'error'` event if a null value is passed.
- `options` `<Object>` Options provided to `new stream.Readable([options])`. By default, `Readable.from()` will set `options.objectMode` to `true`, unless this is explicitly opted out by setting `options.objectMode` to `false`.
- Returns: `<stream.Readable>`

A utility method for creating readable streams out of iterators.

```

const { Readable } = require('stream');

async function * generate() {
  yield 'hello';
  yield 'streams';
}

```

```
const readable = Readable.from(generate());

readable.on('data', (chunk) => {
  console.log(chunk);
});
```

Calling `Readable.from(string)` or `Readable.from(buffer)` will not have the strings or buffers be iterated to match the other streams semantics for performance reasons.

stream.addAbortSignal(signal, stream)

- `signal` `<AbortSignal>` A signal representing possible cancellation
- `stream` `<Stream>` a stream to attach a signal to

Attaches an `AbortSignal` to a readable or writable stream. This lets code control stream destruction using an `AbortController`.

Calling `abort` on the `AbortController` corresponding to the passed `AbortSignal` will behave the same way as calling `.destroy(new AbortError())` on the stream.

```
const fs = require('fs');

const controller = new AbortController();
const read = addAbortSignal(
  controller.signal,
  fs.createReadStream('object.json'))
);
// Later, abort the operation closing the stream
controller.abort();
```

Or using an `AbortSignal` with a readable stream as an async iterable:

```
const controller = new AbortController();
setTimeout(() => controller.abort(), 10_000); // set a timeout
const stream = addAbortSignal(
  controller.signal,
  fs.createReadStream('object.json'))
);
(async () => {
  try {
    for await (const chunk of stream) {
      await process(chunk);
    }
  } catch (e) {
    if (e.name === 'AbortError') {
      // The operation was cancelled
    } else {
      throw e;
    }
  }
})();
```

API for stream implementers

The `stream` module API has been designed to make it possible to easily implement streams using JavaScript's prototypal inheritance model.

First, a stream developer would declare a new JavaScript class that extends one of the four basic stream classes (`stream.Writable`, `stream.Readable`, `stream.Duplex`, or `stream.Transform`), making sure they call the appropriate parent class constructor:

```
const { Writable } = require('stream');

class MyWritable extends Writable {
  constructor({ highWaterMark, ...options }) {
    super({ highWaterMark });
    // ...
  }
}
```

When extending streams, keep in mind what options the user can and should provide before forwarding these to the base constructor. For example, if the implementation makes assumptions in regard to the `autoDestroy` and `emitClose` options, do not allow the user to override these. Be explicit about what options are forwarded instead of implicitly forwarding all options.

The new stream class must then implement one or more specific methods, depending on the type of stream being created, as detailed in the chart below:

Use-case	Class	Method(s) to implement
Reading only	<code>Readable</code>	<code>_read()</code>
Writing only	<code>Writable</code>	<code>_write()</code> , <code>_writev()</code> , <code>_final()</code>
Reading and writing	<code>Duplex</code>	<code>_read()</code> , <code>_write()</code> , <code>_writev()</code> , <code>_final()</code>
Operate on written data, then read the result	<code>Transform</code>	<code>_transform()</code> , <code>_flush()</code> , <code>_final()</code>

The implementation code for a stream should *never* call the "public" methods of a stream that are intended for use by consumers (as described in the [API for stream consumers](#) section). Doing so may lead to adverse side effects in application code consuming the stream.

Avoid overriding public methods such as `write()`, `end()`, `cork()`, `uncork()`, `read()` and `destroy()`, or emitting internal events such as `'error'`, `'data'`, `'end'`, `'finish'` and `'close'` through `.emit()`. Doing so can break current and future stream invariants leading to behavior and/or compatibility issues with other streams, stream utilities, and user expectations.

Simplified construction

For many simple cases, it is possible to create a stream without relying on inheritance. This can be accomplished by directly creating instances of the `stream.Writable`, `stream.Readable`, `stream.Duplex` or `stream.Transform` objects and passing appropriate methods as constructor options.

```
const { Writable } = require('stream');

const myWritable = new Writable({
  construct(callback) {
    // Initialize state and load resources...
  },
});
```

```

write(chunk, encoding, callback) {
  // ...
},
destroy() {
  // Free resources...
}
});

```

Implementing a writable stream

The `stream.Writable` class is extended to implement a `Writable` stream.

Custom `Writable` streams *must* call the `new stream.Writable([options])` constructor and implement the `writable._write()` and/or `writable._writev()` method.

`new stream.Writable([options])`

- `options` `<Object>`
 - `highWaterMark` `<number>` Buffer level when `stream.write()` starts returning `false`. **Default:** 16384 (16 KB), or 16 for `objectMode` streams.
 - `decodeStrings` `<boolean>` Whether to encode `string`s passed to `stream.write()` to `Buffer`s (with the encoding specified in the `stream.write()` call) before passing them to `stream._write()`. Other types of data are not converted (i.e. `Buffer`s are not decoded into `string`s). Setting to false will prevent `string`s from being converted. **Default:** `true`.
 - `defaultEncoding` `<string>` The default encoding that is used when no encoding is specified as an argument to `stream.write()`. **Default:** `'utf8'`.
 - `objectMode` `<boolean>` Whether or not the `stream.write(anyObj)` is a valid operation. When set, it becomes possible to write JavaScript values other than `string`, `Buffer` or `Uint8Array` if supported by the stream implementation. **Default:** `false`.
 - `emitClose` `<boolean>` Whether or not the stream should emit `'close'` after it has been destroyed. **Default:** `true`.
 - `write` `<Function>` Implementation for the `stream._write()` method.
 - `writev` `<Function>` Implementation for the `stream._writev()` method.
 - `destroy` `<Function>` Implementation for the `stream._destroy()` method.
 - `final` `<Function>` Implementation for the `stream._final()` method.
 - `construct` `<Function>` Implementation for the `stream._construct()` method.
 - `autoDestroy` `<boolean>` Whether this stream should automatically call `.destroy()` on itself after ending. **Default:** `true`.
 - `signal` `<AbortSignal>` A signal representing possible cancellation.

```

const { Writable } = require('stream');

class MyWritable extends Writable {
  constructor(options) {
    // Calls the stream.Writable() constructor.
    super(options);
    // ...
  }
}

```

Or, when using pre-ES6 style constructors:


```
const { Writable } = require('stream');
const util = require('util');

function MyWritable(options) {
  if (!(this instanceof MyWritable))
    return new MyWritable(options);
  Writable.call(this, options);
}
util.inherits(MyWritable, Writable);
```

Or, using the simplified constructor approach:

```
const { Writable } = require('stream');

const myWritable = new Writable({
  write(chunk, encoding, callback) {
    // ...
  },
  writev(chunks, callback) {
    // ...
  }
});
```

Calling `abort` on the `AbortController` corresponding to the passed `AbortSignal` will behave the same way as calling `.destroy(new AbortError())` on the writable stream.

```
const { Writable } = require('stream');

const controller = new AbortController();
const myWritable = new Writable({
  write(chunk, encoding, callback) {
    // ...
  },
  writev(chunks, callback) {
    // ...
  },
  signal: controller.signal
});
// Later, abort the operation closing the stream
controller.abort();
```

writable._construct(callback)

- `callback` `<Function>` Call this function (optionally with an error argument) when the stream has finished initializing.

The `_construct()` method MUST NOT be called directly. It may be implemented by child classes, and if so, will be called by the internal `Writable` class methods only.

This optional function will be called in a tick after the stream constructor has returned, delaying any `_write()`, `_final()` and `_destroy()` calls until `callback` is called. This is useful to initialize state or asynchronously initialize resources before the stream can be used.

```

const { Writable } = require('stream');
const fs = require('fs');

class WriteStream extends Writable {
  constructor(filename) {
    super();
    this.filename = filename;
  }
  _construct(callback) {
    fs.open(this.filename, (err, fd) => {
      if (err) {
        callback(err);
      } else {
        this.fd = fd;
        callback();
      }
    });
  }
  _write(chunk, encoding, callback) {
    fs.write(this.fd, chunk, callback);
  }
  _destroy(err, callback) {
    if (this.fd) {
      fs.close(this.fd, (er) => callback(er || err));
    } else {
      callback(err);
    }
  }
}

```

writable._write(chunk, encoding, callback)

- **chunk** `<Buffer> | <string> | <any>` The `Buffer` to be written, converted from the `string` passed to `stream.write()`. If the stream's `decodeStrings` option is `false` or the stream is operating in object mode, the chunk will not be converted & will be whatever was passed to `stream.write()`.
- **encoding** `<string>` If the chunk is a string, then `encoding` is the character encoding of that string. If chunk is a `Buffer`, or if the stream is operating in object mode, `encoding` may be ignored.
- **callback** `<Function>` Call this function (optionally with an error argument) when processing is complete for the supplied chunk.

All `Writable` stream implementations must provide a `writable._write()` and/or `writable._writev()` method to send data to the underlying resource.

`Transform` streams provide their own implementation of the `writable._write()`.

This function MUST NOT be called by application code directly. It should be implemented by child classes, and called by the internal `Writable` class methods only.

The `callback` function must be called synchronously inside of `writable._write()` or asynchronously (i.e. different tick) to signal either that the write completed successfully or failed with an error. The first argument passed to the `callback` must be the `Error` object if the call failed or `null` if the write succeeded.

All calls to `writable.write()` that occur between the time `writable._write()` is called and the `callback` is called will cause the written data to be buffered. When the `callback` is invoked, the stream might emit a `'drain'` event. If a stream implementation is capable of processing multiple chunks of data at once, the `writable._writew()` method should be implemented.

If the `decodeStrings` property is explicitly set to `false` in the constructor options, then `chunk` will remain the same object that is passed to `.write()`, and may be a string rather than a `Buffer`. This is to support implementations that have an optimized handling for certain string data encodings. In that case, the `encoding` argument will indicate the character encoding of the string. Otherwise, the `encoding` argument can be safely ignored.

The `writable._write()` method is prefixed with an underscore because it is internal to the class that defines it, and should never be called directly by user programs.

`writable._writew(chunks, callback)`

- `chunks` `<Object[]>` The data to be written. The value is an array of `<Object>` that each represent a discrete chunk of data to write. The properties of these objects are:
 - `chunk` `<Buffer>` | `<string>` A buffer instance or string containing the data to be written. The `chunk` will be a string if the `writable` was created with the `decodeStrings` option set to `false` and a string was passed to `write()`.
 - `encoding` `<string>` The character encoding of the `chunk`. If `chunk` is a `Buffer`, the `encoding` will be `'buffer'`.
- `callback` `<Function>` A callback function (optionally with an error argument) to be invoked when processing is complete for the supplied chunks.

This function MUST NOT be called by application code directly. It should be implemented by child classes, and called by the internal `writable` class methods only.

The `writable._writew()` method may be implemented in addition or alternatively to `writable._write()` in stream implementations that are capable of processing multiple chunks of data at once. If implemented and if there is buffered data from previous writes, `_writew()` will be called instead of `_write()`.

The `writable._writew()` method is prefixed with an underscore because it is internal to the class that defines it, and should never be called directly by user programs.

`writable._destroy(err, callback)`

- `err` `<Error>` A possible error.
- `callback` `<Function>` A callback function that takes an optional error argument.

The `_destroy()` method is called by `writable.destroy()`. It can be overridden by child classes but it **must not** be called directly.

`writable._final(callback)`

- `callback` `<Function>` Call this function (optionally with an error argument) when finished writing any remaining data.

The `_final()` method **must not** be called directly. It may be implemented by child classes, and if so, will be called by the internal `writable` class methods only.

This optional function will be called before the stream closes, delaying the `'finish'` event until `callback` is called. This is useful to close resources or write buffered data before a stream ends.

Errors while writing

Errors occurring during the processing of the `writable._write()`, `writable._writew()` and `writable._final()` methods must be propagated by invoking the callback and passing the error as the first argument. Throwing an `Error` from within these methods or manually emitting an `'error'` event results in undefined behavior.

If a `Readable` stream pipes into a `Writable` stream when `Writable` emits an error, the `Readable` stream will be unpipe.

```

const { Writable } = require('stream');

const myWritable = new Writable({
  write(chunk, encoding, callback) {
    if (chunk.toString().indexOf('a') >= 0) {
      callback(new Error('chunk is invalid'));
    } else {
      callback();
    }
  }
});

```

An example writable stream

The following illustrates a rather simplistic (and somewhat pointless) custom `Writable` stream implementation. While this specific `Writable` stream instance is not of any real particular usefulness, the example illustrates each of the required elements of a custom `Writable` stream instance:

```

const { Writable } = require('stream');

class MyWritable extends Writable {
  _write(chunk, encoding, callback) {
    if (chunk.toString().indexOf('a') >= 0) {
      callback(new Error('chunk is invalid'));
    } else {
      callback();
    }
  }
}

```

Decoding buffers in a writable stream

Decoding buffers is a common task, for instance, when using transformers whose input is a string. This is not a trivial process when using multi-byte characters encoding, such as UTF-8. The following example shows how to decode multi-byte strings using `StringDecoder` and `Writable`.

```

const { Writable } = require('stream');
const { StringDecoder } = require('string_decoder');

class StringWritable extends Writable {
  constructor(options) {
    super(options);
    this._decoder = new StringDecoder(options && options.defaultEncoding);
    this.data = '';
  }
  _write(chunk, encoding, callback) {
    if (encoding === 'buffer') {
      chunk = this._decoder.write(chunk);
    }
    this.data += chunk;
  }
}

```

```

    callback();
  }
  _final(callback) {
    this.data += this._decoder.end();
    callback();
  }
}

const euro = [[0xE2, 0x82], [0xAC]].map(Buffer.from);
const w = new StringWritable();

w.write('currency: ');
w.write(euro[0]);
w.end(euro[1]);

console.log(w.data); // currency: €

```

Implementing a readable stream

The `stream.Readable` class is extended to implement a `Readable` stream.

Custom `Readable` streams *must* call the `new stream.Readable([options])` constructor and implement the `readable._read()` method.

`new stream.Readable([options])`

- `options` `<Object>`
 - `highWaterMark` `<number>` The maximum `number of bytes` to store in the internal buffer before ceasing to read from the underlying resource. **Default:** 16384 (16 KB), or 16 for `objectMode` streams.
 - `encoding` `<string>` If specified, then buffers will be decoded to strings using the specified encoding. **Default:** `null`.
 - `objectMode` `<boolean>` Whether this stream should behave as a stream of objects. Meaning that `stream.read(n)` returns a single value instead of a `Buffer` of size `n`. **Default:** `false`.
 - `emitClose` `<boolean>` Whether or not the stream should emit `'close'` after it has been destroyed. **Default:** `true`.
 - `read` `<Function>` Implementation for the `stream._read()` method.
 - `destroy` `<Function>` Implementation for the `stream._destroy()` method.
 - `construct` `<Function>` Implementation for the `stream._construct()` method.
 - `autoDestroy` `<boolean>` Whether this stream should automatically call `.destroy()` on itself after ending. **Default:** `true`.
 - `signal` `<AbortSignal>` A signal representing possible cancellation.

```

const { Readable } = require('stream');

class MyReadable extends Readable {
  constructor(options) {
    // Calls the stream.Readable(options) constructor.
    super(options);
    // ...
  }
}

```

Or, when using pre-ES6 style constructors:

```

const { Readable } = require('stream');
const util = require('util');

function MyReadable(options) {
  if (!(this instanceof MyReadable))
    return new MyReadable(options);
  Readable.call(this, options);
}
util.inherits(MyReadable, Readable);

```

Or, using the simplified constructor approach:

```

const { Readable } = require('stream');

const myReadable = new Readable({
  read(size) {
    // ...
  }
});

```

Calling `abort` on the `AbortController` corresponding to the passed `AbortSignal` will behave the same way as calling `.destroy(new AbortError())` on the readable created.

```

const { Readable } = require('stream');
const controller = new AbortController();
const read = new Readable({
  read(size) {
    // ...
  },
  signal: controller.signal
});
// Later, abort the operation closing the stream
controller.abort();

```

readable._construct(callback)

- `callback` `<Function>` Call this function (optionally with an error argument) when the stream has finished initializing.

The `_construct()` method MUST NOT be called directly. It may be implemented by child classes, and if so, will be called by the internal `Readable` class methods only.

This optional function will be scheduled in the next tick by the stream constructor, delaying any `_read()` and `_destroy()` calls until `callback` is called. This is useful to initialize state or asynchronously initialize resources before the stream can be used.

```

const { Readable } = require('stream');
const fs = require('fs');

class ReadStream extends Readable {
  constructor(filename) {
    super();

```

```

    this.filename = filename;
    this.fd = null;
  }
  _construct(callback) {
    fs.open(this.filename, (err, fd) => {
      if (err) {
        callback(err);
      } else {
        this.fd = fd;
        callback();
      }
    });
  }
  _read(n) {
    const buf = Buffer.alloc(n);
    fs.read(this.fd, buf, 0, n, null, (err, bytesRead) => {
      if (err) {
        this.destroy(err);
      } else {
        this.push(bytesRead > 0 ? buf.slice(0, bytesRead) : null);
      }
    });
  }
  _destroy(err, callback) {
    if (this.fd) {
      fs.close(this.fd, (er) => callback(er || err));
    } else {
      callback(err);
    }
  }
}

```

readable._read(size)

- **size** `<number>` Number of bytes to read asynchronously

This function MUST NOT be called by application code directly. It should be implemented by child classes, and called by the internal `Readable` class methods only.

All `Readable` stream implementations must provide an implementation of the `readable._read()` method to fetch data from the underlying resource.

When `readable._read()` is called, if data is available from the resource, the implementation should begin pushing that data into the read queue using the `this.push(dataChunk)` method. `_read()` will be called again after each call to `this.push(dataChunk)` once the stream is ready to accept more data. `_read()` may continue reading from the resource and pushing data until `readable.push()` returns `false`. Only when `_read()` is called again after it has stopped should it resume pushing additional data into the queue.

Once the `readable._read()` method has been called, it will not be called again until more data is pushed through the `readable.push()` method. Empty data such as empty buffers and strings will not cause `readable._read()` to be called.

The `size` argument is advisory. For implementations where a "read" is a single operation that returns data can use the `size` argument to determine how much data to fetch. Other implementations may ignore this argument and simply provide data whenever it becomes available. There is no need to "wait" until `size` bytes are available before calling `stream.push(chunk)`.

The `readable._read()` method is prefixed with an underscore because it is internal to the class that defines it, and should never be called directly by user programs.

`readable._destroy(err, callback)`

- `err` `<Error>` A possible error.
- `callback` `<Function>` A callback function that takes an optional error argument.

The `_destroy()` method is called by `readable.destroy()`. It can be overridden by child classes but it **must not** be called directly.

`readable.push(chunk[, encoding])`

- `chunk` `<Buffer> | <Uint8Array> | <string> | <null> | <any>` Chunk of data to push into the read queue. For streams not operating in object mode, `chunk` must be a string, `Buffer` or `Uint8Array`. For object mode streams, `chunk` may be any JavaScript value.
- `encoding` `<string>` Encoding of string chunks. Must be a valid `Buffer` encoding, such as `'utf8'` or `'ascii'`.
- Returns: `<boolean>` `true` if additional chunks of data may continue to be pushed; `false` otherwise.

When `chunk` is a `Buffer`, `Uint8Array` or `string`, the `chunk` of data will be added to the internal queue for users of the stream to consume. Passing `chunk` as `null` signals the end of the stream (EOF), after which no more data can be written.

When the `Readable` is operating in paused mode, the data added with `readable.push()` can be read out by calling the `readable.read()` method when the `'readable'` event is emitted.

When the `Readable` is operating in flowing mode, the data added with `readable.push()` will be delivered by emitting a `'data'` event.

The `readable.push()` method is designed to be as flexible as possible. For example, when wrapping a lower-level source that provides some form of pause/resume mechanism, and a data callback, the low-level source can be wrapped by the custom `Readable` instance:

```
// `_source` is an object with readStop() and readStart() methods,
// and an `ondata` member that gets called when it has data, and
// an `onend` member that gets called when the data is over.

class SourceWrapper extends Readable {
  constructor(options) {
    super(options);

    this._source = getLowLevelSourceObject();

    // Every time there's data, push it into the internal buffer.
    this._source.ondata = (chunk) => {
      // If push() returns false, then stop reading from source.
      if (!this.push(chunk))
        this._source.readStop();
    };

    // When the source ends, push the EOF-signaling `null` chunk.
    this._source.onend = () => {
      this.push(null);
    };
  }

  // _read() will be called when the stream wants to pull more data in.
  // The advisory size argument is ignored in this case.
}
```



```

_read(size) {
  this._source.readStart();
}
}

```

The `readable.push()` method is used to push the content into the internal buffer. It can be driven by the `readable._read()` method.

For streams not operating in object mode, if the `chunk` parameter of `readable.push()` is `undefined`, it will be treated as empty string or buffer. See `readable.push('')` for more information.

Errors while reading

Errors occurring during processing of the `readable._read()` must be propagated through the `readable.destroy(err)` method. Throwing an `Error` from within `readable._read()` or manually emitting an `'error'` event results in undefined behavior.

```

const { Readable } = require('stream');

const myReadable = new Readable({
  read(size) {
    const err = checkSomeErrorCondition();
    if (err) {
      this.destroy(err);
    } else {
      // Do some work.
    }
  }
});

```

An example counting stream

The following is a basic example of a `Readable` stream that emits the numerals from 1 to 1,000,000 in ascending order, and then ends.

```

const { Readable } = require('stream');

class Counter extends Readable {
  constructor(opt) {
    super(opt);
    this._max = 1000000;
    this._index = 1;
  }

  _read() {
    const i = this._index++;
    if (i > this._max)
      this.push(null);
    else {
      const str = String(i);
      const buf = Buffer.from(str, 'ascii');
      this.push(buf);
    }
  }
}

```

```
}  
}
```

Implementing a duplex stream

A `Duplex` stream is one that implements both `Readable` and `Writable`, such as a TCP socket connection.

Because JavaScript does not have support for multiple inheritance, the `stream.Duplex` class is extended to implement a `Duplex` stream (as opposed to extending the `stream.Readable` and `stream.Writable` classes).

The `stream.Duplex` class prototypically inherits from `stream.Readable` and parasitically from `stream.Writable`, but `instanceof` will work properly for both base classes due to overriding `Symbol.hasInstance` on `stream.Writable`.

Custom `Duplex` streams *must* call the `new stream.Duplex([options])` constructor and implement *both* the `readable._read()` and `writable._write()` methods.

`new stream.Duplex(options)`

- `options` `<Object>` Passed to both `Writable` and `Readable` constructors. Also has the following fields:
 - `allowHalfOpen` `<boolean>` If set to `false`, then the stream will automatically end the writable side when the readable side ends. **Default:** `true`.
 - `readable` `<boolean>` Sets whether the `Duplex` should be readable. **Default:** `true`.
 - `writable` `<boolean>` Sets whether the `Duplex` should be writable. **Default:** `true`.
 - `readableObjectMode` `<boolean>` Sets `objectMode` for readable side of the stream. Has no effect if `objectMode` is `true`. **Default:** `false`.
 - `writableObjectMode` `<boolean>` Sets `objectMode` for writable side of the stream. Has no effect if `objectMode` is `true`. **Default:** `false`.
 - `readableHighWaterMark` `<number>` Sets `highWaterMark` for the readable side of the stream. Has no effect if `highWaterMark` is provided.
 - `writableHighWaterMark` `<number>` Sets `highWaterMark` for the writable side of the stream. Has no effect if `highWaterMark` is provided.

```
const { Duplex } = require('stream');  
  
class MyDuplex extends Duplex {  
  constructor(options) {  
    super(options);  
    // ...  
  }  
}
```

Or, when using pre-ES6 style constructors:

```
const { Duplex } = require('stream');  
const util = require('util');  
  
function MyDuplex(options) {  
  if (!(this instanceof MyDuplex))  
    return new MyDuplex(options);  
  Duplex.call(this, options);  
}
```

```
}  
util.inherits(MyDuplex, Duplex);
```

Or, using the simplified constructor approach:

```
const { Duplex } = require('stream');  
  
const myDuplex = new Duplex({  
  read(size) {  
    // ...  
  },  
  write(chunk, encoding, callback) {  
    // ...  
  }  
});
```

When using pipeline:

```
const { Transform, pipeline } = require('stream');  
const fs = require('fs');  
  
pipeline(  
  fs.createReadStream('object.json')  
    .setEncoding('utf8'),  
  new Transform({  
    decodeStrings: false, // Accept string input rather than Buffers  
    construct(callback) {  
      this.data = '';  
      callback();  
    },  
    transform(chunk, encoding, callback) {  
      this.data += chunk;  
      callback();  
    },  
    flush(callback) {  
      try {  
        // Make sure is valid json.  
        JSON.parse(this.data);  
        this.push(this.data);  
      } catch (err) {  
        callback(err);  
      }  
    }  
  })  
),  
  fs.createWriteStream('valid-object.json'),  
(err) => {  
  if (err) {  
    console.error('failed', err);  
  } else {  
    console.log('completed');
```

```
    }  
  }  
);
```

An example duplex stream

The following illustrates a simple example of a `Duplex` stream that wraps a hypothetical lower-level source object to which data can be written, and from which data can be read, albeit using an API that is not compatible with Node.js streams. The following illustrates a simple example of a `Duplex` stream that buffers incoming written data via the `Writable` interface that is read back out via the `Readable` interface.

```
const { Duplex } = require('stream');  
const kSource = Symbol('source');  
  
class MyDuplex extends Duplex {  
  constructor(source, options) {  
    super(options);  
    this[kSource] = source;  
  }  
  
  _write(chunk, encoding, callback) {  
    // The underlying source only deals with strings.  
    if (Buffer.isBuffer(chunk))  
      chunk = chunk.toString();  
    this[kSource].writeSomeData(chunk);  
    callback();  
  }  
  
  _read(size) {  
    this[kSource].fetchSomeData(size, (data, encoding) => {  
      this.push(Buffer.from(data, encoding));  
    });  
  }  
}
```

The most important aspect of a `Duplex` stream is that the `Readable` and `Writable` sides operate independently of one another despite co-existing within a single object instance.

Object mode duplex streams

For `Duplex` streams, `objectMode` can be set exclusively for either the `Readable` or `Writable` side using the `readableObjectMode` and `writableObjectMode` options respectively.

In the following example, for instance, a new `Transform` stream (which is a type of `Duplex` stream) is created that has an object mode `Writable` side that accepts JavaScript numbers that are converted to hexadecimal strings on the `Readable` side.

```
const { Transform } = require('stream');  
  
// All Transform streams are also Duplex Streams.  
const myTransform = new Transform({  
  writableObjectMode: true,
```

```

transform(chunk, encoding, callback) {
  // Coerce the chunk to a number if necessary.
  chunk |= 0;

  // Transform the chunk into something else.
  const data = chunk.toString(16);

  // Push the data onto the readable queue.
  callback(null, '0'.repeat(data.length % 2) + data);
}
});

myTransform.setEncoding('ascii');
myTransform.on('data', (chunk) => console.log(chunk));

myTransform.write(1);
// Prints: 01
myTransform.write(10);
// Prints: 0a
myTransform.write(100);
// Prints: 64

```

Implementing a transform stream

A **Transform** stream is a **Duplex** stream where the output is computed in some way from the input. Examples include **zlib** streams or **crypto** streams that compress, encrypt, or decrypt data.

There is no requirement that the output be the same size as the input, the same number of chunks, or arrive at the same time. For example, a **Hash** stream will only ever have a single chunk of output which is provided when the input is ended. A **zlib** stream will produce output that is either much smaller or much larger than its input.

The **stream.Transform** class is extended to implement a **Transform** stream.

The **stream.Transform** class prototypically inherits from **stream.Duplex** and implements its own versions of the **writable._write()** and **readable._read()** methods. Custom **Transform** implementations *must* implement the **transform._transform()** method and *may* also implement the **transform._flush()** method.

Care must be taken when using **Transform** streams in that data written to the stream can cause the **Writable** side of the stream to become paused if the output on the **Readable** side is not consumed.

new stream.Transform([options])

- **options** **<Object>** Passed to both **Writable** and **Readable** constructors. Also has the following fields:
 - **transform** **<Function>** Implementation for the **stream._transform()** method.
 - **flush** **<Function>** Implementation for the **stream._flush()** method.

```

const { Transform } = require('stream');

class MyTransform extends Transform {
  constructor(options) {
    super(options);
    // ...
  }
}

```

```
}  
}
```

Or, when using pre-ES6 style constructors:

```
const { Transform } = require('stream');  
const util = require('util');  
  
function MyTransform(options) {  
  if (!(this instanceof MyTransform))  
    return new MyTransform(options);  
  Transform.call(this, options);  
}  
util.inherits(MyTransform, Transform);
```

Or, using the simplified constructor approach:

```
const { Transform } = require('stream');  
  
const myTransform = new Transform({  
  transform(chunk, encoding, callback) {  
    // ...  
  }  
});
```

Event: 'end'

The 'end' event is from the `stream.Readable` class. The 'end' event is emitted after all data has been output, which occurs after the callback in `transform._flush()` has been called. In the case of an error, 'end' should not be emitted.

Event: 'finish'

The 'finish' event is from the `stream.Writable` class. The 'finish' event is emitted after `stream.end()` is called and all chunks have been processed by `stream._transform()`. In the case of an error, 'finish' should not be emitted.

transform._flush(callback)

- `callback` `<Function>` A callback function (optionally with an error argument and data) to be called when remaining data has been flushed.

This function MUST NOT be called by application code directly. It should be implemented by child classes, and called by the internal `Readable` class methods only.

In some cases, a transform operation may need to emit an additional bit of data at the end of the stream. For example, a `zlib` compression stream will store an amount of internal state used to optimally compress the output. When the stream ends, however, that additional data needs to be flushed so that the compressed data will be complete.

Custom `Transform` implementations *may* implement the `transform._flush()` method. This will be called when there is no more written data to be consumed, but before the 'end' event is emitted signaling the end of the `Readable` stream.

Within the `transform._flush()` implementation, the `transform.push()` method may be called zero or more times, as appropriate. The `callback` function must be called when the flush operation is complete.

The `transform._flush()` method is prefixed with an underscore because it is internal to the class that defines it, and should never be called directly by user programs.

transform._transform(chunk, encoding, callback)

- `chunk` `<Buffer> | <string> | <any>` The `Buffer` to be transformed, converted from the `string` passed to `stream.write()`. If the stream's `decodeStrings` option is `false` or the stream is operating in object mode, the chunk will not be converted & will be whatever was passed to `stream.write()`.
- `encoding` `<string>` If the chunk is a string, then this is the encoding type. If chunk is a buffer, then this is the special value `'buffer'`. Ignore it in that case.
- `callback` `<Function>` A callback function (optionally with an error argument and data) to be called after the supplied `chunk` has been processed.

This function MUST NOT be called by application code directly. It should be implemented by child classes, and called by the internal `Readable` class methods only.

All `Transform` stream implementations must provide a `_transform()` method to accept input and produce output. The `transform._transform()` implementation handles the bytes being written, computes an output, then passes that output off to the readable portion using the `transform.push()` method.

The `transform.push()` method may be called zero or more times to generate output from a single input chunk, depending on how much is to be output as a result of the chunk.

It is possible that no output is generated from any given chunk of input data.

The `callback` function must be called only when the current chunk is completely consumed. The first argument passed to the `callback` must be an `Error` object if an error occurred while processing the input or `null` otherwise. If a second argument is passed to the `callback`, it will be forwarded on to the `transform.push()` method. In other words, the following are equivalent:

```
transform.prototype._transform = function(data, encoding, callback) {
  this.push(data);
  callback();
};

transform.prototype._transform = function(data, encoding, callback) {
  callback(null, data);
};
```

The `transform._transform()` method is prefixed with an underscore because it is internal to the class that defines it, and should never be called directly by user programs.

`transform._transform()` is never called in parallel; streams implement a queue mechanism, and to receive the next chunk, `callback` must be called, either synchronously or asynchronously.

Class: stream.PassThrough

The `stream.PassThrough` class is a trivial implementation of a `Transform` stream that simply passes the input bytes across to the output. Its purpose is primarily for examples and testing, but there are some use cases where `stream.PassThrough` is useful as a building block for novel sorts of streams.

Additional notes

Streams compatibility with async generators and async iterators

With the support of async generators and iterators in JavaScript, async generators are effectively a first-class language-level stream construct at this point.

Some common interop cases of using Node.js streams with async generators and async iterators are provided below.

Consuming readable streams with async iterators

```
(async function() {  
  for await (const chunk of readable) {  
    console.log(chunk);  
  }  
})();
```

Async iterators register a permanent error handler on the stream to prevent any unhandled post-destroy errors.

Creating readable streams with async generators

A Node.js readable stream can be created from an asynchronous generator using the `Readable.from()` utility method:

```
const { Readable } = require('stream');  
  
async function * generate() {  
  yield 'a';  
  yield 'b';  
  yield 'c';  
}  
  
const readable = Readable.from(generate());  
  
readable.on('data', (chunk) => {  
  console.log(chunk);  
});
```

Piping to writable streams from async iterators

When writing to a writable stream from an async iterator, ensure correct handling of backpressure and errors. `stream.pipeline()` abstracts away the handling of backpressure and backpressure-related errors:

```
const fs = require('fs');  
const { pipeline } = require('stream');  
const { pipeline: pipelinePromise } = require('stream/promises');  
  
const writable = fs.createWriteStream('./file');  
  
// Callback Pattern  
pipeline(iterator, writable, (err, value) => {  
  if (err) {  
    console.error(err);  
  } else {  
    console.log(value, 'value returned');  
  }  
})
```



```
});

// Promise Pattern
pipelinePromise(iterator, writable)
  .then((value) => {
    console.log(value, 'value returned');
  })
  .catch(console.error);
```

Compatibility with older Node.js versions

Prior to Node.js 0.10, the `Readable` stream interface was simpler, but also less powerful and less useful.

- Rather than waiting for calls to the `stream.read()` method, `'data'` events would begin emitting immediately. Applications that would need to perform some amount of work to decide how to handle data were required to store read data into buffers so the data would not be lost.
- The `stream.pause()` method was advisory, rather than guaranteed. This meant that it was still necessary to be prepared to receive `'data'` events *even when the stream was in a paused state*.

In Node.js 0.10, the `Readable` class was added. For backward compatibility with older Node.js programs, `Readable` streams switch into "flowing mode" when a `'data'` event handler is added, or when the `stream.resume()` method is called. The effect is that, even when not using the new `stream.read()` method and `'readable'` event, it is no longer necessary to worry about losing `'data'` chunks.

While most applications will continue to function normally, this introduces an edge case in the following conditions:

- No `'data'` event listener is added.
- The `stream.resume()` method is never called.
- The stream is not piped to any writable destination.

For example, consider the following code:

```
// WARNING!  BROKEN!
net.createServer((socket) => {

  // We add an 'end' listener, but never consume the data.
  socket.on('end', () => {
    // It will never get here.
    socket.end('The message was received but was not processed.\n');
  });

}).listen(1337);
```

Prior to Node.js 0.10, the incoming message data would be simply discarded. However, in Node.js 0.10 and beyond, the socket remains paused forever.

The workaround in this situation is to call the `stream.resume()` method to begin the flow of data:

```
// Workaround.
net.createServer((socket) => {
  socket.on('end', () => {
    socket.resume();
    socket.end('The message was received but was not processed.\n');
  });
});
```

```
// Start the flow of data, discarding it.
socket.resume();
}).listen(1337);
```

In addition to new `Readable` streams switching into flowing mode, pre-0.10 style streams can be wrapped in a `Readable` class using the `readable.wrap()` method.

readable.read(0)

There are some cases where it is necessary to trigger a refresh of the underlying readable stream mechanisms, without actually consuming any data. In such cases, it is possible to call `readable.read(0)`, which will always return `null`.

If the internal read buffer is below the `highWaterMark`, and the stream is not currently reading, then calling `stream.read(0)` will trigger a low-level `stream._read()` call.

While most applications will almost never need to do this, there are situations within Node.js where this is done, particularly in the `Readable` stream class internals.

readable.push("")

Use of `readable.push('')` is not recommended.

Pushing a zero-byte string, `Buffer` or `Uint8Array` to a stream that is not in object mode has an interesting side effect. Because it is a call to `readable.push()`, the call will end the reading process. However, because the argument is an empty string, no data is added to the readable buffer so there is nothing for a user to consume.

highWaterMark discrepancy after calling readable.setEncoding()

The use of `readable.setEncoding()` will change the behavior of how the `highWaterMark` operates in non-object mode.

Typically, the size of the current buffer is measured against the `highWaterMark` in *bytes*. However, after `setEncoding()` is called, the comparison function will begin to measure the buffer's size in *characters*.

This is not a problem in common cases with `latin1` or `ascii`. But it is advised to be mindful about this behavior when working with strings that could contain multi-byte characters.

String decoder

Stability: 2 - Stable

Source Code: [lib/string_decoder.js](#)

The `string_decoder` module provides an API for decoding `Buffer` objects into strings in a manner that preserves encoded multi-byte UTF-8 and UTF-16 characters. It can be accessed using:

```
const { StringDecoder } = require('string_decoder');
```

The following example shows the basic use of the `StringDecoder` class.

```
const { StringDecoder } = require('string_decoder');
const decoder = new StringDecoder('utf8');
```

```
const cent = Buffer.from([0xC2, 0xA2]);
console.log(decoder.write(cent));

const euro = Buffer.from([0xE2, 0x82, 0xAC]);
console.log(decoder.write(euro));
```

When a `Buffer` instance is written to the `StringDecoder` instance, an internal buffer is used to ensure that the decoded string does not contain any incomplete multibyte characters. These are held in the buffer until the next call to `stringDecoder.write()` or until `stringDecoder.end()` is called.

In the following example, the three UTF-8 encoded bytes of the European Euro symbol (€) are written over three separate operations:

```
const { StringDecoder } = require('string_decoder');
const decoder = new StringDecoder('utf8');

decoder.write(Buffer.from([0xE2]));
decoder.write(Buffer.from([0x82]));
console.log(decoder.end(Buffer.from([0xAC])));
```

Class: StringDecoder

new StringDecoder([encoding])

- `encoding` `<string>` The character `encoding` the `StringDecoder` will use. **Default:** `'utf8'`.

Creates a new `StringDecoder` instance.

stringDecoder.end([buffer])

- `buffer` `<Buffer>` | `<TypedArray>` | `<DataView>` A `Buffer`, or `TypedArray`, or `DataView` containing the bytes to decode.
- Returns: `<string>`

Returns any remaining input stored in the internal buffer as a string. Bytes representing incomplete UTF-8 and UTF-16 characters will be replaced with substitution characters appropriate for the character encoding.

If the `buffer` argument is provided, one final call to `stringDecoder.write()` is performed before returning the remaining input. After `end()` is called, the `stringDecoder` object can be reused for new input.

stringDecoder.write(buffer)

- `buffer` `<Buffer>` | `<TypedArray>` | `<DataView>` A `Buffer`, or `TypedArray`, or `DataView` containing the bytes to decode.
- Returns: `<string>`

Returns a decoded string, ensuring that any incomplete multibyte characters at the end of the `Buffer`, or `TypedArray`, or `DataView` are omitted from the returned string and stored in an internal buffer for the next call to `stringDecoder.write()` or `stringDecoder.end()`.

Timers

Stability: 2 - Stable

Source Code: [lib/timers.js](#)

The `timer` module exposes a global API for scheduling functions to be called at some future period of time. Because the timer functions are globals, there is no need to call `require('timers')` to use the API.

The timer functions within Node.js implement a similar API as the timers API provided by Web Browsers but use a different internal implementation that is built around the Node.js `Event Loop`.

Class: Immediate

This object is created internally and is returned from `setImmediate()`. It can be passed to `clearImmediate()` in order to cancel the scheduled actions.

By default, when an immediate is scheduled, the Node.js event loop will continue running as long as the immediate is active. The `Immediate` object returned by `setImmediate()` exports both `immediate.ref()` and `immediate.unref()` functions that can be used to control this default behavior.

immediate.hasRef()

- Returns: `<boolean>`

If true, the `Immediate` object will keep the Node.js event loop active.

immediate.ref()

- Returns: `<Immediate>` a reference to `immediate`

When called, requests that the Node.js event loop *not* exit so long as the `Immediate` is active. Calling `immediate.ref()` multiple times will have no effect.

By default, all `Immediate` objects are "ref'ed", making it normally unnecessary to call `immediate.ref()` unless `immediate.unref()` had been called previously.

immediate.unref()

- Returns: `<Immediate>` a reference to `immediate`

When called, the active `Immediate` object will not require the Node.js event loop to remain active. If there is no other activity keeping the event loop running, the process may exit before the `Immediate` object's callback is invoked. Calling `immediate.unref()` multiple times will have no effect.

Class: Timeout

This object is created internally and is returned from `setTimeout()` and `setInterval()`. It can be passed to either `clearTimeout()` or `clearInterval()` in order to cancel the scheduled actions.

By default, when a timer is scheduled using either `setTimeout()` or `setInterval()`, the Node.js event loop will continue running as long as the timer is active. Each of the `Timeout` objects returned by these functions export both `timeout.ref()` and `timeout.unref()` functions that can be used to control this default behavior.

timeout.hasRef()

- Returns: `<boolean>`

If true, the `Timeout` object will keep the Node.js event loop active.

timeout.ref()

- Returns: `<Timeout>` a reference to `timeout`

When called, requests that the Node.js event loop *not* exit so long as the `Timeout` is active. Calling `timeout.ref()` multiple times will have no effect.

By default, all `Timeout` objects are "ref'ed", making it normally unnecessary to call `timeout.ref()` unless `timeout.unref()` had been called previously.

`timeout.refresh()`

- Returns: `<Timeout>` a reference to `timeout`

Sets the timer's start time to the current time, and reschedules the timer to call its callback at the previously specified duration adjusted to the current time. This is useful for refreshing a timer without allocating a new JavaScript object.

Using this on a timer that has already called its callback will reactivate the timer.

`timeout.unref()`

- Returns: `<Timeout>` a reference to `timeout`

When called, the active `Timeout` object will not require the Node.js event loop to remain active. If there is no other activity keeping the event loop running, the process may exit before the `Timeout` object's callback is invoked. Calling `timeout.unref()` multiple times will have no effect.

Calling `timeout.unref()` creates an internal timer that will wake the Node.js event loop. Creating too many of these can adversely impact performance of the Node.js application.

`timeout[Symbol.toPrimitive]()`

- Returns: `<integer>` a number that can be used to reference this `timeout`

Coerce a `Timeout` to a primitive. The primitive can be used to clear the `Timeout`. The primitive can only be used in the same thread where the timeout was created. Therefore, to use it across `worker_threads` it must first be passed to the correct thread. This allows enhanced compatibility with browser `setTimeout()` and `setInterval()` implementations.

Scheduling timers

A timer in Node.js is an internal construct that calls a given function after a certain period of time. When a timer's function is called varies depending on which method was used to create the timer and what other work the Node.js event loop is doing.

`setImmediate(callback[, ...args])`

- `callback` `<Function>` The function to call at the end of this turn of the Node.js `Event Loop`
- `...args` `<any>` Optional arguments to pass when the `callback` is called.
- Returns: `<Immediate>` for use with `clearImmediate()`

Schedules the "immediate" execution of the `callback` after I/O events' callbacks.

When multiple calls to `setImmediate()` are made, the `callback` functions are queued for execution in the order in which they are created. The entire callback queue is processed every event loop iteration. If an immediate timer is queued from inside an executing callback, that timer will not be triggered until the next event loop iteration.

If `callback` is not a function, a `TypeError` will be thrown.

This method has a custom variant for promises that is available using `timersPromises.setImmediate()`.

`setInterval(callback[, delay[, ...args]])`

- `callback` `<Function>` The function to call when the timer elapses.
- `delay` `<number>` The number of milliseconds to wait before calling the `callback`. **Default: 1**.
- `...args` `<any>` Optional arguments to pass when the `callback` is called.
- Returns: `<Timeout>` for use with `clearInterval()`

Schedules repeated execution of `callback` every `delay` milliseconds.

When `delay` is larger than `2147483647` or less than `1`, the `delay` will be set to `1`. Non-integer delays are truncated to an integer.

If `callback` is not a function, a `TypeError` will be thrown.

This method has a custom variant for promises that is available using `timersPromises.setInterval()`.

setTimeout(callback[, delay[, ...args]])

- `callback` `<Function>` The function to call when the timer elapses.
- `delay` `<number>` The number of milliseconds to wait before calling the `callback`. **Default: 1**.
- `...args` `<any>` Optional arguments to pass when the `callback` is called.
- Returns: `<Timeout>` for use with `clearTimeout()`

Schedules execution of a one-time `callback` after `delay` milliseconds.

The `callback` will likely not be invoked in precisely `delay` milliseconds. Node.js makes no guarantees about the exact timing of when callbacks will fire, nor of their ordering. The callback will be called as close as possible to the time specified.

When `delay` is larger than `2147483647` or less than `1`, the `delay` will be set to `1`. Non-integer delays are truncated to an integer.

If `callback` is not a function, a `TypeError` will be thrown.

This method has a custom variant for promises that is available using `timersPromises.setTimeout()`.

Cancelling timers

The `setImmediate()`, `setInterval()`, and `setTimeout()` methods each return objects that represent the scheduled timers. These can be used to cancel the timer and prevent it from triggering.

For the promisified variants of `setImmediate()` and `setTimeout()`, an `AbortController` may be used to cancel the timer. When canceled, the returned Promises will be rejected with an `'AbortError'`.

For `setImmediate()`:

```
const { setImmediate: setImmediatePromise } = require('timers/promises');

const ac = new AbortController();
const signal = ac.signal;

setImmediatePromise('foobar', { signal })
  .then(console.log)
  .catch((err) => {
    if (err.name === 'AbortError')
      console.log('The immediate was aborted');
  });
```

```
ac.abort();
```

For `setTimeout()`:

```
const { setTimeout: setTimeoutPromise } = require('timers/promises');

const ac = new AbortController();
const signal = ac.signal;

setTimeoutPromise(1000, 'foobar', { signal })
  .then(console.log)
  .catch((err) => {
    if (err.name === 'AbortError')
      console.log('The timeout was aborted');
  });

ac.abort();
```

clearImmediate(immediate)

- `immediate` `<Immediate>` An `Immediate` object as returned by `setImmediate()`.

Cancels an `Immediate` object created by `setImmediate()`.

clearInterval(timeout)

- `timeout` `<Timeout>` | `<string>` | `<number>` A `Timeout` object as returned by `setInterval()` or the `primitive` of the `Timeout` object as a string or a number.

Cancels a `Timeout` object created by `setInterval()`.

clearTimeout(timeout)

- `timeout` `<Timeout>` | `<string>` | `<number>` A `Timeout` object as returned by `setTimeout()` or the `primitive` of the `Timeout` object as a string or a number.

Cancels a `Timeout` object created by `setTimeout()`.

Timers Promises API

The `timers/promises` API provides an alternative set of timer functions that return `Promise` objects. The API is accessible via `require('timers/promises')`.

```
import {
  setTimeout,
  setImmediate,
  setInterval,
} from 'timers/promises'; const {
  setTimeout,
  setImmediate,
  setInterval,
} = require('timers/promises');
```

timersPromises.setTimeout([delay[, value[, options]])

- `delay` `<number>` The number of milliseconds to wait before fulfilling the promise. **Default:** `1`.
- `value` `<any>` A value with which the promise is fulfilled.
- `options` `<Object>`
 - `ref` `<boolean>` Set to `false` to indicate that the scheduled `Timeout` should not require the Node.js event loop to remain active. **Default:** `true`.
 - `signal` `<AbortSignal>` An optional `AbortSignal` that can be used to cancel the scheduled `Timeout`.

```
import {
  setTimeout,
} from 'timers/promises';

const res = await setTimeout(100, 'result');

console.log(res); // Prints 'result'const {
  setTimeout,
} = require('timers/promises');

setTimeout(100, 'result').then((res) => {
  console.log(res); // Prints 'result'
});
```

timersPromises.setImmediate([value[, options]])

- `value` `<any>` A value with which the promise is fulfilled.
- `options` `<Object>`
 - `ref` `<boolean>` Set to `false` to indicate that the scheduled `Immediate` should not require the Node.js event loop to remain active. **Default:** `true`.
 - `signal` `<AbortSignal>` An optional `AbortSignal` that can be used to cancel the scheduled `Immediate`.

```
import {
  setImmediate,
} from 'timers/promises';

const res = await setImmediate('result');

console.log(res); // Prints 'result'const {
  setImmediate,
} = require('timers/promises');

setImmediate('result').then((res) => {
  console.log(res); // Prints 'result'
});
```

timersPromises.setInterval([delay[, value[, options]])

Returns an async iterator that generates values in an interval of `delay` ms.

- `delay` `<number>` The number of milliseconds to wait between iterations. **Default:** `1`.
- `value` `<any>` A value with which the iterator returns.
- `options` `<Object>`
 - `ref` `<boolean>` Set to `false` to indicate that the scheduled `Timeout` between iterations should not require the Node.js event loop to remain active. **Default:** `true`.
 - `signal` `<AbortSignal>` An optional `AbortSignal` that can be used to cancel the scheduled `Timeout` between operations.

```
import {
  setInterval,
} from 'timers/promises';

const interval = 100;
for await (const startTime of setInterval(interval, Date.now())) {
  const now = Date.now();
  console.log(now);
  if ((now - startTime) > 1000)
    break;
}
console.log(Date.now());const {
  setInterval,
} = require('timers/promises');
const interval = 100;

(async function() {
  for await (const startTime of setInterval(interval, Date.now())) {
    const now = Date.now();
    console.log(now);
    if ((now - startTime) > 1000)
      break;
  }
  console.log(Date.now());
})();
```

TLS (SSL)

Stability: 2 - Stable

Source Code: [lib/tls.js](#)

The `tls` module provides an implementation of the Transport Layer Security (TLS) and Secure Socket Layer (SSL) protocols that is built on top of OpenSSL. The module can be accessed using:

```
const tls = require('tls');
```

TLS/SSL concepts

The TLS/SSL is a public/private key infrastructure (PKI). For most common cases, each client and server must have a *private key*.

Private keys can be generated in multiple ways. The example below illustrates use of the OpenSSL command-line interface to generate a 2048-bit RSA private key:

```
openssl genrsa -out ryans-key.pem 2048
```

With TLS/SSL, all servers (and some clients) must have a *certificate*. Certificates are *public* keys that correspond to a private key, and that are digitally signed either by a Certificate Authority or by the owner of the private key (such certificates are referred to as "self-signed"). The first step to obtaining a certificate is to create a *Certificate Signing Request* (CSR) file.

The OpenSSL command-line interface can be used to generate a CSR for a private key:

```
openssl req -new -sha256 -key ryans-key.pem -out ryans-csr.pem
```

Once the CSR file is generated, it can either be sent to a Certificate Authority for signing or used to generate a self-signed certificate.

Creating a self-signed certificate using the OpenSSL command-line interface is illustrated in the example below:

```
openssl x509 -req -in ryans-csr.pem -signkey ryans-key.pem -out ryans-cert.pem
```

Once the certificate is generated, it can be used to generate a `.pfx` or `.p12` file:

```
openssl pkcs12 -export -in ryans-cert.pem -inkey ryans-key.pem \
    -certfile ca-cert.pem -out ryans.pfx
```

Where:

- `in` : is the signed certificate
- `inkey` : is the associated private key
- `certfile` : is a concatenation of all Certificate Authority (CA) certs into a single file, e.g. `cat ca1-cert.pem ca2-cert.pem > ca-cert.pem`

Perfect forward secrecy

The term *forward secrecy* or *perfect forward secrecy* describes a feature of key-agreement (i.e., key-exchange) methods. That is, the server and client keys are used to negotiate new temporary keys that are used specifically and only for the current communication session. Practically, this means that even if the server's private key is compromised, communication can only be decrypted by eavesdroppers if the attacker manages to obtain the key-pair specifically generated for the session.

Perfect forward secrecy is achieved by randomly generating a key pair for key-agreement on every TLS/SSL handshake (in contrast to using the same key for all sessions). Methods implementing this technique are called "ephemeral".

Currently two methods are commonly used to achieve perfect forward secrecy (note the character "E" appended to the traditional abbreviations):

- **DHE** : An ephemeral version of the Diffie-Hellman key-agreement protocol.
- **ECDHE** : An ephemeral version of the Elliptic Curve Diffie-Hellman key-agreement protocol.

Ephemeral methods may have some performance drawbacks, because key generation is expensive.

To use perfect forward secrecy using **DHE** with the `tls` module, it is required to generate Diffie-Hellman parameters and specify them with the `dhparam` option to `tls.createSecureContext()`. The following illustrates the use of the OpenSSL command-line interface to generate such parameters:

```
openssl dhparam -outform PEM -out dhparam.pem 2048
```

If using perfect forward secrecy using `ECDHE`, Diffie-Hellman parameters are not required and a default ECDHE curve will be used. The `ecdhCurve` property can be used when creating a TLS Server to specify the list of names of supported curves to use, see `tls.createServer()` for more info.

Perfect forward secrecy was optional up to TLSv1.2, but it is not optional for TLSv1.3, because all TLSv1.3 cipher suites use ECDHE.

ALPN and SNI

ALPN (Application-Layer Protocol Negotiation Extension) and SNI (Server Name Indication) are TLS handshake extensions:

- ALPN: Allows the use of one TLS server for multiple protocols (HTTP, HTTP/2)
- SNI: Allows the use of one TLS server for multiple hostnames with different SSL certificates.

Pre-shared keys

TLS-PSK support is available as an alternative to normal certificate-based authentication. It uses a pre-shared key instead of certificates to authenticate a TLS connection, providing mutual authentication. TLS-PSK and public key infrastructure are not mutually exclusive. Clients and servers can accommodate both, choosing either of them during the normal cipher negotiation step.

TLS-PSK is only a good choice where means exist to securely share a key with every connecting machine, so it does not replace PKI (Public Key Infrastructure) for the majority of TLS uses. The TLS-PSK implementation in OpenSSL has seen many security flaws in recent years, mostly because it is used only by a minority of applications. Please consider all alternative solutions before switching to PSK ciphers. Upon generating PSK it is of critical importance to use sufficient entropy as discussed in [RFC 4086](#). Deriving a shared secret from a password or other low-entropy sources is not secure.

PSK ciphers are disabled by default, and using TLS-PSK thus requires explicitly specifying a cipher suite with the `ciphers` option. The list of available ciphers can be retrieved via `openssl ciphers -v 'PSK'`. All TLS 1.3 ciphers are eligible for PSK but currently only those that use SHA256 digest are supported they can be retrieved via `openssl ciphers -v -s -tls1_3 -psk`.

According to the [RFC 4279](#), PSK identities up to 128 bytes in length and PSKs up to 64 bytes in length must be supported. As of OpenSSL 1.1.0 maximum identity size is 128 bytes, and maximum PSK length is 256 bytes.

The current implementation doesn't support asynchronous PSK callbacks due to the limitations of the underlying OpenSSL API.

Client-initiated renegotiation attack mitigation

The TLS protocol allows clients to renegotiate certain aspects of the TLS session. Unfortunately, session renegotiation requires a disproportionate amount of server-side resources, making it a potential vector for denial-of-service attacks.

To mitigate the risk, renegotiation is limited to three times every ten minutes. An `'error'` event is emitted on the `tls.TLSSocket` instance when this threshold is exceeded. The limits are configurable:

- `tls.CLIENT_RENEG_LIMIT` `<number>` Specifies the number of renegotiation requests. **Default:** `3`.
- `tls.CLIENT_RENEG_WINDOW` `<number>` Specifies the time renegotiation window in seconds. **Default:** `600` (10 minutes).

The default renegotiation limits should not be modified without a full understanding of the implications and risks.

TLSv1.3 does not support renegotiation.

Session resumption

Establishing a TLS session can be relatively slow. The process can be sped up by saving and later reusing the session state. There are several mechanisms to do so, discussed here from oldest to newest (and preferred).

Session identifiers

Servers generate a unique ID for new connections and send it to the client. Clients and servers save the session state. When reconnecting, clients send the ID of their saved session state and if the server also has the state for that ID, it can agree to use it. Otherwise, the server will create a new session. See [RFC 2246](#) for more information, page 23 and 30.

Resumption using session identifiers is supported by most web browsers when making HTTPS requests.

For Node.js, clients wait for the `'session'` event to get the session data, and provide the data to the `session` option of a subsequent `tls.connect()` to reuse the session. Servers must implement handlers for the `'newSession'` and `'resumeSession'` events to save and restore the session data using the session ID as the lookup key to reuse sessions. To reuse sessions across load balancers or cluster workers, servers must use a shared session cache (such as Redis) in their session handlers.

Session tickets

The servers encrypt the entire session state and send it to the client as a "ticket". When reconnecting, the state is sent to the server in the initial connection. This mechanism avoids the need for server-side session cache. If the server doesn't use the ticket, for any reason (failure to decrypt it, it's too old, etc.), it will create a new session and send a new ticket. See [RFC 5077](#) for more information.

Resumption using session tickets is becoming commonly supported by many web browsers when making HTTPS requests.

For Node.js, clients use the same APIs for resumption with session identifiers as for resumption with session tickets. For debugging, if `tls.TLSSocket.getTLSTicket()` returns a value, the session data contains a ticket, otherwise it contains client-side session state.

With TLSv1.3, be aware that multiple tickets may be sent by the server, resulting in multiple `'session'` events, see `'session'` for more information.

Single process servers need no specific implementation to use session tickets. To use session tickets across server restarts or load balancers, servers must all have the same ticket keys. There are three 16-byte keys internally, but the `tls` API exposes them as a single 48-byte buffer for convenience.

It's possible to get the ticket keys by calling `server.getTicketKeys()` on one server instance and then distribute them, but it is more reasonable to securely generate 48 bytes of secure random data and set them with the `ticketKeys` option of `tls.createServer()`. The keys should be regularly regenerated and server's keys can be reset with `server.setTicketKeys()`.

Session ticket keys are cryptographic keys, and they **must be stored securely**. With TLS 1.2 and below, if they are compromised all sessions that used tickets encrypted with them can be decrypted. They should not be stored on disk, and they should be regenerated regularly.

If clients advertise support for tickets, the server will send them. The server can disable tickets by supplying `require('constants').SSL_OP_NO_TICKET` in `secureOptions`.

Both session identifiers and session tickets timeout, causing the server to create new sessions. The timeout can be configured with the `sessionTimeout` option of `tls.createServer()`.

For all the mechanisms, when resumption fails, servers will create new sessions. Since failing to resume the session does not cause TLS/HTTPS connection failures, it is easy to not notice unnecessarily poor TLS performance. The OpenSSL CLI can be used to verify that servers are resuming sessions. Use the `-reconnect` option to `openssl s_client`, for example:

```
$ openssl s_client -connect localhost:443 -reconnect
```

Read through the debug output. The first connection should say "New", for example:

```
New, TLSv1.2, Cipher is ECDHE-RSA-AES128-GCM-SHA256
```

Subsequent connections should say "Reused", for example:

```
Reused, TLSv1.2, Cipher is ECDHE-RSA-AES128-GCM-SHA256
```

Modifying the default TLS cipher suite

Node.js is built with a default suite of enabled and disabled TLS ciphers. This default cipher list can be configured when building Node.js to allow distributions to provide their own default list.

The following command can be used to show the default cipher suite:

```
node -p crypto.constants.defaultCoreCipherList | tr ':' '\n'
TLS_AES_256_GCM_SHA384
TLS_CHACHA20_POLY1305_SHA256
TLS_AES_128_GCM_SHA256
ECDHE-RSA-AES128-GCM-SHA256
ECDHE-ECDSA-AES128-GCM-SHA256
ECDHE-RSA-AES256-GCM-SHA384
ECDHE-ECDSA-AES256-GCM-SHA384
DHE-RSA-AES128-GCM-SHA256
ECDHE-RSA-AES128-SHA256
DHE-RSA-AES128-SHA256
ECDHE-RSA-AES256-SHA384
DHE-RSA-AES256-SHA384
ECDHE-RSA-AES256-SHA256
DHE-RSA-AES256-SHA256
HIGH
!aNULL
!eNULL
!EXPORT
!DES
!RC4
!MD5
!PSK
!SRP
!CAMELLIA
```

This default can be replaced entirely using the `--tls-cipher-list` command-line switch (directly, or via the `NODE_OPTIONS` environment variable). For instance, the following makes `ECDHE-RSA-AES128-GCM-SHA256: !RC4` the default TLS cipher suite:

```
node --tls-cipher-list='ECDHE-RSA-AES128-GCM-SHA256: !RC4' server.js

export NODE_OPTIONS=--tls-cipher-list='ECDHE-RSA-AES128-GCM-SHA256: !RC4'
node server.js
```

The default can also be replaced on a per client or server basis using the `ciphers` option from `tls.createSecureContext()`, which is also available in `tls.createServer()`, `tls.connect()`, and when creating new `tls.TLSSocket`s.

The ciphers list can contain a mixture of TLSv1.3 cipher suite names, the ones that start with `'TLS_'`, and specifications for TLSv1.2 and below cipher suites. The TLSv1.2 ciphers support a legacy specification format, consult the OpenSSL [cipher list format](#) documentation for

details, but those specifications do *not* apply to TLSv1.3 ciphers. The TLSv1.3 suites can only be enabled by including their full name in the cipher list. They cannot, for example, be enabled or disabled by using the legacy TLSv1.2 `'EECDH'` or `'!EECDH'` specification.

Despite the relative order of TLSv1.3 and TLSv1.2 cipher suites, the TLSv1.3 protocol is significantly more secure than TLSv1.2, and will always be chosen over TLSv1.2 if the handshake indicates it is supported, and if any TLSv1.3 cipher suites are enabled.

The default cipher suite included within Node.js has been carefully selected to reflect current security best practices and risk mitigation. Changing the default cipher suite can have a significant impact on the security of an application. The `-tls-cipher-list` switch and `ciphers` option should be used only if absolutely necessary.

The default cipher suite prefers GCM ciphers for [Chrome's 'modern cryptography' setting](#) and also prefers ECDHE and DHE ciphers for perfect forward secrecy, while offering *some* backward compatibility.

128 bit AES is preferred over 192 and 256 bit AES in light of [specific attacks affecting larger AES key sizes](#).

Old clients that rely on insecure and deprecated RC4 or DES-based ciphers (like Internet Explorer 6) cannot complete the handshaking process with the default configuration. If these clients *must* be supported, the [TLS recommendations](#) may offer a compatible cipher suite. For more details on the format, see the OpenSSL [cipher list format](#) documentation.

There are only 5 TLSv1.3 cipher suites:

- `'TLS_AES_256_GCM_SHA384'`
- `'TLS_CHACHA20_POLY1305_SHA256'`
- `'TLS_AES_128_GCM_SHA256'`
- `'TLS_AES_128_CCM_SHA256'`
- `'TLS_AES_128_CCM_8_SHA256'`

The first 3 are enabled by default. The last 2 CCM-based suites are supported by TLSv1.3 because they may be more performant on constrained systems, but they are not enabled by default since they offer less security.

X509 Certificate Error codes

Multiple functions can fail due to certificate errors that are reported by OpenSSL. In such a case, the function provides an `<Error>` via its callback that has the property `code` which can take one of the following values:

- `'UNABLE_TO_GET_ISSUER_CERT'` : Unable to get issuer certificate.
- `'UNABLE_TO_GET_CRL'` : Unable to get certificate CRL.
- `'UNABLE_TO_DECRYPT_CERT_SIGNATURE'` : Unable to decrypt certificate's signature.
- `'UNABLE_TO_DECRYPT_CRL_SIGNATURE'` : Unable to decrypt CRL's signature.
- `'UNABLE_TO_DECODE_ISSUER_PUBLIC_KEY'` : Unable to decode issuer public key.
- `'CERT_SIGNATURE_FAILURE'` : Certificate signature failure.
- `'CRL_SIGNATURE_FAILURE'` : CRL signature failure.
- `'CERT_NOT_YET_VALID'` : Certificate is not yet valid.
- `'CERT_HAS_EXPIRED'` : Certificate has expired.
- `'CRL_NOT_YET_VALID'` : CRL is not yet valid.
- `'CRL_HAS_EXPIRED'` : CRL has expired.
- `'ERROR_IN_CERT_NOT_BEFORE_FIELD'` : Format error in certificate's notBefore field.
- `'ERROR_IN_CERT_NOT_AFTER_FIELD'` : Format error in certificate's notAfter field.
- `'ERROR_IN_CRL_LAST_UPDATE_FIELD'` : Format error in CRL's lastUpdate field.
- `'ERROR_IN_CRL_NEXT_UPDATE_FIELD'` : Format error in CRL's nextUpdate field.

- `'OUT_OF_MEM'` : Out of memory.
- `'DEPTH_ZERO_SELF_SIGNED_CERT'` : Self signed certificate.
- `'SELF_SIGNED_CERT_IN_CHAIN'` : Self signed certificate in certificate chain.
- `'UNABLE_TO_GET_ISSUER_CERT_LOCALLY'` : Unable to get local issuer certificate.
- `'UNABLE_TO_VERIFY_LEAF_SIGNATURE'` : Unable to verify the first certificate.
- `'CERT_CHAIN_TOO_LONG'` : Certificate chain too long.
- `'CERT_REVOKED'` : Certificate revoked.
- `'INVALID_CA'` : Invalid CA certificate.
- `'PATH_LENGTH_EXCEEDED'` : Path length constraint exceeded.
- `'INVALID_PURPOSE'` : Unsupported certificate purpose.
- `'CERT_UNTRUSTED'` : Certificate not trusted.
- `'CERT_REJECTED'` : Certificate rejected.
- `'HOSTNAME_MISMATCH'` : Hostname mismatch.

Class: `tls.CryptoStream`

Stability: 0 - Deprecated: Use `tls.TLSSocket` instead.

The `tls.CryptoStream` class represents a stream of encrypted data. This class is deprecated and should no longer be used.

`cryptoStream.bytesWritten`

The `cryptoStream.bytesWritten` property returns the total number of bytes written to the underlying socket *including* the bytes required for the implementation of the TLS protocol.

Class: `tls.SecurePair`

Stability: 0 - Deprecated: Use `tls.TLSSocket` instead.

Returned by `tls.createSecurePair()`.

Event: `'secure'`

The `'secure'` event is emitted by the `SecurePair` object once a secure connection has been established.

As with checking for the server `'secureConnection'` event, `pair.cleartext.authorized` should be inspected to confirm whether the certificate used is properly authorized.

Class: `tls.Server`

- Extends: `<net.Server>`

Accepts encrypted connections using TLS or SSL.

Event: `'connection'`

- `socket` `<stream.Duplex>`

This event is emitted when a new TCP stream is established, before the TLS handshake begins. `socket` is typically an object of type `net.Socket`. Usually users will not want to access this event.

This event can also be explicitly emitted by users to inject connections into the TLS server. In that case, any `Duplex` stream can be passed.

Event: 'keylog'

- `line` `<Buffer>` Line of ASCII text, in NSS `SSLKEYLOGFILE` format.
- `tlsSocket` `<tls.TLSSocket>` The `tls.TLSSocket` instance on which it was generated.

The `keylog` event is emitted when key material is generated or received by a connection to this server (typically before handshake has completed, but not necessarily). This keying material can be stored for debugging, as it allows captured TLS traffic to be decrypted. It may be emitted multiple times for each socket.

A typical use case is to append received lines to a common text file, which is later used by software (such as Wireshark) to decrypt the traffic:

```
const logFile = fs.createWriteStream('/tmp/ssl-keys.log', { flags: 'a' });
// ...
server.on('keylog', (line, tlsSocket) => {
  if (tlsSocket.remoteAddress !== '...')
    return; // Only log keys for a particular IP
  logFile.write(line);
});
```

Event: 'newSession'

The `'newSession'` event is emitted upon creation of a new TLS session. This may be used to store sessions in external storage. The data should be provided to the `'resumeSession'` callback.

The listener callback is passed three arguments when called:

- `sessionId` `<Buffer>` The TLS session identifier
- `sessionData` `<Buffer>` The TLS session data
- `callback` `<Function>` A callback function taking no arguments that must be invoked in order for data to be sent or received over the secure connection.

Listening for this event will have an effect only on connections established after the addition of the event listener.

Event: 'OCSPRequest'

The `'OCSPRequest'` event is emitted when the client sends a certificate status request. The listener callback is passed three arguments when called:

- `certificate` `<Buffer>` The server certificate
- `issuer` `<Buffer>` The issuer's certificate
- `callback` `<Function>` A callback function that must be invoked to provide the results of the OCSP request.

The server's current certificate can be parsed to obtain the OCSP URL and certificate ID; after obtaining an OCSP response, `callback(null, resp)` is then invoked, where `resp` is a `Buffer` instance containing the OCSP response. Both `certificate` and `issuer` are `Buffer` DER-representations of the primary and issuer's certificates. These can be used to obtain the OCSP certificate ID and OCSP endpoint URL.

Alternatively, `callback(null, null)` may be called, indicating that there was no OCSP response.

Calling `callback(err)` will result in a `socket.destroy(err)` call.

The typical flow of an OCSP Request is as follows:

1. Client connects to the server and sends an `'OCSPRequest'` (via the status info extension in `ClientHello`).
2. Server receives the request and emits the `'OCSPRequest'` event, calling the listener if registered.
3. Server extracts the OCSP URL from either the `certificate` or `issuer` and performs an `OCSP request` to the CA.
4. Server receives `'OCSPResponse'` from the CA and sends it back to the client via the `callback` argument
5. Client validates the response and either destroys the socket or performs a handshake.

The `issuer` can be `null` if the certificate is either self-signed or the issuer is not in the root certificates list. (An issuer may be provided via the `ca` option when establishing the TLS connection.)

Listening for this event will have an effect only on connections established after the addition of the event listener.

An npm module like `asn1.js` may be used to parse the certificates.

Event: `'resumeSession'`

The `'resumeSession'` event is emitted when the client requests to resume a previous TLS session. The listener callback is passed two arguments when called:

- `sessionId` `<Buffer>` The TLS session identifier
- `callback` `<Function>` A callback function to be called when the prior session has been recovered: `callback([err[, sessionId]])`
 - `err` `<Error>`
 - `sessionId` `<Buffer>`

The event listener should perform a lookup in external storage for the `sessionId` saved by the `'newSession'` event handler using the given `sessionId`. If found, call `callback(null, sessionId)` to resume the session. If not found, the session cannot be resumed. `callback()` must be called without `sessionId` so that the handshake can continue and a new session can be created. It is possible to call `callback(err)` to terminate the incoming connection and destroy the socket.

Listening for this event will have an effect only on connections established after the addition of the event listener.

The following illustrates resuming a TLS session:

```
const tlsSessionStore = {};  
server.on('newSession', (id, data, cb) => {  
  tlsSessionStore[id.toString('hex')] = data;  
  cb();  
});  
server.on('resumeSession', (id, cb) => {  
  cb(null, tlsSessionStore[id.toString('hex')] || null);  
});
```

Event: `'secureConnection'`

The `'secureConnection'` event is emitted after the handshaking process for a new connection has successfully completed. The listener callback is passed a single argument when called:

- `tlsSocket` `<tls.TLSSocket>` The established TLS socket.

The `tlsSocket.authorized` property is a `boolean` indicating whether the client has been verified by one of the supplied Certificate Authorities for the server. If `tlsSocket.authorized` is `false`, then `socket.authorizationError` is set to describe how authorization failed. Depending on the settings of the TLS server, unauthorized connections may still be accepted.

The `tlsSocket.alpnProtocol` property is a string that contains the selected ALPN protocol. When ALPN has no selected protocol, `tlsSocket.alpnProtocol` equals `false`.

The `tlsSocket.servername` property is a string containing the server name requested via SNI.

Event: 'tlsClientError'

The `'tlsClientError'` event is emitted when an error occurs before a secure connection is established. The listener callback is passed two arguments when called:

- `exception` `<Error>` The `Error` object describing the error
- `tlsSocket` `<tls.TLSSocket>` The `tls.TLSSocket` instance from which the error originated.

server.addContext(hostname, context)

- `hostname` `<string>` A SNI host name or wildcard (e.g. `'*'`)
- `context` `<Object>` An object containing any of the possible properties from the `tls.createSecureContext()` `options` arguments (e.g. `key`, `cert`, `ca`, etc).

The `server.addContext()` method adds a secure context that will be used if the client request's SNI name matches the supplied `hostname` (or wildcard).

When there are multiple matching contexts, the most recently added one is used.

server.address()

- Returns: `<Object>`

Returns the bound address, the address family name, and port of the server as reported by the operating system. See `net.Server.address()` for more information.

server.close([callback])

- `callback` `<Function>` A listener callback that will be registered to listen for the server instance's `'close'` event.
- Returns: `<tls.Server>`

The `server.close()` method stops the server from accepting new connections.

This function operates asynchronously. The `'close'` event will be emitted when the server has no more open connections.

server.getTicketKeys()

- Returns: `<Buffer>` A 48-byte buffer containing the session ticket keys.

Returns the session ticket keys.

See [Session Resumption](#) for more information.

server.listen()

Starts the server listening for encrypted connections. This method is identical to `server.listen()` from `net.Server`.

server.setSecureContext(options)

- `options` `<Object>` An object containing any of the possible properties from the `tls.createSecureContext()` `options` arguments (e.g. `key`, `cert`, `ca`, etc).

The `server.setSecureContext()` method replaces the secure context of an existing server. Existing connections to the server are not interrupted.

`server.setTicketKeys(keys)`

- `keys` `<Buffer>` | `<TypedArray>` | `<DataView>` A 48-byte buffer containing the session ticket keys.

Sets the session ticket keys.

Changes to the ticket keys are effective only for future server connections. Existing or currently pending server connections will use the previous keys.

See [Session Resumption](#) for more information.

Class: `tls.TLSSocket`

- Extends: `<net.Socket>`

Performs transparent encryption of written data and all required TLS negotiation.

Instances of `tls.TLSSocket` implement the duplex `Stream` interface.

Methods that return TLS connection metadata (e.g. `tls.TLSSocket.getPeerCertificate()`) will only return data while the connection is open.

`new tls.TLSSocket(socket[, options])`

- `socket` `<net.Socket>` | `<stream.Duplex>` On the server side, any `Duplex` stream. On the client side, any instance of `net.Socket` (for generic `Duplex` stream support on the client side, `tls.connect()` must be used).
- `options` `<Object>`
 - `enableTrace`: See `tls.createServer()`
 - `isServer`: The SSL/TLS protocol is asymmetrical, `TLSSockets` must know if they are to behave as a server or a client. If `true` the `TLS` socket will be instantiated as a server. **Default:** `false`.
 - `server` `<net.Server>` A `net.Server` instance.
 - `requestCert`: Whether to authenticate the remote peer by requesting a certificate. Clients always request a server certificate. Servers (`isServer` is true) may set `requestCert` to true to request a client certificate.
 - `rejectUnauthorized`: See `tls.createServer()`
 - `ALPNProtocols`: See `tls.createServer()`
 - `SNICallback`: See `tls.createServer()`
 - `session` `<Buffer>` A `Buffer` instance containing a `TLS` session.
 - `requestOCSP` `<boolean>` If `true`, specifies that the OCSP status request extension will be added to the client hello and an `'OCSPResponse'` event will be emitted on the socket before establishing a secure communication
 - `secureContext`: `TLS` context object created with `tls.createSecureContext()`. If a `secureContext` is *not* provided, one will be created by passing the entire `options` object to `tls.createSecureContext()`.
 - `...`: `tls.createSecureContext()` options that are used if the `secureContext` option is missing. Otherwise, they are ignored.

Construct a new `tls.TLSSocket` object from an existing TCP socket.

Event: `'keylog'`

- `line` `<Buffer>` Line of ASCII text, in NSS `SSLKEYLOGFILE` format.

The `keylog` event is emitted on a `tls.TLSSocket` when key material is generated or received by the socket. This keying material can be stored for debugging, as it allows captured TLS traffic to be decrypted. It may be emitted multiple times, before or after the handshake completes.

A typical use case is to append received lines to a common text file, which is later used by software (such as Wireshark) to decrypt the traffic:

```
const logFile = fs.createWriteStream('/tmp/ssl-keys.log', { flags: 'a' });
// ...
tlsSocket.on('keylog', (line) => logFile.write(line));
```

Event: 'OCSPResponse'

The `'OCSPResponse'` event is emitted if the `requestOCSP` option was set when the `tls.TLSSocket` was created and an OCSP response has been received. The listener callback is passed a single argument when called:

- `response` `<Buffer>` The server's OCSP response

Typically, the `response` is a digitally signed object from the server's CA that contains information about server's certificate revocation status.

Event: 'secureConnect'

The `'secureConnect'` event is emitted after the handshaking process for a new connection has successfully completed. The listener callback will be called regardless of whether or not the server's certificate has been authorized. It is the client's responsibility to check the `tlsSocket.authorized` property to determine if the server certificate was signed by one of the specified CAs. If `tlsSocket.authorized === false`, then the error can be found by examining the `tlsSocket.authorizationError` property. If ALPN was used, the `tlsSocket.alpnProtocol` property can be checked to determine the negotiated protocol.

The `'secureConnect'` event is not emitted when a `<tls.TLSSocket>` is created using the `new tls.TLSSocket()` constructor.

Event: 'session'

- `session` `<Buffer>`

The `'session'` event is emitted on a client `tls.TLSSocket` when a new session or TLS ticket is available. This may or may not be before the handshake is complete, depending on the TLS protocol version that was negotiated. The event is not emitted on the server, or if a new session was not created, for example, when the connection was resumed. For some TLS protocol versions the event may be emitted multiple times, in which case all the sessions can be used for resumption.

On the client, the `session` can be provided to the `session` option of `tls.connect()` to resume the connection.

See [Session Resumption](#) for more information.

For TLSv1.2 and below, `tls.TLSSocket.getSession()` can be called once the handshake is complete. For TLSv1.3, only ticket-based resumption is allowed by the protocol, multiple tickets are sent, and the tickets aren't sent until after the handshake completes. So it is necessary to wait for the `'session'` event to get a resumable session. Applications should use the `'session'` event instead of `getSession()` to ensure they will work for all TLS versions. Applications that only expect to get or use one session should listen for this event only once:

```
tlsSocket.once('session', (session) => {
  // The session can be used immediately or later.
  tls.connect({
    session: session,
    // Other connect options...
  });
});
```

tlsSocket.address()

- Returns: `<Object>`

Returns the bound `address`, the address `family` name, and `port` of the underlying socket as reported by the operating system: `{ port: 12346, family: 'IPv4', address: '127.0.0.1' }`.

tlsSocket.authorizationError

Returns the reason why the peer's certificate was not been verified. This property is set only when `tlsSocket.authorized === false`.

tlsSocket.authorized

- Returns: `<boolean>`

Returns `true` if the peer certificate was signed by one of the CAs specified when creating the `tls.TLSSocket` instance, otherwise `false`.

tlsSocket.disableRenegotiation()

Disables TLS renegotiation for this `TLSSocket` instance. Once called, attempts to renegotiate will trigger an `'error'` event on the `TLSSocket`.

tlsSocket.enableTrace()

When enabled, TLS packet trace information is written to `stderr`. This can be used to debug TLS connection problems.

Note: The format of the output is identical to the output of `openssl s_client -trace` or `openssl s_server -trace`. While it is produced by OpenSSL's `SSL_trace()` function, the format is undocumented, can change without notice, and should not be relied on.

tlsSocket.encrypted

Always returns `true`. This may be used to distinguish TLS sockets from regular `net.Socket` instances.

tlsSocket.exportKeyingMaterial(length, label[, context])

- `length` `<number>` number of bytes to retrieve from keying material
- `label` `<string>` an application specific label, typically this will be a value from the [IANA Exporter Label Registry](#).
- `context` `<Buffer>` Optionally provide a context.
- Returns: `<Buffer>` requested bytes of the keying material

Keying material is used for validations to prevent different kind of attacks in network protocols, for example in the specifications of IEEE 802.1X.

Example

```
const keyingMaterial = tlsSocket.exportKeyingMaterial(
  128,
  'client finished');

/**
 * Example return value of keyingMaterial:
 * <Buffer 76 26 af 99 c5 56 8e 42 09 91 ef 9f 93 cb ad 6c 7b 65 f8 53 f1 d8 d9
```

```
12 5a 33 b8 b5 25 df 7b 37 9f e0 e2 4f b8 67 83 a3 2f cd 5d 41 42 4c 91
74 ef 2c ... 78 more bytes>

*/
```

See the OpenSSL [SSL_export_keying_material](#) documentation for more information.

tlsSocket.getCertificate()

- Returns: `<Object>`

Returns an object representing the local certificate. The returned object has some properties corresponding to the fields of the certificate.

See `tls.TLSocket.getPeerCertificate()` for an example of the certificate structure.

If there is no local certificate, an empty object will be returned. If the socket has been destroyed, `null` will be returned.

tlsSocket.getCipher()

- Returns: `<Object>`
 - `name` `<string>` OpenSSL name for the cipher suite.
 - `standardName` `<string>` IETF name for the cipher suite.
 - `version` `<string>` The minimum TLS protocol version supported by this cipher suite.

Returns an object containing information on the negotiated cipher suite.

For example:

```
{
  "name": "AES128-SHA256",
  "standardName": "TLS_RSA_WITH_AES_128_CBC_SHA256",
  "version": "TLSv1.2"
}
```

See [SSL_CIPHER_get_name](#) for more information.

tlsSocket.getEphemeralKeyInfo()

- Returns: `<Object>`

Returns an object representing the type, name, and size of parameter of an ephemeral key exchange in [perfect forward secrecy](#) on a client connection. It returns an empty object when the key exchange is not ephemeral. As this is only supported on a client socket; `null` is returned if called on a server socket. The supported types are `'DH'` and `'ECDH'`. The `name` property is available only when type is `'ECDH'`.

For example: `{ type: 'ECDH', name: 'prime256v1', size: 256 }`.

tlsSocket.getFinished()

- Returns: `<Buffer>` | `<undefined>` The latest `Finished` message that has been sent to the socket as part of a SSL/TLS handshake, or `undefined` if no `Finished` message has been sent yet.

As the `Finished` messages are message digests of the complete handshake (with a total of 192 bits for TLS 1.0 and more for SSL 3.0), they can be used for external authentication procedures when the authentication provided by SSL/TLS is not desired or is not enough.

Corresponds to the `SSL_get_finished` routine in OpenSSL and may be used to implement the `tls-unique` channel binding from [RFC 5929](#).

tlsSocket.getPeerCertificate([detailed])

- `detailed` `<boolean>` Include the full certificate chain if `true`, otherwise include just the peer's certificate.
- Returns: `<Object>` A certificate object.

Returns an object representing the peer's certificate. If the peer does not provide a certificate, an empty object will be returned. If the socket has been destroyed, `null` will be returned.

If the full certificate chain was requested, each certificate will include an `issuerCertificate` property containing an object representing its issuer's certificate.

Certificate object

A certificate object has properties corresponding to the fields of the certificate.

- `raw` `<Buffer>` The DER encoded X.509 certificate data.
- `subject` `<Object>` The certificate subject, described in terms of Country (`C`), StateOrProvince (`ST`), Locality (`L`), Organization (`O`), OrganizationalUnit (`OU`), and CommonName (`CN`). The CommonName is typically a DNS name with TLS certificates. Example: `{C: 'UK', ST: 'BC', L: 'Metro', O: 'Node Fans', OU: 'Docs', CN: 'example.com'}`.
- `issuer` `<Object>` The certificate issuer, described in the same terms as the `subject`.
- `valid_from` `<string>` The date-time the certificate is valid from.
- `valid_to` `<string>` The date-time the certificate is valid to.
- `serialNumber` `<string>` The certificate serial number, as a hex string. Example: `'B9B0D332A1AA5635'`.
- `fingerprint` `<string>` The SHA-1 digest of the DER encoded certificate. It is returned as a `:` separated hexadecimal string. Example: `'2A:7A:C2:DD:...'`.
- `fingerprint256` `<string>` The SHA-256 digest of the DER encoded certificate. It is returned as a `:` separated hexadecimal string. Example: `'2A:7A:C2:DD:...'`.
- `ext_key_usage` `<Array>` (Optional) The extended key usage, a set of OIDs.
- `subjectAltname` `<string>` (Optional) A string containing concatenated names for the subject, an alternative to the `subject` names.
- `infoAccess` `<Array>` (Optional) An array describing the AuthorityInfoAccess, used with OCSP.
- `issuerCertificate` `<Object>` (Optional) The issuer certificate object. For self-signed certificates, this may be a circular reference.

The certificate may contain information about the public key, depending on the key type.

For RSA keys, the following properties may be defined:

- `bits` `<number>` The RSA bit size. Example: `1024`.
- `exponent` `<string>` The RSA exponent, as a string in hexadecimal number notation. Example: `'0x010001'`.
- `modulus` `<string>` The RSA modulus, as a hexadecimal string. Example: `'B56CE45CB7...'`.
- `pubkey` `<Buffer>` The public key.

For EC keys, the following properties may be defined:

- `pubkey` `<Buffer>` The public key.
- `bits` `<number>` The key size in bits. Example: `256`.
- `asn1Curve` `<string>` (Optional) The ASN.1 name of the OID of the elliptic curve. Well-known curves are identified by an OID. While it is unusual, it is possible that the curve is identified by its mathematical properties, in which case it will not have an OID. Example: `'prime256v1'`.
- `nistCurve` `<string>` (Optional) The NIST name for the elliptic curve, if it has one (not all well-known curves have been assigned names by NIST). Example: `'P-256'`.

Example certificate:

```

{ subject:
  { OU: [ 'Domain Control Validated', 'PositiveSSL Wildcard' ],
    CN: '*.nodejs.org' },
  issuer:
    { C: 'GB',
      ST: 'Greater Manchester',
      L: 'Salford',
      O: 'COMODO CA Limited',
      CN: 'COMODO RSA Domain Validation Secure Server CA' },
  subjectaltname: 'DNS:*.nodejs.org, DNS:nodejs.org',
  infoAccess:
    { 'CA Issuers - URI':
      [ 'http://crt.comodoca.com/COMODORSADomainValidationSecureServerCA.crt' ],

      'OCSP - URI': [ 'http://ocsp.comodoca.com' ] },
  modulus: 'B56CE45CB740B09A13F64AC543B712FF9EE8E4C284B542A1708A27E82A8D151CA178153E12E6DDA15BF70FFD96CB8A88618641BDFCCA6',
  exponent: '0x10001',
  pubkey: <Buffer ... >,
  valid_from: 'Aug 14 00:00:00 2017 GMT',
  valid_to: 'Nov 20 23:59:59 2019 GMT',
  fingerprint: '01:02:59:D9:C3:D2:0D:08:F7:82:4E:44:A4:B4:53:C5:E2:3A:87:4D',
  fingerprint256: '69:AE:1A:6A:D4:3D:C6:C1:1B:EA:C6:23:DE:BA:2A:14:62:62:93:5C:7A:EA:06:41:9B:0B:BC:87:CE:48:4E:02',
  ext_key_usage: [ '1.3.6.1.5.5.7.3.1', '1.3.6.1.5.5.7.3.2' ],
  serialNumber: '66593D57F20CBC573E433381B5FEC280',
  raw: <Buffer ... > }

```

tlsSocket.getPeerFinished()

- Returns: `<Buffer>` | `<undefined>` The latest `Finished` message that is expected or has actually been received from the socket as part of a SSL/TLS handshake, or `undefined` if there is no `Finished` message so far.

As the `Finished` messages are message digests of the complete handshake (with a total of 192 bits for TLS 1.0 and more for SSL 3.0), they can be used for external authentication procedures when the authentication provided by SSL/TLS is not desired or is not enough.

Corresponds to the `SSL_get_peer_finished` routine in OpenSSL and may be used to implement the `tls-unique` channel binding from [RFC 5929](#).

tlsSocket.getPeerX509Certificate()

- Returns: `<X509Certificate>`

Returns the peer certificate as an `<X509Certificate>` object.

If there is no peer certificate, or the socket has been destroyed, `undefined` will be returned.

tlsSocket.getProtocol()

- Returns: `<string>` | `<null>`

Returns a string containing the negotiated SSL/TLS protocol version of the current connection. The value `'unknown'` will be returned for connected sockets that have not completed the handshaking process. The value `null` will be returned for server sockets or disconnected client sockets.

Protocol versions are:

- 'SSLv3'
- 'TLSv1'
- 'TLSv1.1'
- 'TLSv1.2'
- 'TLSv1.3'

See the OpenSSL [SSL_get_version](#) documentation for more information.

tlsSocket.getSession()

- `<Buffer>`

Returns the TLS session data or `undefined` if no session was negotiated. On the client, the data can be provided to the `session` option of `tls.connect()` to resume the connection. On the server, it may be useful for debugging.

See [Session Resumption](#) for more information.

Note: `getSession()` works only for TLSv1.2 and below. For TLSv1.3, applications must use the `'session'` event (it also works for TLSv1.2 and below).

tlsSocket.getSharedSigalgs()

- Returns: `<Array>` List of signature algorithms shared between the server and the client in the order of decreasing preference.

See [SSL_get_shared_sigalgs](#) for more information.

tlsSocket.getTLSTicket()

- `<Buffer>`

For a client, returns the TLS session ticket if one is available, or `undefined`. For a server, always returns `undefined`.

It may be useful for debugging.

See [Session Resumption](#) for more information.

tlsSocket.getX509Certificate()

- Returns: `<X509Certificate>`

Returns the local certificate as an `<X509Certificate>` object.

If there is no local certificate, or the socket has been destroyed, `undefined` will be returned.

tlsSocket.isSessionReused()

- Returns: `<boolean>` `true` if the session was reused, `false` otherwise.

See [Session Resumption](#) for more information.

tlsSocket.localAddress

- `<string>`

Returns the string representation of the local IP address.

tlsSocket.localPort

- `<number>`

Returns the numeric representation of the local port.

`tlsSocket.remoteAddress`

- `<string>`

Returns the string representation of the remote IP address. For example, `'74.125.127.100'` or `'2001:4860:a005::68'`.

`tlsSocket.remoteFamily`

- `<string>`

Returns the string representation of the remote IP family. `'IPv4'` or `'IPv6'`.

`tlsSocket.remotePort`

- `<number>`

Returns the numeric representation of the remote port. For example, `443`.

`tlsSocket.renegotiate(options, callback)`

- `options` `<Object>`
 - `rejectUnauthorized` `<boolean>` If not `false`, the server certificate is verified against the list of supplied CAs. An `'error'` event is emitted if verification fails; `err.code` contains the OpenSSL error code. **Default:** `true`.
 - `requestCert`
- `callback` `<Function>` If `renegotiate()` returned `true`, `callback` is attached once to the `'secure'` event. If `renegotiate()` returned `false`, `callback` will be called in the next tick with an error, unless the `tlsSocket` has been destroyed, in which case `callback` will not be called at all.
- Returns: `<boolean>` `true` if renegotiation was initiated, `false` otherwise.

The `tlsSocket.renegotiate()` method initiates a TLS renegotiation process. Upon completion, the `callback` function will be passed a single argument that is either an `Error` (if the request failed) or `null`.

This method can be used to request a peer's certificate after the secure connection has been established.

When running as the server, the socket will be destroyed with an error after `handshakeTimeout` timeout.

For TLSv1.3, renegotiation cannot be initiated, it is not supported by the protocol.

`tlsSocket.setMaxSendFragment(size)`

- `size` `<number>` The maximum TLS fragment size. The maximum value is `16384`. **Default:** `16384`.
- Returns: `<boolean>`

The `tlsSocket.setMaxSendFragment()` method sets the maximum TLS fragment size. Returns `true` if setting the limit succeeded; `false` otherwise.

Smaller fragment sizes decrease the buffering latency on the client: larger fragments are buffered by the TLS layer until the entire fragment is received and its integrity is verified; large fragments can span multiple roundtrips and their processing can be delayed due to packet loss or reordering. However, smaller fragments add extra TLS framing bytes and CPU overhead, which may decrease overall server throughput.

`tls.checkServerIdentity(hostname, cert)`

- `hostname` `<string>` The host name or IP address to verify the certificate against.

- `cert` `<Object>` A `certificate object` representing the peer's certificate.
- Returns: `<Error>` | `<undefined>`

Verifies the certificate `cert` is issued to `hostname`.

Returns `<Error>` object, populating it with `reason`, `host`, and `cert` on failure. On success, returns `<undefined>`.

This function can be overwritten by providing alternative function as part of the `options.checkServerIdentity` option passed to `tls.connect()`. The overwriting function can call `tls.checkServerIdentity()` of course, to augment the checks done with additional verification.

This function is only called if the certificate passed all other checks, such as being issued by trusted CA (`options.ca`).

tls.connect(options[, callback])

- `options` `<Object>`
 - `enableTrace`: See `tls.createServer()`
 - `host` `<string>` Host the client should connect to. **Default:** `'localhost'`.
 - `port` `<number>` Port the client should connect to.
 - `path` `<string>` Creates Unix socket connection to path. If this option is specified, `host` and `port` are ignored.
 - `socket` `<stream.Duplex>` Establish secure connection on a given socket rather than creating a new socket. Typically, this is an instance of `net.Socket`, but any `Duplex` stream is allowed. If this option is specified, `path`, `host` and `port` are ignored, except for certificate validation. Usually, a socket is already connected when passed to `tls.connect()`, but it can be connected later. Connection/disconnection/destruction of `socket` is the user's responsibility; calling `tls.connect()` will not cause `net.connect()` to be called.
 - `allowHalfOpen` `<boolean>` If set to `false`, then the socket will automatically end the writable side when the readable side ends. If the `socket` option is set, this option has no effect. See the `allowHalfOpen` option of `net.Socket` for details. **Default:** `false`.
 - `rejectUnauthorized` `<boolean>` If not `false`, the server certificate is verified against the list of supplied CAs. An `'error'` event is emitted if verification fails; `err.code` contains the OpenSSL error code. **Default:** `true`.
 - `pskCallback` `<Function>`
 - `hint`: `<string>` optional message sent from the server to help client decide which identity to use during negotiation. Always `null` if TLS 1.3 is used.
 - Returns: `<Object>` in the form `{ psk: <Buffer|TypedArray|DataView>, identity: <string> }` or `null` to stop the negotiation process. `psk` must be compatible with the selected cipher's digest. `identity` must use UTF-8 encoding.

When negotiating TLS-PSK (pre-shared keys), this function is called with optional `identity hint` provided by the server or `null` in case of TLS 1.3 where `hint` was removed. It will be necessary to provide a custom `tls.checkServerIdentity()` for the connection as the default one will try to check host name/IP of the server against the certificate but that's not applicable for PSK because there won't be a certificate present. More information can be found in the [RFC 4279](#).
 - `ALPNProtocols`: `<string[]>` | `<Buffer[]>` | `<TypedArray[]>` | `<DataView[]>` | `<Buffer>` | `<TypedArray>` | `<DataView>`
An array of strings, `Buffer`s or `TypedArray`s or `DataView`s, or a single `Buffer` or `TypedArray` or `DataView` containing the supported ALPN protocols. `Buffer`s should have the format `[len][name][len][name]...` e.g. `'\x08http/1.1\x08http/1.0'`, where the `len` byte is the length of the next protocol name. Passing an array is usually much simpler, e.g. `['http/1.1', 'http/1.0']`. Protocols earlier in the list have higher preference than those later.
 - `servername`: `<string>` Server name for the SNI (Server Name Indication) TLS extension. It is the name of the host being connected to, and must be a host name, and not an IP address. It can be used by a multi-homed server to choose the correct certificate to present to the client, see the `SNICallback` option to `tls.createServer()`.
 - `checkServerIdentity(servername, cert)` `<Function>` A callback function to be used (instead of the builtin `tls.checkServerIdentity()` function) when checking the server's host name (or the provided `servername` when explicitly set) against the certificate. This should return an `<Error>` if verification fails. The method should return `undefined` if the `servername` and `cert` are verified.
 - `session` `<Buffer>` A `Buffer` instance, containing TLS session.

- `minDHSize` `<number>` Minimum size of the DH parameter in bits to accept a TLS connection. When a server offers a DH parameter with a size less than `minDHSize`, the TLS connection is destroyed and an error is thrown. **Default:** 1024.
- `highWaterMark`: `<number>` Consistent with the readable stream `highWaterMark` parameter. **Default:** 16 * 1024.
- `secureContext`: TLS context object created with `tls.createSecureContext()`. If a `secureContext` is *not* provided, one will be created by passing the entire `options` object to `tls.createSecureContext()`.
- `onread` `<Object>` If the `socket` option is missing, incoming data is stored in a single `buffer` and passed to the supplied `callback` when data arrives on the socket, otherwise the option is ignored. See the `onread` option of `net.Socket` for details.
- ...: `tls.createSecureContext()` options that are used if the `secureContext` option is missing, otherwise they are ignored.
- ...: Any `socket.connect()` option not already listed.
- `callback` `<Function>`
- Returns: `<tls.TLSSocket>`

The `callback` function, if specified, will be added as a listener for the `'secureConnect'` event.

`tls.connect()` returns a `tls.TLSSocket` object.

Unlike the `https` API, `tls.connect()` does not enable the SNI (Server Name Indication) extension by default, which may cause some servers to return an incorrect certificate or reject the connection altogether. To enable SNI, set the `servername` option in addition to `host`.

The following illustrates a client for the echo server example from `tls.createServer()`:

```
// Assumes an echo server that is listening on port 8000.
const tls = require('tls');
const fs = require('fs');

const options = {
  // Necessary only if the server requires client certificate authentication.
  key: fs.readFileSync('client-key.pem'),
  cert: fs.readFileSync('client-cert.pem'),

  // Necessary only if the server uses a self-signed certificate.
  ca: [ fs.readFileSync('server-cert.pem') ],

  // Necessary only if the server's cert isn't for "localhost".
  checkServerIdentity: () => { return null; },
};

const socket = tls.connect(8000, options, () => {
  console.log('client connected',
    socket.authorized ? 'authorized' : 'unauthorized');
  process.stdin.pipe(socket);
  process.stdin.resume();
});
socket.setEncoding('utf8');
socket.on('data', (data) => {
  console.log(data);
});
socket.on('end', () => {
  console.log('server ends connection');
});
```

tls.connect(path[, options][, callback])

- `path` `<string>` Default value for `options.path`.
- `options` `<Object>` See `tls.connect()`.
- `callback` `<Function>` See `tls.connect()`.
- Returns: `<tls.TLSSocket>`

Same as `tls.connect()` except that `path` can be provided as an argument instead of an option.

A path option, if specified, will take precedence over the path argument.

tls.connect(port[, host][, options][, callback])

- `port` `<number>` Default value for `options.port`.
- `host` `<string>` Default value for `options.host`.
- `options` `<Object>` See `tls.connect()`.
- `callback` `<Function>` See `tls.connect()`.
- Returns: `<tls.TLSSocket>`

Same as `tls.connect()` except that `port` and `host` can be provided as arguments instead of options.

A port or host option, if specified, will take precedence over any port or host argument.

tls.createSecureContext([options])

- `options` `<Object>`
 - `ca` `<string> | <string[]> | <Buffer> | <Buffer[]>` Optionally override the trusted CA certificates. Default is to trust the well-known CAs curated by Mozilla. Mozilla's CAs are completely replaced when CAs are explicitly specified using this option. The value can be a string or `Buffer`, or an `Array` of strings and/or `Buffer`s. Any string or `Buffer` can contain multiple PEM CAs concatenated together. The peer's certificate must be chainable to a CA trusted by the server for the connection to be authenticated. When using certificates that are not chainable to a well-known CA, the certificate's CA must be explicitly specified as a trusted or the connection will fail to authenticate. If the peer uses a certificate that doesn't match or chain to one of the default CAs, use the `ca` option to provide a CA certificate that the peer's certificate can match or chain to. For self-signed certificates, the certificate is its own CA, and must be provided. For PEM encoded certificates, supported types are "TRUSTED CERTIFICATE", "X509 CERTIFICATE", and "CERTIFICATE". See also `tls.rootCertificates`.
 - `cert` `<string> | <string[]> | <Buffer> | <Buffer[]>` Cert chains in PEM format. One cert chain should be provided per private key. Each cert chain should consist of the PEM formatted certificate for a provided private `key`, followed by the PEM formatted intermediate certificates (if any), in order, and not including the root CA (the root CA must be pre-known to the peer, see `ca`). When providing multiple cert chains, they do not have to be in the same order as their private keys in `key`. If the intermediate certificates are not provided, the peer will not be able to validate the certificate, and the handshake will fail.
 - `sigalgs` `<string>` Colon-separated list of supported signature algorithms. The list can contain digest algorithms (SHA256, MD5 etc.), public key algorithms (RSA-PSS, ECDSA etc.), combination of both (e.g. 'RSA+SHA384') or TLS v1.3 scheme names (e.g. `rsa_pss_pss_sha512`). See [OpenSSL man pages](#) for more info.
 - `ciphers` `<string>` Cipher suite specification, replacing the default. For more information, see [modifying the default cipher suite](#). Permitted ciphers can be obtained via `tls.getCiphers()`. Cipher names must be uppercased in order for OpenSSL to accept them.
 - `clientCertEngine` `<string>` Name of an OpenSSL engine which can provide the client certificate.
 - `crl` `<string> | <string[]> | <Buffer> | <Buffer[]>` PEM formatted CRLs (Certificate Revocation Lists).
 - `dhparam` `<string> | <Buffer>` Diffie-Hellman parameters, required for [perfect forward secrecy](#). Use `openssl dhparam` to create the parameters. The key length must be greater than or equal to 1024 bits or else an error will be thrown. Although 1024 bits is permissible, use 2048 bits or larger for stronger security. If omitted or invalid, the parameters are silently discarded and DHE ciphers will not be available.

- `ecdhCurve` `<string>` A string describing a named curve or a colon separated list of curve NIDs or names, for example `P-521:P-384:P-256`, to use for ECDH key agreement. Set to `auto` to select the curve automatically. Use `crypto.getCurves()` to obtain a list of available curve names. On recent releases, `openssl ecparam -list_curves` will also display the name and description of each available elliptic curve. **Default:** `tls.DEFAULT_ECDH_CURVE`.
- `honorCipherOrder` `<boolean>` Attempt to use the server's cipher suite preferences instead of the client's. When `true`, causes `SSL_OP_CIPHER_SERVER_PREFERENCE` to be set in `secureOptions`, see [OpenSSL Options](#) for more information.
- `key` `<string>` | `<string[]>` | `<Buffer>` | `<Buffer[]>` | `<Object[]>` Private keys in PEM format. PEM allows the option of private keys being encrypted. Encrypted keys will be decrypted with `options.passphrase`. Multiple keys using different algorithms can be provided either as an array of unencrypted key strings or buffers, or an array of objects in the form `{pem: <string|buffer>[, passphrase: <string>]}`. The object form can only occur in an array. `object.passphrase` is optional. Encrypted keys will be decrypted with `object.passphrase` if provided, or `options.passphrase` if it is not.
- `privateKeyEngine` `<string>` Name of an OpenSSL engine to get private key from. Should be used together with `privateKeyIdentifier`.
- `privateKeyIdentifier` `<string>` Identifier of a private key managed by an OpenSSL engine. Should be used together with `privateKeyEngine`. Should not be set together with `key`, because both options define a private key in different ways.
- `maxVersion` `<string>` Optionally set the maximum TLS version to allow. One of `'TLSv1.3'`, `'TLSv1.2'`, `'TLSv1.1'`, or `'TLSv1'`. Cannot be specified along with the `secureProtocol` option; use one or the other. **Default:** `tls.DEFAULT_MAX_VERSION`.
- `minVersion` `<string>` Optionally set the minimum TLS version to allow. One of `'TLSv1.3'`, `'TLSv1.2'`, `'TLSv1.1'`, or `'TLSv1'`. Cannot be specified along with the `secureProtocol` option; use one or the other. Avoid setting to less than TLSv1.2, but it may be required for interoperability. **Default:** `tls.DEFAULT_MIN_VERSION`.
- `passphrase` `<string>` Shared passphrase used for a single private key and/or a PFX.
- `pfx` `<string>` | `<string[]>` | `<Buffer>` | `<Buffer[]>` | `<Object[]>` PFX or PKCS12 encoded private key and certificate chain. `pfx` is an alternative to providing `key` and `cert` individually. PFX is usually encrypted, if it is, `passphrase` will be used to decrypt it. Multiple PFX can be provided either as an array of unencrypted PFX buffers, or an array of objects in the form `{buf: <string|buffer>[, passphrase: <string>]}`. The object form can only occur in an array. `object.passphrase` is optional. Encrypted PFX will be decrypted with `object.passphrase` if provided, or `options.passphrase` if it is not.
- `secureOptions` `<number>` Optionally affect the OpenSSL protocol behavior, which is not usually necessary. This should be used carefully if at all! Value is a numeric bitmask of the `SSL_OP_*` options from [OpenSSL Options](#).
- `secureProtocol` `<string>` Legacy mechanism to select the TLS protocol version to use, it does not support independent control of the minimum and maximum version, and does not support limiting the protocol to TLSv1.3. Use `minVersion` and `maxVersion` instead. The possible values are listed as `SSL_METHODS`, use the function names as strings. For example, use `'TLSv1_1_method'` to force TLS version 1.1, or `'TLS_method'` to allow any TLS protocol version up to TLSv1.3. It is not recommended to use TLS versions less than 1.2, but it may be required for interoperability. **Default:** none, see `minVersion`.
- `sessionIdContext` `<string>` Opaque identifier used by servers to ensure session state is not shared between applications. Unused by clients.
- `ticketKeys` `<Buffer>` 48-bytes of cryptographically strong pseudorandom data. See [Session Resumption](#) for more information.
- `sessionTimeout` `<number>` The number of seconds after which a TLS session created by the server will no longer be resumable. See [Session Resumption](#) for more information. **Default:** `300`.

`tls.createServer()` sets the default value of the `honorCipherOrder` option to `true`, other APIs that create secure contexts leave it unset.

`tls.createServer()` uses a 128 bit truncated SHA1 hash value generated from `process.argv` as the default value of the `sessionIdContext` option, other APIs that create secure contexts have no default value.

The `tls.createSecureContext()` method creates a `SecureContext` object. It is usable as an argument to several `tls` APIs, such as `tls.createServer()` and `server.addContext()`, but has no public methods.

A key is required for ciphers that use certificates. Either `key` or `pfx` can be used to provide it.

If the `ca` option is not given, then Node.js will default to using [Mozilla's publicly trusted list of CAs](#).

`tls.createSecurePair([context][, isServer][, requestCert][, rejectUnauthorized][, options])`

Stability: 0 - Deprecated: Use `tls.TLSSocket` instead.

- `context` `<Object>` A secure context object as returned by `tls.createSecureContext()`
- `isServer` `<boolean>` `true` to specify that this TLS connection should be opened as a server.
- `requestCert` `<boolean>` `true` to specify whether a server should request a certificate from a connecting client. Only applies when `isServer` is `true`.
- `rejectUnauthorized` `<boolean>` If not `false` a server automatically reject clients with invalid certificates. Only applies when `isServer` is `true`.
- `options`
 - `enableTrace`: See `tls.createServer()`
 - `secureContext`: A TLS context object from `tls.createSecureContext()`
 - `isServer`: If `true` the TLS socket will be instantiated in server-mode. **Default:** `false`.
 - `server` `<net.Server>` A `net.Server` instance
 - `requestCert`: See `tls.createServer()`
 - `rejectUnauthorized`: See `tls.createServer()`
 - `ALPNProtocols`: See `tls.createServer()`
 - `SNICallback`: See `tls.createServer()`
 - `session` `<Buffer>` A `Buffer` instance containing a TLS session.
 - `requestOCSP` `<boolean>` If `true`, specifies that the OCSP status request extension will be added to the client hello and an 'OCSPResponse' event will be emitted on the socket before establishing a secure communication.

Creates a new secure pair object with two streams, one of which reads and writes the encrypted data and the other of which reads and writes the cleartext data. Generally, the encrypted stream is piped to/from an incoming encrypted data stream and the cleartext one is used as a replacement for the initial encrypted stream.

`tls.createSecurePair()` returns a `tls.SecurePair` object with `cleartext` and `encrypted` stream properties.

Using `cleartext` has the same API as `tls.TLSSocket`.

The `tls.createSecurePair()` method is now deprecated in favor of `tls.TLSSocket()`. For example, the code:

```
pair = tls.createSecurePair(/* ... */);
pair.encrypted.pipe(socket);
socket.pipe(pair.encrypted);
```

can be replaced by:

```
secureSocket = tls.TLSSocket(socket, options);
```

where `secureSocket` has the same API as `pair.cleartext`.

`tls.createServer([options][, secureConnectionListener])`

- `options` `<Object>`
 - `ALPNProtocols`: `<string[]> | <Buffer[]> | <TypedArray[]> | <DataView[]> | <Buffer> | <TypedArray> | <DataView>`
An array of strings, `Buffer` s or `TypedArray` s or `DataView` s, or a single `Buffer` or `TypedArray` or `DataView` containing the supported ALPN protocols. `Buffer` s should have the format `[len][name][len][name]...` e.g. `0x05hello0x05world`, where the first byte is the length of the next protocol name. Passing an array is usually much simpler, e.g. `['hello', 'world']`. (Protocols should be ordered by their priority.)
 - `clientCertEngine` `<string>` Name of an OpenSSL engine which can provide the client certificate.
 - `enableTrace` `<boolean>` If `true`, `tls.TLSSocket.enableTrace()` will be called on new connections. Tracing can be enabled after the secure connection is established, but this option must be used to trace the secure connection setup. **Default:** `false`.
 - `handshakeTimeout` `<number>` Abort the connection if the SSL/TLS handshake does not finish in the specified number of milliseconds. A `'tlsClientError'` is emitted on the `tls.Server` object whenever a handshake times out. **Default:** `120000` (120 seconds).
 - `rejectUnauthorized` `<boolean>` If not `false` the server will reject any connection which is not authorized with the list of supplied CAs. This option only has an effect if `requestCert` is `true`. **Default:** `true`.
 - `requestCert` `<boolean>` If `true` the server will request a certificate from clients that connect and attempt to verify that certificate. **Default:** `false`.
 - `sessionTimeout` `<number>` The number of seconds after which a TLS session created by the server will no longer be resumable. See [Session Resumption](#) for more information. **Default:** `300`.
 - `SNICallback(servername, callback)` `<Function>` A function that will be called if the client supports SNI TLS extension. Two arguments will be passed when called: `servername` and `callback`. `callback` is an error-first callback that takes two optional arguments: `error` and `ctx`. `ctx`, if provided, is a `SecureContext` instance. `tls.createSecureContext()` can be used to get a proper `SecureContext`. If `callback` is called with a falsy `ctx` argument, the default secure context of the server will be used. If `SNICallback` wasn't provided the default callback with high-level API will be used (see below).
 - `ticketKeys`: `<Buffer>` 48-bytes of cryptographically strong pseudorandom data. See [Session Resumption](#) for more information.
 - `pskCallback` `<Function>`
 - `socket`: `<tls.TLSSocket>` the server `tls.TLSSocket` instance for this connection.
 - `identity`: `<string>` identity parameter sent from the client.
 - **Returns:** `<Buffer> | <TypedArray> | <DataView>` pre-shared key that must either be a buffer or `null` to stop the negotiation process. Returned PSK must be compatible with the selected cipher's digest.

When negotiating TLS-PSK (pre-shared keys), this function is called with the identity provided by the client. If the return value is `null` the negotiation process will stop and an "unknown_psk_identity" alert message will be sent to the other party. If the server wishes to hide the fact that the PSK identity was not known, the callback must provide some random data as `psk` to make the connection fail with "decrypt_error" before negotiation is finished. PSK ciphers are disabled by default, and using TLS-PSK thus requires explicitly specifying a cipher suite with the `ciphers` option. More information can be found in the [RFC 4279](#).
 - `pskIdentityHint` `<string>` optional hint to send to a client to help with selecting the identity during TLS-PSK negotiation. Will be ignored in TLS 1.3. Upon failing to set `pskIdentityHint` `'tlsClientError'` will be emitted with `'ERR_TLS_PSK_SET_IDENTITY_HINT_FAILED'` code.
 - ...: Any `tls.createSecureContext()` option can be provided. For servers, the identity options (`pfx`, `key` / `cert` or `pskCallback`) are usually required.
 - ...: Any `net.createServer()` option can be provided.
- `secureConnectionListener` `<Function>`
- **Returns:** `<tls.Server>`

Creates a new `tls.Server`. The `secureConnectionListener`, if provided, is automatically set as a listener for the `'secureConnection'` event.

The `ticketKeys` options is automatically shared between `cluster` module workers.

The following illustrates a simple echo server:


```

const tls = require('tls');
const fs = require('fs');

const options = {
  key: fs.readFileSync('server-key.pem'),
  cert: fs.readFileSync('server-cert.pem'),

  // This is necessary only if using client certificate authentication.
  requestCert: true,

  // This is necessary only if the client uses a self-signed certificate.
  ca: [ fs.readFileSync('client-cert.pem') ]
};

const server = tls.createServer(options, (socket) => {
  console.log('server connected',
    socket.authorized ? 'authorized' : 'unauthorized');
  socket.write('welcome!\n');
  socket.setEncoding('utf8');
  socket.pipe(socket);
});
server.listen(8000, () => {
  console.log('server bound');
});

```

The server can be tested by connecting to it using the example client from `tls.connect()`.

tls.getCiphers()

- Returns: `<string[]>`

Returns an array with the names of the supported TLS ciphers. The names are lower-case for historical reasons, but must be uppercased to be used in the `ciphers` option of `tls.createSecureContext()`.

Cipher names that start with `'tls_'` are for TLSv1.3, all the others are for TLSv1.2 and below.

```

console.log(tls.getCiphers()); // ['aes128-gcm-sha256', 'aes128-sha', ...]

```

tls.rootCertificates

- `<string[]>`

An immutable array of strings representing the root certificates (in PEM format) from the bundled Mozilla CA store as supplied by current Node.js version.

The bundled CA store, as supplied by Node.js, is a snapshot of Mozilla CA store that is fixed at release time. It is identical on all supported platforms.

tls.DEFAULT_ECDH_CURVE

The default curve name to use for ECDH key agreement in a `tls` server. The default value is `'auto'`. See `tls.createSecureContext()` for further information.

`tls.DEFAULT_MAX_VERSION`

- `<string>` The default value of the `maxVersion` option of `tls.createSecureContext()`. It can be assigned any of the supported TLS protocol versions, `'TLSv1.3'`, `'TLSv1.2'`, `'TLSv1.1'`, or `'TLSv1'`. **Default:** `'TLSv1.3'`, unless changed using CLI options. Using `--tls-max-v1.2` sets the default to `'TLSv1.2'`. Using `--tls-max-v1.3` sets the default to `'TLSv1.3'`. If multiple of the options are provided, the highest maximum is used.

`tls.DEFAULT_MIN_VERSION`

- `<string>` The default value of the `minVersion` option of `tls.createSecureContext()`. It can be assigned any of the supported TLS protocol versions, `'TLSv1.3'`, `'TLSv1.2'`, `'TLSv1.1'`, or `'TLSv1'`. **Default:** `'TLSv1.2'`, unless changed using CLI options. Using `--tls-min-v1.0` sets the default to `'TLSv1'`. Using `--tls-min-v1.1` sets the default to `'TLSv1.1'`. Using `--tls-min-v1.3` sets the default to `'TLSv1.3'`. If multiple of the options are provided, the lowest minimum is used.

Trace events

Stability: 1 - Experimental

Source Code: [lib/trace_events.js](#)

The `trace_events` module provides a mechanism to centralize tracing information generated by V8, Node.js core, and userspace code.

Tracing can be enabled with the `--trace-event-categories` command-line flag or by using the `trace_events` module. The `--trace-event-categories` flag accepts a list of comma-separated category names.

The available categories are:

- `node`: An empty placeholder.
- `node.async_hooks`: Enables capture of detailed `async_hooks` trace data. The `async_hooks` events have a unique `asyncId` and a special `triggerId` `triggerAsyncId` property.
- `node.bootstrap`: Enables capture of Node.js bootstrap milestones.
- `node.console`: Enables capture of `console.time()` and `console.count()` output.
- `node.dns.native`: Enables capture of trace data for DNS queries.
- `node.environment`: Enables capture of Node.js Environment milestones.
- `node.fs.sync`: Enables capture of trace data for file system sync methods.
- `node.perf`: Enables capture of `Performance API` measurements.
 - `node.perf.usertiming`: Enables capture of only Performance API User Timing measures and marks.
 - `node.perf.timerify`: Enables capture of only Performance API timerify measurements.
- `node.promises.rejections`: Enables capture of trace data tracking the number of unhandled Promise rejections and handled-after-rejections.
- `node.vm.script`: Enables capture of trace data for the `vm` module's `runInNewContext()`, `runInContext()`, and `runInThisContext()` methods.
- `v8`: The `V8` events are GC, compiling, and execution related.

By default the `node`, `node.async_hooks`, and `v8` categories are enabled.

```
node --trace-event-categories v8,node,node.async_hooks server.js
```

Prior versions of Node.js required the use of the `--trace-events-enabled` flag to enable trace events. This requirement has been removed. However, the `--trace-events-enabled` flag *may* still be used and will enable the `node`, `node.async_hooks`, and `v8` trace event categories by default.

```
node --trace-events-enabled

# is equivalent to

node --trace-event-categories v8,node,node.async_hooks
```

Alternatively, trace events may be enabled using the `trace_events` module:

```
const trace_events = require('trace_events');
const tracing = trace_events.createTracing({ categories: ['node.perf'] });
tracing.enable(); // Enable trace event capture for the 'node.perf' category

// do work

tracing.disable(); // Disable trace event capture for the 'node.perf' category
```

Running Node.js with tracing enabled will produce log files that can be opened in the `chrome://tracing` tab of Chrome.

The logging file is by default called `node_trace.${rotation}.log`, where `${rotation}` is an incrementing log-rotation id. The filepath pattern can be specified with `--trace-event-file-pattern` that accepts a template string that supports `${rotation}` and `${pid}`:

```
node --trace-event-categories v8 --trace-event-file-pattern '${pid}-${rotation}.log' server.js
```

The tracing system uses the same time source as the one used by `process.hrtime()`. However the trace-event timestamps are expressed in microseconds, unlike `process.hrtime()` which returns nanoseconds.

The features from this module are not available in `Worker` threads.

The `trace_events` module

Tracing object

The `Tracing` object is used to enable or disable tracing for sets of categories. Instances are created using the `trace_events.createTracing()` method.

When created, the `Tracing` object is disabled. Calling the `tracing.enable()` method adds the categories to the set of enabled trace event categories. Calling `tracing.disable()` will remove the categories from the set of enabled trace event categories.

`tracing.categories`

- `<string>`

A comma-separated list of the trace event categories covered by this `Tracing` object.

tracing.disable()

Disables this `Tracing` object.

Only trace event categories *not* covered by other enabled `Tracing` objects and *not* specified by the `--trace-event-categories` flag will be disabled.

```
const trace_events = require('trace_events');
const t1 = trace_events.createTracing({ categories: ['node', 'v8'] });
const t2 = trace_events.createTracing({ categories: ['node.perf', 'node'] });
t1.enable();
t2.enable();

// Prints 'node,node.perf,v8'
console.log(trace_events.getEnabledCategories());

t2.disable(); // Will only disable emission of the 'node.perf' category

// Prints 'node,v8'
console.log(trace_events.getEnabledCategories());
```

tracing.enable()

Enables this `Tracing` object for the set of categories covered by the `Tracing` object.

tracing.enabled

- `<boolean>` `true` only if the `Tracing` object has been enabled.

trace_events.createTracing(options)

- `options` `<Object>`
 - `categories` `<string[]>` An array of trace category names. Values included in the array are coerced to a string when possible. An error will be thrown if the value cannot be coerced.
- Returns: `<Tracing>`.

Creates and returns a `Tracing` object for the given set of `categories`.

```
const trace_events = require('trace_events');
const categories = ['node.perf', 'node.async_hooks'];
const tracing = trace_events.createTracing({ categories });
tracing.enable();
// do stuff
tracing.disable();
```

trace_events.getEnabledCategories()

- Returns: `<string>`

Returns a comma-separated list of all currently-enabled trace event categories. The current set of enabled trace event categories is determined by the *union* of all currently-enabled `Tracing` objects and any categories enabled using the `--trace-event-categories` flag.

Given the file `test.js` below, the command `node --trace-event-categories node.perf test.js` will print `'node.async_hooks,node.perf'` to the console.

```
const trace_events = require('trace_events');
const t1 = trace_events.createTracing({ categories: ['node.async_hooks'] });
const t2 = trace_events.createTracing({ categories: ['node.perf'] });
const t3 = trace_events.createTracing({ categories: ['v8'] });

t1.enable();
t2.enable();

console.log(trace_events.getEnabledCategories());
```

TTY

Stability: 2 - Stable

Source Code: [lib/tty.js](#)

The `tty` module provides the `tty.ReadStream` and `tty.WriteStream` classes. In most cases, it will not be necessary or possible to use this module directly. However, it can be accessed using:

```
const tty = require('tty');
```

When Node.js detects that it is being run with a text terminal ("TTY") attached, `process.stdin` will, by default, be initialized as an instance of `tty.ReadStream` and both `process.stdout` and `process.stderr` will, by default, be instances of `tty.WriteStream`. The preferred method of determining whether Node.js is being run within a TTY context is to check that the value of the `process.stdout.isTTY` property is `true`:

```
$ node -p -e "Boolean(process.stdout.isTTY)"
true
$ node -p -e "Boolean(process.stdout.isTTY)" | cat
false
```

In most cases, there should be little to no reason for an application to manually create instances of the `tty.ReadStream` and `tty.WriteStream` classes.

Class: `tty.ReadStream`

- Extends: `<net.Socket>`

Represents the readable side of a TTY. In normal circumstances `process.stdin` will be the only `tty.ReadStream` instance in a Node.js process and there should be no reason to create additional instances.

`readStream.isRaw`

A `boolean` that is `true` if the TTY is currently configured to operate as a raw device. Defaults to `false`.

readStream.isTTY

A boolean that is always true for `tty.ReadStream` instances.

readStream.setRawMode(mode)

- `mode` `<boolean>` If `true`, configures the `tty.ReadStream` to operate as a raw device. If `false`, configures the `tty.ReadStream` to operate in its default mode. The `readStream.isRaw` property will be set to the resulting mode.
- Returns: `<this>` The read stream instance.

Allows configuration of `tty.ReadStream` so that it operates as a raw device.

When in raw mode, input is always available character-by-character, not including modifiers. Additionally, all special processing of characters by the terminal is disabled, including echoing input characters. `ctrl + c` will no longer cause a `SIGINT` when in this mode.

Class: tty.WriteStream

- Extends: `<net.Socket>`

Represents the writable side of a TTY. In normal circumstances, `process.stdout` and `process.stderr` will be the only `tty.WriteStream` instances created for a Node.js process and there should be no reason to create additional instances.

Event: 'resize'

The `'resize'` event is emitted whenever either of the `writeStream.columns` or `writeStream.rows` properties have changed. No arguments are passed to the listener callback when called.

```
process.stdout.on('resize', () => {
  console.log('screen size has changed!');
  console.log(`${process.stdout.columns}x${process.stdout.rows}`);
});
```

writeStream.clearLine(dir[, callback])

- `dir` `<number>`
 - `-1`: to the left from cursor
 - `1`: to the right from cursor
 - `0`: the entire line
- `callback` `<Function>` Invoked once the operation completes.
- Returns: `<boolean>` `false` if the stream wishes for the calling code to wait for the `'drain'` event to be emitted before continuing to write additional data; otherwise `true`.

`writeStream.clearLine()` clears the current line of this `WriteStream` in a direction identified by `dir`.

writeStream.clearScreenDown([callback])

- `callback` `<Function>` Invoked once the operation completes.
- Returns: `<boolean>` `false` if the stream wishes for the calling code to wait for the `'drain'` event to be emitted before continuing to write additional data; otherwise `true`.

`writeStream.clearScreenDown()` clears this `WriteStream` from the current cursor down.

writeStream.columns

A `number` specifying the number of columns the TTY currently has. This property is updated whenever the `'resize'` event is emitted.

`writeStream.cursorTo(x[, y][, callback])`

- `x` `<number>`
- `y` `<number>`
- `callback` `<Function>` Invoked once the operation completes.
- Returns: `<boolean>` `false` if the stream wishes for the calling code to wait for the `'drain'` event to be emitted before continuing to write additional data; otherwise `true`.

`writeStream.cursorTo()` moves this `WriteStream`'s cursor to the specified position.

`writeStream.getColorDepth([env])`

- `env` `<Object>` An object containing the environment variables to check. This enables simulating the usage of a specific terminal.
Default: `process.env`.
- Returns: `<number>`

Returns:

- `1` for 2,
- `4` for 16,
- `8` for 256,
- `24` for 16,777,216

colors supported.

Use this to determine what colors the terminal supports. Due to the nature of colors in terminals it is possible to either have false positives or false negatives. It depends on process information and the environment variables that may lie about what terminal is used. It is possible to pass in an `env` object to simulate the usage of a specific terminal. This can be useful to check how specific environment settings behave.

To enforce a specific color support, use one of the below environment settings.

- 2 colors: `FORCE_COLOR = 0` (Disables colors)
- 16 colors: `FORCE_COLOR = 1`
- 256 colors: `FORCE_COLOR = 2`
- 16,777,216 colors: `FORCE_COLOR = 3`

Disabling color support is also possible by using the `NO_COLOR` and `NODE_DISABLE_COLORS` environment variables.

`writeStream.getWindowSize()`

- Returns: `<number[]>`

`writeStream.getWindowSize()` returns the size of the TTY corresponding to this `WriteStream`. The array is of the type `[numColumns, numRows]` where `numColumns` and `numRows` represent the number of columns and rows in the corresponding TTY.

`writeStream.hasColors([count][, env])`

- `count` `<integer>` The number of colors that are requested (minimum 2). **Default:** 16.
- `env` `<Object>` An object containing the environment variables to check. This enables simulating the usage of a specific terminal.
Default: `process.env`.
- Returns: `<boolean>`

Returns `true` if the `writeStream` supports at least as many colors as provided in `count`. Minimum support is 2 (black and white).

This has the same false positives and negatives as described in `writeStream.getColorDepth()`.

```
process.stdout.hasColors();
// Returns true or false depending on if `stdout` supports at least 16 colors.
process.stdout.hasColors(256);
// Returns true or false depending on if `stdout` supports at least 256 colors.
process.stdout.hasColors({ TMUX: '1' });
// Returns true.
process.stdout.hasColors(2 ** 24, { TMUX: '1' });
// Returns false (the environment setting pretends to support 2 ** 8 colors).
```

`writeStream.isTTY`

A `boolean` that is always `true`.

`writeStream.moveCursor(dx, dy[, callback])`

- `dx` `<number>`
- `dy` `<number>`
- `callback` `<Function>` Invoked once the operation completes.
- Returns: `<boolean>` `false` if the stream wishes for the calling code to wait for the `'drain'` event to be emitted before continuing to write additional data; otherwise `true`.

`writeStream.moveCursor()` moves this `WriteStream`'s cursor *relative* to its current position.

`writeStream.rows`

A `number` specifying the number of rows the TTY currently has. This property is updated whenever the `'resize'` event is emitted.

`tty.isatty(fd)`

- `fd` `<number>` A numeric file descriptor
- Returns: `<boolean>`

The `tty.isatty()` method returns `true` if the given `fd` is associated with a TTY and `false` if it is not, including whenever `fd` is not a non-negative integer.

UDP/datagram sockets

Stability: 2 - Stable

Source Code: [lib/dgram.js](#)

The `dgram` module provides an implementation of UDP datagram sockets.

```
import dgram from 'dgram';

const server = dgram.createSocket('udp4');
```



```

server.on('error', (err) => {
  console.log(`server error:\n${err.stack}`);
  server.close();
});

server.on('message', (msg, rinfo) => {
  console.log(`server got: ${msg} from ${rinfo.address}:${rinfo.port}`);
});

server.on('listening', () => {
  const address = server.address();
  console.log(`server listening ${address.address}:${address.port}`);
});

server.bind(41234);
// Prints: server listening 0.0.0.0:41234
const dgram = require('dgram');
const server = dgram.createSocket('udp4');

server.on('error', (err) => {
  console.log(`server error:\n${err.stack}`);
  server.close();
});

server.on('message', (msg, rinfo) => {
  console.log(`server got: ${msg} from ${rinfo.address}:${rinfo.port}`);
});

server.on('listening', () => {
  const address = server.address();
  console.log(`server listening ${address.address}:${address.port}`);
});

server.bind(41234);
// Prints: server listening 0.0.0.0:41234

```

Class: `dgram.Socket`

- Extends: `<EventEmitter>`

Encapsulates the datagram functionality.

New instances of `dgram.Socket` are created using `dgram.createSocket()`. The `new` keyword is not to be used to create `dgram.Socket` instances.

Event: `'close'`

The `'close'` event is emitted after a socket is closed with `close()`. Once triggered, no new `'message'` events will be emitted on this socket.

Event: `'connect'`

The `'connect'` event is emitted after a socket is associated to a remote address as a result of a successful `connect()` call.

Event: 'error'

- `exception` `<Error>`

The 'error' event is emitted whenever any error occurs. The event handler function is passed a single `Error` object.

Event: 'listening'

The 'listening' event is emitted once the `dgram.Socket` is addressable and can receive data. This happens either explicitly with `socket.bind()` or implicitly the first time data is sent using `socket.send()`. Until the `dgram.Socket` is listening, the underlying system resources do not exist and calls such as `socket.address()` and `socket.setTTL()` will fail.

Event: 'message'

The 'message' event is emitted when a new datagram is available on a socket. The event handler function is passed two arguments: `msg` and `rinfo`.

- `msg` `<Buffer>` The message.
- `rinfo` `<Object>` Remote address information.
 - `address` `<string>` The sender address.
 - `family` `<string>` The address family ('IPv4' or 'IPv6').
 - `port` `<number>` The sender port.
 - `size` `<number>` The message size.

If the source address of the incoming packet is an IPv6 link-local address, the interface name is added to the `address`. For example, a packet received on the `en0` interface might have the address field set to `'fe80::2618:1234:ab11:3b9c%en0'`, where `'%en0'` is the interface name as a zone ID suffix.

`socket.addMembership(multicastAddress[, multicastInterface])`

- `multicastAddress` `<string>`
- `multicastInterface` `<string>`

Tells the kernel to join a multicast group at the given `multicastAddress` and `multicastInterface` using the `IP_ADD_MEMBERSHIP` socket option. If the `multicastInterface` argument is not specified, the operating system will choose one interface and will add membership to it. To add membership to every available interface, call `addMembership` multiple times, once per interface.

When called on an unbound socket, this method will implicitly bind to a random port, listening on all interfaces.

When sharing a UDP socket across multiple `cluster` workers, the `socket.addMembership()` function must be called only once or an `EADDRINUSE` error will occur:

```
import cluster from 'cluster';
import dgram from 'dgram';

if (cluster.isPrimary) {
  cluster.fork(); // Works ok.
  cluster.fork(); // Fails with EADDRINUSE.
} else {
  const s = dgram.createSocket('udp4');
  s.bind(1234, () => {
    s.addMembership('224.0.0.114');
  });
}const cluster = require('cluster');
```

```
const dgram = require('dgram');

if (cluster.isPrimary) {
  cluster.fork(); // Works ok.
  cluster.fork(); // Fails with EADDRINUSE.
} else {
  const s = dgram.createSocket('udp4');
  s.bind(1234, () => {
    s.addMembership('224.0.0.114');
  });
}
```

socket.addSourceSpecificMembership(sourceAddress, groupAddress[, multicastInterface])

- `sourceAddress` `<string>`
- `groupAddress` `<string>`
- `multicastInterface` `<string>`

Tells the kernel to join a source-specific multicast channel at the given `sourceAddress` and `groupAddress`, using the `multicastInterface` with the `IP_ADD_SOURCE_MEMBERSHIP` socket option. If the `multicastInterface` argument is not specified, the operating system will choose one interface and will add membership to it. To add membership to every available interface, call `socket.addSourceSpecificMembership()` multiple times, once per interface.

When called on an unbound socket, this method will implicitly bind to a random port, listening on all interfaces.

socket.address()

- Returns: `<Object>`

Returns an object containing the address information for a socket. For UDP sockets, this object will contain `address`, `family` and `port` properties.

This method throws `EBADF` if called on an unbound socket.

socket.bind([port][, address][, callback])

- `port` `<integer>`
- `address` `<string>`
- `callback` `<Function>` with no parameters. Called when binding is complete.

For UDP sockets, causes the `dgram.Socket` to listen for datagram messages on a named `port` and optional `address`. If `port` is not specified or is `0`, the operating system will attempt to bind to a random port. If `address` is not specified, the operating system will attempt to listen on all addresses. Once binding is complete, a `'listening'` event is emitted and the optional `callback` function is called.

Specifying both a `'listening'` event listener and passing a `callback` to the `socket.bind()` method is not harmful but not very useful.

A bound datagram socket keeps the Node.js process running to receive datagram messages.

If binding fails, an `'error'` event is generated. In rare case (e.g. attempting to bind with a closed socket), an `Error` may be thrown.

Example of a UDP server listening on port 41234:

```

import dgram from 'dgram';

const server = dgram.createSocket('udp4');

server.on('error', (err) => {
  console.log(`server error:\n${err.stack}`);
  server.close();
});

server.on('message', (msg, rinfo) => {
  console.log(`server got: ${msg} from ${rinfo.address}:${rinfo.port}`);
});

server.on('listening', () => {
  const address = server.address();
  console.log(`server listening ${address.address}:${address.port}`);
});

server.bind(41234);
// Prints: server listening 0.0.0.0:41234
const dgram = require('dgram');
const server = dgram.createSocket('udp4');

server.on('error', (err) => {
  console.log(`server error:\n${err.stack}`);
  server.close();
});

server.on('message', (msg, rinfo) => {
  console.log(`server got: ${msg} from ${rinfo.address}:${rinfo.port}`);
});

server.on('listening', () => {
  const address = server.address();
  console.log(`server listening ${address.address}:${address.port}`);
});

server.bind(41234);
// Prints: server listening 0.0.0.0:41234

```

socket.bind(options[, callback])

- **options** `<Object>` Required. Supports the following properties:
 - **port** `<integer>`
 - **address** `<string>`
 - **exclusive** `<boolean>`
 - **fd** `<integer>`
- **callback** `<Function>`

For UDP sockets, causes the `dgram.Socket` to listen for datagram messages on a named `port` and optional `address` that are passed as properties of an `options` object passed as the first argument. If `port` is not specified or is `0`, the operating system will attempt to bind to a

random port. If `address` is not specified, the operating system will attempt to listen on all addresses. Once binding is complete, a `'listening'` event is emitted and the optional `callback` function is called.

The `options` object may contain a `fd` property. When a `fd` greater than `0` is set, it will wrap around an existing socket with the given file descriptor. In this case, the properties of `port` and `address` will be ignored.

Specifying both a `'listening'` event listener and passing a `callback` to the `socket.bind()` method is not harmful but not very useful.

The `options` object may contain an additional `exclusive` property that is used when using `dgram.Socket` objects with the `cluster` module. When `exclusive` is set to `false` (the default), cluster workers will use the same underlying socket handle allowing connection handling duties to be shared. When `exclusive` is `true`, however, the handle is not shared and attempted port sharing results in an error.

A bound datagram socket keeps the Node.js process running to receive datagram messages.

If binding fails, an `'error'` event is generated. In rare case (e.g. attempting to bind with a closed socket), an `Error` may be thrown.

An example socket listening on an exclusive port is shown below.

```
socket.bind({
  address: 'localhost',
  port: 8000,
  exclusive: true
});
```

socket.close([callback])

- `callback` `<Function>` Called when the socket has been closed.

Close the underlying socket and stop listening for data on it. If a callback is provided, it is added as a listener for the `'close'` event.

socket.connect(port[, address][, callback])

- `port` `<integer>`
- `address` `<string>`
- `callback` `<Function>` Called when the connection is completed or on error.

Associates the `dgram.Socket` to a remote address and port. Every message sent by this handle is automatically sent to that destination. Also, the socket will only receive messages from that remote peer. Trying to call `connect()` on an already connected socket will result in an `ERR_SOCKET_DGRAM_IS_CONNECTED` exception. If `address` is not provided, `'127.0.0.1'` (for `udp4` sockets) or `:::1` (for `udp6` sockets) will be used by default. Once the connection is complete, a `'connect'` event is emitted and the optional `callback` function is called. In case of failure, the `callback` is called or, failing this, an `'error'` event is emitted.

socket.disconnect()

A synchronous function that disassociates a connected `dgram.Socket` from its remote address. Trying to call `disconnect()` on an unbound or already disconnected socket will result in an `ERR_SOCKET_DGRAM_NOT_CONNECTED` exception.

socket.dropMembership(multicastAddress[, multicastInterface])

- `multicastAddress` `<string>`
- `multicastInterface` `<string>`

Instructs the kernel to leave a multicast group at `multicastAddress` using the `IP_DROP_MEMBERSHIP` socket option. This method is automatically called by the kernel when the socket is closed or the process terminates, so most apps will never have reason to call this.

If `multicastInterface` is not specified, the operating system will attempt to drop membership on all valid interfaces.

`socket.dropSourceSpecificMembership(sourceAddress, groupAddress[, multicastInterface])`

- `sourceAddress` `<string>`
- `groupAddress` `<string>`
- `multicastInterface` `<string>`

Instructs the kernel to leave a source-specific multicast channel at the given `sourceAddress` and `groupAddress` using the `IP_DROP_SOURCE_MEMBERSHIP` socket option. This method is automatically called by the kernel when the socket is closed or the process terminates, so most apps will never have reason to call this.

If `multicastInterface` is not specified, the operating system will attempt to drop membership on all valid interfaces.

`socket.getRecvBufferSize()`

- Returns: `<number>` the `SO_RCVBUF` socket receive buffer size in bytes.

This method throws `ERR_SOCKET_BUFFER_SIZE` if called on an unbound socket.

`socket.getSendBufferSize()`

- Returns: `<number>` the `SO_SNDBUF` socket send buffer size in bytes.

This method throws `ERR_SOCKET_BUFFER_SIZE` if called on an unbound socket.

`socket.ref()`

- Returns: `<dgram.Socket>`

By default, binding a socket will cause it to block the Node.js process from exiting as long as the socket is open. The `socket.unref()` method can be used to exclude the socket from the reference counting that keeps the Node.js process active. The `socket.ref()` method adds the socket back to the reference counting and restores the default behavior.

Calling `socket.ref()` multiples times will have no additional effect.

The `socket.ref()` method returns a reference to the socket so calls can be chained.

`socket.remoteAddress()`

- Returns: `<Object>`

Returns an object containing the `address`, `family`, and `port` of the remote endpoint. This method throws an `ERR_SOCKET_DGRAM_NOT_CONNECTED` exception if the socket is not connected.

`socket.send(msg[, offset, length][, port][, address][, callback])`

- `msg` `<Buffer>` | `<TypedArray>` | `<DataView>` | `<string>` | `<Array>` Message to be sent.
- `offset` `<integer>` Offset in the buffer where the message starts.
- `length` `<integer>` Number of bytes in the message.
- `port` `<integer>` Destination port.
- `address` `<string>` Destination host name or IP address.
- `callback` `<Function>` Called when the message has been sent.

Broadcasts a datagram on the socket. For connectionless sockets, the destination `port` and `address` must be specified. Connected sockets, on the other hand, will use their associated remote endpoint, so the `port` and `address` arguments must not be set.

The `msg` argument contains the message to be sent. Depending on its type, different behavior can apply. If `msg` is a `Buffer`, any `TypedArray` or a `DataView`, the `offset` and `length` specify the offset within the `Buffer` where the message begins and the number of bytes in the message, respectively. If `msg` is a `String`, then it is automatically converted to a `Buffer` with `'utf8'` encoding. With messages that contain multi-byte characters, `offset` and `length` will be calculated with respect to `byte length` and not the character position. If `msg` is an array, `offset` and `length` must not be specified.

The `address` argument is a string. If the value of `address` is a host name, DNS will be used to resolve the address of the host. If `address` is not provided or otherwise falsy, `'127.0.0.1'` (for `udp4` sockets) or `'::1'` (for `udp6` sockets) will be used by default.

If the socket has not been previously bound with a call to `bind`, the socket is assigned a random port number and is bound to the "all interfaces" address (`'0.0.0.0'` for `udp4` sockets, `'::0'` for `udp6` sockets.)

An optional `callback` function may be specified to as a way of reporting DNS errors or for determining when it is safe to reuse the `buf` object. DNS lookups delay the time to send for at least one tick of the Node.js event loop.

The only way to know for sure that the datagram has been sent is by using a `callback`. If an error occurs and a `callback` is given, the error will be passed as the first argument to the `callback`. If a `callback` is not given, the error is emitted as an `'error'` event on the `socket` object.

Offset and length are optional but both *must* be set if either are used. They are supported only when the first argument is a `Buffer`, a `TypedArray`, or a `DataView`.

This method throws `ERR_SOCKET_BAD_PORT` if called on an unbound socket.

Example of sending a UDP packet to a port on `localhost`;

```
import dgram from 'dgram';
import { Buffer } from 'buffer';

const message = Buffer.from('Some bytes');
const client = dgram.createSocket('udp4');
client.send(message, 41234, 'localhost', (err) => {
  client.close();
});const dgram = require('dgram');
const { Buffer } = require('buffer');

const message = Buffer.from('Some bytes');
const client = dgram.createSocket('udp4');
client.send(message, 41234, 'localhost', (err) => {
  client.close();
});
```

Example of sending a UDP packet composed of multiple buffers to a port on `127.0.0.1`;

```
import dgram from 'dgram';
import { Buffer } from 'buffer';

const buf1 = Buffer.from('Some ');
const buf2 = Buffer.from('bytes');
const client = dgram.createSocket('udp4');
client.send([buf1, buf2], 41234, (err) => {
  client.close();
});
```

```

});const dgram = require('dgram');
const { Buffer } = require('buffer');

const buf1 = Buffer.from('Some ');
const buf2 = Buffer.from('bytes');
const client = dgram.createSocket('udp4');
client.send([buf1, buf2], 41234, (err) => {
  client.close();
});

```

Sending multiple buffers might be faster or slower depending on the application and operating system. Run benchmarks to determine the optimal strategy on a case-by-case basis. Generally speaking, however, sending multiple buffers is faster.

Example of sending a UDP packet using a socket connected to a port on `localhost` :

```

import dgram from 'dgram';
import { Buffer } from 'buffer';

const message = Buffer.from('Some bytes');
const client = dgram.createSocket('udp4');
client.connect(41234, 'localhost', (err) => {
  client.send(message, (err) => {
    client.close();
  });
});const dgram = require('dgram');
const { Buffer } = require('buffer');

const message = Buffer.from('Some bytes');
const client = dgram.createSocket('udp4');
client.connect(41234, 'localhost', (err) => {
  client.send(message, (err) => {
    client.close();
  });
});

```

Note about UDP datagram size

The maximum size of an IPv4/v6 datagram depends on the `MTU` (Maximum Transmission Unit) and on the `Payload Length` field size.

- The `Payload Length` field is 16 bits wide, which means that a normal payload cannot exceed 64K octets including the internet header and data (65,507 bytes = 65,535 – 8 bytes UDP header – 20 bytes IP header); this is generally true for loopback interfaces, but such long datagram messages are impractical for most hosts and networks.
- The `MTU` is the largest size a given link layer technology can support for datagram messages. For any link, IPv4 mandates a minimum `MTU` of 68 octets, while the recommended `MTU` for IPv4 is 576 (typically recommended as the `MTU` for dial-up type applications), whether they arrive whole or in fragments.

For IPv6, the minimum `MTU` is 1280 octets. However, the mandatory minimum fragment reassembly buffer size is 1500 octets. The value of 68 octets is very small, since most current link layer technologies, like Ethernet, have a minimum `MTU` of 1500.

It is impossible to know in advance the MTU of each link through which a packet might travel. Sending a datagram greater than the receiver `MTU` will not work because the packet will get silently dropped without informing the source that the data did not reach its intended recipient.

socket.setBroadcast(flag)

- `flag` `<boolean>`

Sets or clears the `SO_BROADCAST` socket option. When set to `true`, UDP packets may be sent to a local interface's broadcast address.

This method throws `EBADF` if called on an unbound socket.

socket.setMulticastInterface(multicastInterface)

- `multicastInterface` `<string>`

All references to scope in this section are referring to *IPv6 Zone Indices*, which are defined by *RFC 4007*. In string form, an IP with a scope index is written as `'IP%scope'` where *scope* is an interface name or interface number.

Sets the default outgoing multicast interface of the socket to a chosen interface or back to system interface selection. The `multicastInterface` must be a valid string representation of an IP from the socket's family.

For IPv4 sockets, this should be the IP configured for the desired physical interface. All packets sent to multicast on the socket will be sent on the interface determined by the most recent successful use of this call.

For IPv6 sockets, `multicastInterface` should include a scope to indicate the interface as in the examples that follow. In IPv6, individual `send` calls can also use explicit scope in addresses, so only packets sent to a multicast address without specifying an explicit scope are affected by the most recent successful use of this call.

This method throws `EBADF` if called on an unbound socket.

Example: IPv6 outgoing multicast interface

On most systems, where scope format uses the interface name:

```
const socket = dgram.createSocket('udp6');

socket.bind(1234, () => {
  socket.setMulticastInterface('::eth1');
});
```

On Windows, where scope format uses an interface number:

```
const socket = dgram.createSocket('udp6');

socket.bind(1234, () => {
  socket.setMulticastInterface('::%2');
});
```

Example: IPv4 outgoing multicast interface

All systems use an IP of the host on the desired physical interface:

```
const socket = dgram.createSocket('udp4');

socket.bind(1234, () => {
  socket.setMulticastInterface('10.0.0.2');
});
```

Call results

A call on a socket that is not ready to send or no longer open may throw a *Not running Error* .

If `multicastInterface` can not be parsed into an IP then an *EINVAL System Error* is thrown.

On IPv4, if `multicastInterface` is a valid address but does not match any interface, or if the address does not match the family then a *System Error* such as `EADDRNOTAVAIL` or `EPROTONOSUP` is thrown.

On IPv6, most errors with specifying or omitting scope will result in the socket continuing to use (or returning to) the system's default interface selection.

A socket's address family's ANY address (IPv4 `'0.0.0.0'` or IPv6 `':: '`) can be used to return control of the sockets default outgoing interface to the system for future multicast packets.

socket.setMulticastLoopback(flag)

- `flag` *<boolean>*

Sets or clears the `IP_MULTICAST_LOOP` socket option. When set to `true`, multicast packets will also be received on the local interface.

This method throws `EBADF` if called on an unbound socket.

socket.setMulticastTTL(ttl)

- `ttl` *<integer>*

Sets the `IP_MULTICAST_TTL` socket option. While TTL generally stands for "Time to Live", in this context it specifies the number of IP hops that a packet is allowed to travel through, specifically for multicast traffic. Each router or gateway that forwards a packet decrements the TTL. If the TTL is decremented to 0 by a router, it will not be forwarded.

The `ttl` argument may be between 0 and 255. The default on most systems is `1`.

This method throws `EBADF` if called on an unbound socket.

socket.setRecvBufferSize(size)

- `size` *<integer>*

Sets the `SO_RCVBUF` socket option. Sets the maximum socket receive buffer in bytes.

This method throws `ERR_SOCKET_BUFFER_SIZE` if called on an unbound socket.

socket.setSendBufferSize(size)

- `size` *<integer>*

Sets the `SO_SNDBUF` socket option. Sets the maximum socket send buffer in bytes.

This method throws `ERR_SOCKET_BUFFER_SIZE` if called on an unbound socket.

socket.setTTL(ttl)

- `ttl` *<integer>*

Sets the `IP_TTL` socket option. While TTL generally stands for "Time to Live", in this context it specifies the number of IP hops that a packet is allowed to travel through. Each router or gateway that forwards a packet decrements the TTL. If the TTL is decremented to 0 by a router, it will not be forwarded. Changing TTL values is typically done for network probes or when multicasting.

The `ttl` argument may be between between 1 and 255. The default on most systems is 64.

This method throws `EBADF` if called on an unbound socket.

`socket.unref()`

- Returns: `<dgram.Socket>`

By default, binding a socket will cause it to block the Node.js process from exiting as long as the socket is open. The `socket.unref()` method can be used to exclude the socket from the reference counting that keeps the Node.js process active, allowing the process to exit even if the socket is still listening.

Calling `socket.unref()` multiple times will have no addition effect.

The `socket.unref()` method returns a reference to the socket so calls can be chained.

`dgram` module functions

`dgram.createSocket(options[, callback])`

- `options` `<Object>` Available options are:
 - `type` `<string>` The family of socket. Must be either `'udp4'` or `'udp6'`. Required.
 - `reuseAddr` `<boolean>` When `true` `socket.bind()` will reuse the address, even if another process has already bound a socket on it. **Default:** `false`.
 - `ipv6only` `<boolean>` Setting `ipv6only` to `true` will disable dual-stack support, i.e., binding to address `::` won't make `0.0.0.0` be bound. **Default:** `false`.
 - `recvBufferSize` `<number>` Sets the `SO_RCVBUF` socket value.
 - `sendBufferSize` `<number>` Sets the `SO_SNDBUF` socket value.
 - `lookup` `<Function>` Custom lookup function. **Default:** `dns.lookup()`.
 - `signal` `<AbortSignal>` An `AbortSignal` that may be used to close a socket.
- `callback` `<Function>` Attached as a listener for `'message'` events. Optional.
- Returns: `<dgram.Socket>`

Creates a `dgram.Socket` object. Once the socket is created, calling `socket.bind()` will instruct the socket to begin listening for datagram messages. When `address` and `port` are not passed to `socket.bind()` the method will bind the socket to the "all interfaces" address on a random port (it does the right thing for both `udp4` and `udp6` sockets). The bound address and port can be retrieved using `socket.address().address` and `socket.address().port`.

If the `signal` option is enabled, calling `.abort()` on the corresponding `AbortController` is similar to calling `.close()` on the socket:

```
const controller = new AbortController();
const { signal } = controller;
const server = dgram.createSocket({ type: 'udp4', signal });
server.on('message', (msg, rinfo) => {
  console.log(`server got: ${msg} from ${rinfo.address}:${rinfo.port}`);
});
// Later, when you want to close the server.
controller.abort();
```

`dgram.createSocket(type[, callback])`

- `type` `<string>` Either `'udp4'` or `'udp6'`.
- `callback` `<Function>` Attached as a listener to `'message'` events.

- Creates a `dgram.Socket` object of the specified `type`.

URL

Source Code: [lib/url.js](#)

```
import url from 'url';const url = require('url');
```

href									
protocol		auth			host		path		hash
					hostname	port	pathname	search	
								query	
"	https:	//	user	: pass	@ sub.example.com	: 8080	/p/a/t/h	? query=string	#hash
					hostname	port			
protocol		username	password		host				
origin					origin		pathname	search	hash
href									

```
(All spaces in the "" line should be ignored. They are purely for formatting.)
```

Parsing the URL string using the WHATWG API:

```
const myURL =  
  new URL('https://user:pass@sub.example.com:8080/p/a/t/h?query=string#hash');
```

Parsing the URL string using the Legacy API:

```
import url from 'url';  
const myURL =  
  url.parse('https://user:pass@sub.example.com:8080/p/a/t/h?query=string#hash');const url = require('url');  
const myURL =  
  url.parse('https://user:pass@sub.example.com:8080/p/a/t/h?query=string#hash');
```

Constructing a URL from component parts and getting the constructed string

It is possible to construct a WHATWG URL from component parts using either the property setters or a template literal string:

```
const myURL = new URL('https://example.org');  
myURL.pathname = '/a/b/c';  
myURL.search = '?d=e';  
myURL.hash = '#fgh';
```

```
const pathname = '/a/b/c';  
const search = '?d=e';  
const hash = '#fgh';  
const myURL = new URL(`https://example.org${pathname}${search}${hash}`);
```

To get the constructed URL string, use the `href` property accessor:

```
console.log(myURL.href);
```

The WHATWG URL API

Class: `URL`

Browser-compatible `URL` class, implemented by following the WHATWG URL Standard. [Examples of parsed URLs](#) may be found in the Standard itself. The `URL` class is also available on the global object.

In accordance with browser conventions, all properties of `URL` objects are implemented as getters and setters on the class prototype, rather than as data properties on the object itself. Thus, unlike `legacy urlObject`s, using the `delete` keyword on any properties of `URL` objects (e.g. `delete myURL.protocol`, `delete myURL.pathname`, etc) has no effect but will still return `true`.

`new URL(input[, base])`

- `input` `<string>` The absolute or relative input URL to parse. If `input` is relative, then `base` is required. If `input` is absolute, the `base` is ignored.
- `base` `<string> | <URL>` The base URL to resolve against if the `input` is not absolute.

Creates a new `URL` object by parsing the `input` relative to the `base`. If `base` is passed as a string, it will be parsed equivalent to `new URL(base)`.

```
const myURL = new URL('/foo', 'https://example.org/');
// https://example.org/foo
```

The `URL` constructor is accessible as a property on the global object. It can also be imported from the built-in `url` module:

```
import { URL } from 'url';
console.log(URL === globalThis.URL); // Prints 'true'.
console.log(URL === require('url').URL); // Prints 'true'.
```

A `TypeError` will be thrown if the `input` or `base` are not valid URLs. Note that an effort will be made to coerce the given values into strings. For instance:

```
const myURL = new URL({ toString: () => 'https://example.org/' });
// https://example.org/
```

Unicode characters appearing within the host name of `input` will be automatically converted to ASCII using the `Punycode` algorithm.

```
const myURL = new URL('https://測試');
// https://xn--g6w251d/
```

This feature is only available if the `node` executable was compiled with `ICU` enabled. If not, the domain names are passed through unchanged.

In cases where it is not known in advance if `input` is an absolute URL and a `base` is provided, it is advised to validate that the `origin` of the `URL` object is what is expected.

```
let myURL = new URL('http://Example.com/', 'https://example.org/');
// http://example.com/

myURL = new URL('https://Example.com/', 'https://example.org/');
// https://example.com/

myURL = new URL('foo://Example.com/', 'https://example.org/');
// foo://Example.com/

myURL = new URL('http:Example.com/', 'https://example.org/');
// http://example.com/

myURL = new URL('https:Example.com/', 'https://example.org/');
// https://example.org/Example.com/

myURL = new URL('foo:Example.com/', 'https://example.org/');
// foo:Example.com/
```

url.hash

- `<string>`

Gets and sets the fragment portion of the URL.

```
const myURL = new URL('https://example.org/foo#bar');
console.log(myURL.hash);
// Prints #bar

myURL.hash = 'baz';
console.log(myURL.href);
// Prints https://example.org/foo#baz
```

Invalid URL characters included in the value assigned to the `hash` property are `percent-encoded`. The selection of which characters to percent-encode may vary somewhat from what the `url.parse()` and `url.format()` methods would produce.

url.host

- `<string>`

Gets and sets the host portion of the URL.

```
const myURL = new URL('https://example.org:81/foo');
console.log(myURL.host);
// Prints example.org:81

myURL.host = 'example.com:82';
console.log(myURL.href);
// Prints https://example.com:82/foo
```

Invalid host values assigned to the `host` property are ignored.

url.hostname

- `<string>`

Gets and sets the host name portion of the URL. The key difference between `url.host` and `url.hostname` is that `url.hostname` does not include the port.

```
const myURL = new URL('https://example.org:81/foo');
console.log(myURL.hostname);
// Prints example.org

// Setting the hostname does not change the port
myURL.hostname = 'example.com:82';
console.log(myURL.href);
// Prints https://example.com:81/foo

// Use myURL.host to change the hostname and port
myURL.host = 'example.org:82';
console.log(myURL.href);
// Prints https://example.org:82/foo
```

Invalid host name values assigned to the `hostname` property are ignored.

`url.href`

- `<string>`

Gets and sets the serialized URL.

```
const myURL = new URL('https://example.org/foo');
console.log(myURL.href);
// Prints https://example.org/foo

myURL.href = 'https://example.com/bar';
console.log(myURL.href);
// Prints https://example.com/bar
```

Getting the value of the `href` property is equivalent to calling `url.toString()`.

Setting the value of this property to a new value is equivalent to creating a new `URL` object using `new URL(value)`. Each of the `URL` object's properties will be modified.

If the value assigned to the `href` property is not a valid URL, a `TypeError` will be thrown.

`url.origin`

- `<string>`

Gets the read-only serialization of the URL's origin.

```
const myURL = new URL('https://example.org/foo/bar?baz');
console.log(myURL.origin);
// Prints https://example.org
```

```
const idnURL = new URL('https://測試');
console.log(idnURL.origin);
// Prints https://xn--g6w251d

console.log(idnURL.hostname);
// Prints xn--g6w251d
```

`url.password`

- `<string>`

Gets and sets the password portion of the URL.

```
const myURL = new URL('https://abc:xyz@example.com');
console.log(myURL.password);
// Prints xyz

myURL.password = '123';
```



```
console.log(myURL.href);  
// Prints https://abc:123@example.com
```

Invalid URL characters included in the value assigned to the `password` property are **percent-encoded**. The selection of which characters to percent-encode may vary somewhat from what the `url.parse()` and `url.format()` methods would produce.

url.pathname

- `<string>`

Gets and sets the path portion of the URL.

```
const myURL = new URL('https://example.org/abc/xyz?123');  
console.log(myURL.pathname);  
// Prints /abc/xyz  
  
myURL.pathname = '/abcdef';  
console.log(myURL.href);  
// Prints https://example.org/abcdef?123
```

Invalid URL characters included in the value assigned to the `pathname` property are **percent-encoded**. The selection of which characters to percent-encode may vary somewhat from what the `url.parse()` and `url.format()` methods would produce.

url.port

- `<string>`

Gets and sets the port portion of the URL.

The port value may be a number or a string containing a number in the range `0` to `65535` (inclusive). Setting the value to the default port of the `URL` objects given `protocol` will result in the `port` value becoming the empty string (`''`).

The port value can be an empty string in which case the port depends on the protocol/scheme:

protocol	port
"ftp"	21
"file"	
"http"	80
"https"	443
"ws"	80
"wss"	443

Upon assigning a value to the port, the value will first be converted to a string using `.toString()`.

If that string is invalid but it begins with a number, the leading number is assigned to `port`. If the number lies outside the range denoted above, it is ignored.

```
const myURL = new URL('https://example.org:8888');  
console.log(myURL.port);  
// Prints 8888
```

```

// Default ports are automatically transformed to the empty string
// (HTTPS protocol's default port is 443)
myURL.port = '443';
console.log(myURL.port);
// Prints the empty string
console.log(myURL.href);
// Prints https://example.org/

myURL.port = 1234;
console.log(myURL.port);
// Prints 1234
console.log(myURL.href);
// Prints https://example.org:1234/

// Completely invalid port strings are ignored
myURL.port = 'abcd';
console.log(myURL.port);
// Prints 1234

// Leading numbers are treated as a port number
myURL.port = '5678abcd';
console.log(myURL.port);
// Prints 5678

// Non-integers are truncated
myURL.port = 1234.5678;
console.log(myURL.port);
// Prints 1234

// Out-of-range numbers which are not represented in scientific notation
// will be ignored.
myURL.port = 1e10; // 10000000000, will be range-checked as described below
console.log(myURL.port);
// Prints 1234

```

Numbers which contain a decimal point, such as floating-point numbers or numbers in scientific notation, are not an exception to this rule. Leading numbers up to the decimal point will be set as the URL's port, assuming they are valid:

```

myURL.port = 4.567e21;
console.log(myURL.port);
// Prints 4 (because it is the leading number in the string '4.567e21')

```

url.protocol

- `<string>`

Gets and sets the protocol portion of the URL.

```

const myURL = new URL('https://example.org');
console.log(myURL.protocol);

```

```
// Prints https:

myURL.protocol = 'ftp';
console.log(myURL.href);
// Prints ftp://example.org/
```

Invalid URL protocol values assigned to the `protocol` property are ignored.

Special schemes

The [WHATWG URL Standard](#) considers a handful of URL protocol schemes to be *special* in terms of how they are parsed and serialized. When a URL is parsed using one of these special protocols, the `url.protocol` property may be changed to another special protocol but cannot be changed to a non-special protocol, and vice versa.

For instance, changing from `http` to `https` works:

```
const u = new URL('http://example.org');
u.protocol = 'https';
console.log(u.href);
// https://example.org
```

However, changing from `http` to a hypothetical `fish` protocol does not because the new protocol is not special.

```
const u = new URL('http://example.org');
u.protocol = 'fish';
console.log(u.href);
// http://example.org
```

Likewise, changing from a non-special protocol to a special protocol is also not permitted:

```
const u = new URL('fish://example.org');
u.protocol = 'http';
console.log(u.href);
// fish://example.org
```

According to the WHATWG URL Standard, special protocol schemes are `ftp`, `file`, `http`, `https`, `ws`, and `wss`.

url.search

- `<string>`

Gets and sets the serialized query portion of the URL.

```
const myURL = new URL('https://example.org/abc?123');
console.log(myURL.search);
// Prints ?123

myURL.search = 'abc=xyz';
console.log(myURL.href);
// Prints https://example.org/abc?abc=xyz
```

Any invalid URL characters appearing in the value assigned the `search` property will be [percent-encoded](#). The selection of which characters to percent-encode may vary somewhat from what the `url.parse()` and `url.format()` methods would produce.

`url.searchParams`

- `<URLSearchParams>`

Gets the `URLSearchParams` object representing the query parameters of the URL. This property is read-only but the `URLSearchParams` object it provides can be used to mutate the URL instance; to replace the entirety of query parameters of the URL, use the `url.search` setter. See [URLSearchParams](#) documentation for details.

Use care when using `.searchParams` to modify the `URL` because, per the WHATWG specification, the `URLSearchParams` object uses different rules to determine which characters to percent-encode. For instance, the `URL` object will not percent encode the ASCII tilde (~) character, while `URLSearchParams` will always encode it:

```
const myUrl = new URL('https://example.org/abc?foo=~bar');

console.log(myUrl.search); // prints ?foo=~bar

// Modify the URL via searchParams...
myUrl.searchParams.sort();

console.log(myUrl.search); // prints ?foo=%7Ebar
```

`url.username`

- `<string>`

Gets and sets the username portion of the URL.

```
const myURL = new URL('https://abc:xyz@example.com');
console.log(myURL.username);
// Prints abc

myURL.username = '123';
console.log(myURL.href);
// Prints https://123:xyz@example.com/
```

Any invalid URL characters appearing in the value assigned the `username` property will be [percent-encoded](#). The selection of which characters to percent-encode may vary somewhat from what the `url.parse()` and `url.format()` methods would produce.

`url.toString()`

- Returns: `<string>`

The `toString()` method on the `URL` object returns the serialized URL. The value returned is equivalent to that of `url.href` and `url.toJSON()`.

`url.toJSON()`

- Returns: `<string>`

The `toJSON()` method on the `URL` object returns the serialized URL. The value returned is equivalent to that of `url.href` and `url.toString()`.

This method is automatically called when an `URL` object is serialized with `JSON.stringify()`.

```
const myURLs = [
  new URL('https://www.example.com'),
  new URL('https://test.example.org'),
];
console.log(JSON.stringify(myURLs));
// Prints ["https://www.example.com/", "https://test.example.org/"]
```

URL.createObjectURL(blob)

Stability: 1 - Experimental

- `blob` `<Blob>`
- Returns: `<string>`

Creates a `'blob:nodata: ...'` URL string that represents the given `<Blob>` object and can be used to retrieve the `Blob` later.

```
const {
  Blob,
  resolveObjectURL,
} = require('buffer');

const blob = new Blob(['hello']);
const id = URL.createObjectURL(blob);

// later...

const otherBlob = resolveObjectURL(id);
console.log(otherBlob.size);
```

The data stored by the registered `<Blob>` will be retained in memory until `URL.revokeObjectURL()` is called to remove it.

`Blob` objects are registered within the current thread. If using Worker Threads, `Blob` objects registered within one Worker will not be available to other workers or the main thread.

URL.revokeObjectURL(id)

Stability: 1 - Experimental

- `id` `<string>` A `'blob:nodata: ...'` URL string returned by a prior call to `URL.createObjectURL()`.

Removes the stored `<Blob>` identified by the given ID.

Class: URLSearchParams

The `URLSearchParams` API provides read and write access to the query of a `URL`. The `URLSearchParams` class can also be used standalone with one of the four following constructors. The `URLSearchParams` class is also available on the global object.

The WHATWG `URLSearchParams` interface and the `querystring` module have similar purpose, but the purpose of the `querystring` module is more general, as it allows the customization of delimiter characters (`&` and `=`). On the other hand, this API is designed purely for URL query strings.

```
const myURL = new URL('https://example.org/?abc=123');
console.log(myURL.searchParams.get('abc'));
// Prints 123

myURL.searchParams.append('abc', 'xyz');
console.log(myURL.href);
// Prints https://example.org/?abc=123&abc=xyz

myURL.searchParams.delete('abc');
myURL.searchParams.set('a', 'b');
console.log(myURL.href);
// Prints https://example.org/?a=b

const newSearchParams = new URLSearchParams(myURL.searchParams);
// The above is equivalent to
// const newSearchParams = new URLSearchParams(myURL.search);

newSearchParams.append('a', 'c');
console.log(myURL.href);
// Prints https://example.org/?a=b
console.log(newSearchParams.toString());
// Prints a=b&a=c

// newSearchParams.toString() is implicitly called
myURL.search = newSearchParams;
console.log(myURL.href);
// Prints https://example.org/?a=b&a=c
newSearchParams.delete('a');
console.log(myURL.href);
// Prints https://example.org/?a=b&a=c
```

new URLSearchParams()

Instantiate a new empty `URLSearchParams` object.

new URLSearchParams(string)

- `string` `<string>` A query string

Parse the `string` as a query string, and use it to instantiate a new `URLSearchParams` object. A leading `'?'`, if present, is ignored.

```
let params;

params = new URLSearchParams('user=abc&query=xyz');
console.log(params.get('user'));
// Prints 'abc'
console.log(params.toString());
```

```
// Prints 'user=abc&query=xyz'

params = new URLSearchParams('?user=abc&query=xyz');
console.log(params.toString());
// Prints 'user=abc&query=xyz'
```

new URLSearchParams(obj)

- `obj` `<Object>` An object representing a collection of key-value pairs

Instantiate a new `URLSearchParams` object with a query hash map. The key and value of each property of `obj` are always coerced to strings.

Unlike `querystring` module, duplicate keys in the form of array values are not allowed. Arrays are stringified using `array.toString()`, which simply joins all array elements with commas.

```
const params = new URLSearchParams({
  user: 'abc',
  query: ['first', 'second']
});
console.log(params.getAll('query'));
// Prints [ 'first,second' ]
console.log(params.toString());
// Prints 'user=abc&query=first%2Csecond'
```

new URLSearchParams(iterable)

- `iterable` `<Iterable>` An iterable object whose elements are key-value pairs

Instantiate a new `URLSearchParams` object with an iterable map in a way that is similar to `Map`'s constructor. `iterable` can be an `Array` or any iterable object. That means `iterable` can be another `URLSearchParams`, in which case the constructor will simply create a clone of the provided `URLSearchParams`. Elements of `iterable` are key-value pairs, and can themselves be any iterable object.

Duplicate keys are allowed.

```
let params;

// Using an array
params = new URLSearchParams([
  ['user', 'abc'],
  ['query', 'first'],
  ['query', 'second'],
]);
console.log(params.toString());
// Prints 'user=abc&query=first&query=second'

// Using a Map object
const map = new Map();
map.set('user', 'abc');
map.set('query', 'xyz');
params = new URLSearchParams(map);
console.log(params.toString());
// Prints 'user=abc&query=xyz'
```

```
// Using a generator function
function* getQueryPairs() {
  yield ['user', 'abc'];
  yield ['query', 'first'];
  yield ['query', 'second'];
}

params = new URLSearchParams(getQueryPairs());
console.log(params.toString());
// Prints 'user=abc&query=first&query=second'

// Each key-value pair must have exactly two elements
new URLSearchParams([
  ['user', 'abc', 'error'],
]);
// Throws TypeError [ERR_INVALID_TUPLE]:
//     Each query pair must be an iterable [name, value] tuple
```

urlSearchParams.append(name, value)

- `name` `<string>`
- `value` `<string>`

Append a new name-value pair to the query string.

urlSearchParams.delete(name)

- `name` `<string>`

Remove all name-value pairs whose name is `name`.

urlSearchParams.entries()

- Returns: `<Iterator>`

Returns an ES6 `Iterator` over each of the name-value pairs in the query. Each item of the iterator is a JavaScript `Array`. The first item of the `Array` is the `name`, the second item of the `Array` is the `value`.

Alias for `urlSearchParams[Symbol.iterator]()`.

urlSearchParams.forEach(fn[, thisArg])

- `fn` `<Function>` Invoked for each name-value pair in the query
- `thisArg` `<Object>` To be used as `this` value for when `fn` is called

Iterates over each name-value pair in the query and invokes the given function.

```
const myURL = new URL('https://example.org/?a=b&c=d');
myURL.searchParams.forEach((value, name, searchParams) => {
  console.log(name, value, myURL.searchParams === searchParams);
});
// Prints:
//   a b true
//   c d true
```


urlSearchParams.get(name)

- `name` `<string>`
- Returns: `<string>` or `null` if there is no name-value pair with the given `name`.

Returns the value of the first name-value pair whose name is `name`. If there are no such pairs, `null` is returned.

urlSearchParams.getAll(name)

- `name` `<string>`
- Returns: `<string[]>`

Returns the values of all name-value pairs whose name is `name`. If there are no such pairs, an empty array is returned.

urlSearchParams.has(name)

- `name` `<string>`
- Returns: `<boolean>`

Returns `true` if there is at least one name-value pair whose name is `name`.

urlSearchParams.keys()

- Returns: `<Iterator>`

Returns an ES6 `Iterator` over the names of each name-value pair.

```
const params = new URLSearchParams('foo=bar&foo=baz');
for (const name of params.keys()) {
  console.log(name);
}
// Prints:
//   foo
//   foo
```

urlSearchParams.set(name, value)

- `name` `<string>`
- `value` `<string>`

Sets the value in the `URLSearchParams` object associated with `name` to `value`. If there are any pre-existing name-value pairs whose names are `name`, set the first such pair's value to `value` and remove all others. If not, append the name-value pair to the query string.

```
const params = new URLSearchParams();
params.append('foo', 'bar');
params.append('foo', 'baz');
params.append('abc', 'def');
console.log(params.toString());
// Prints foo=bar&foo=baz&abc=def

params.set('foo', 'def');
params.set('xyz', 'opq');
console.log(params.toString());
// Prints foo=def&abc=def&xyz=opq
```

urlSearchParams.sort()

Sort all existing name-value pairs in-place by their names. Sorting is done with a [stable sorting algorithm](#), so relative order between name-value pairs with the same name is preserved.

This method can be used, in particular, to increase cache hits.

```
const params = new URLSearchParams('query[]=abc&type=search&query[]=123');
params.sort();
console.log(params.toString());
// Prints query%5B%5D=abc&query%5B%5D=123&type=search
```

urlSearchParams.toString()

- Returns: `<string>`

Returns the search parameters serialized as a string, with characters percent-encoded where necessary.

urlSearchParams.values()

- Returns: `<Iterator>`

Returns an ES6 `Iterator` over the values of each name-value pair.

urlSearchParams[Symbol.iterator]()

- Returns: `<Iterator>`

Returns an ES6 `Iterator` over each of the name-value pairs in the query string. Each item of the iterator is a JavaScript `Array`. The first item of the `Array` is the `name`, the second item of the `Array` is the `value`.

Alias for `urlSearchParams.entries()`.

```
const params = new URLSearchParams('foo=bar&xyz=baz');
for (const [name, value] of params) {
  console.log(name, value);
}
// Prints:
//   foo bar
//   xyz baz
```

url.domainToASCII(domain)

- `domain` `<string>`
- Returns: `<string>`

Returns the [Punycode](#) ASCII serialization of the `domain`. If `domain` is an invalid domain, the empty string is returned.

It performs the inverse operation to `url.domainToUnicode()`.

This feature is only available if the `node` executable was compiled with [ICU](#) enabled. If not, the domain names are passed through unchanged.

```
import url from 'url';

console.log(url.domainToASCII('español.com'));
```

```
// Prints xn--espaol-zwa.com
console.log(url.domainToASCII('中文.com'));
// Prints xn--fiq228c.com
console.log(url.domainToASCII('xn--iñvalid.com'));
// Prints an empty stringconst url = require('url');

console.log(url.domainToASCII('español.com'));
// Prints xn--espaol-zwa.com
console.log(url.domainToASCII('中文.com'));
// Prints xn--fiq228c.com
console.log(url.domainToASCII('xn--iñvalid.com'));
// Prints an empty string
```

url.domainToUnicode(domain)

- `domain` `<string>`
- Returns: `<string>`

Returns the Unicode serialization of the `domain`. If `domain` is an invalid domain, the empty string is returned.

It performs the inverse operation to `url.domainToASCII()`.

This feature is only available if the `node` executable was compiled with `ICU` enabled. If not, the domain names are passed through unchanged.

```
import url from 'url';

console.log(url.domainToUnicode('xn--espaol-zwa.com'));
// Prints español.com
console.log(url.domainToUnicode('xn--fiq228c.com'));
// Prints 中文.com
console.log(url.domainToUnicode('xn--iñvalid.com'));
// Prints an empty stringconst url = require('url');

console.log(url.domainToUnicode('xn--espaol-zwa.com'));
// Prints español.com
console.log(url.domainToUnicode('xn--fiq228c.com'));
// Prints 中文.com
console.log(url.domainToUnicode('xn--iñvalid.com'));
// Prints an empty string
```

url.fileURLToPath(url)

- `url` `<URL>` | `<string>` The file URL string or URL object to convert to a path.
- Returns: `<string>` The fully-resolved platform-specific Node.js file path.

This function ensures the correct decodings of percent-encoded characters as well as ensuring a cross-platform valid absolute path string.

```
import { fileURLToPath } from 'url';

const __filename = fileURLToPath(import.meta.url);
```

```

new URL('file:///C:/path/').pathname;    // Incorrect: /C:/path/
fileURLToPath('file:///C:/path/');       // Correct:   C:\path\ (Windows)

new URL('file://nas/foo.txt').pathname;   // Incorrect: /foo.txt
fileURLToPath('file://nas/foo.txt');      // Correct:   \\nas\foo.txt (Windows)

new URL('file:///你好.txt').pathname;     // Incorrect: /%E4%BD%A0%E5%A5%BD.txt
fileURLToPath('file:///你好.txt');        // Correct:   /你好.txt (POSIX)

new URL('file:///hello world').pathname;  // Incorrect: /hello%20world
fileURLToPath('file:///hello world');     // Correct:   /hello world (POSIX)
const { fileURLToPath } = require('url');

new URL('file:///C:/path/').pathname;     // Incorrect: /C:/path/
fileURLToPath('file:///C:/path/');        // Correct:   C:\path\ (Windows)

new URL('file://nas/foo.txt').pathname;   // Incorrect: /foo.txt
fileURLToPath('file://nas/foo.txt');      // Correct:   \\nas\foo.txt (Windows)

new URL('file:///你好.txt').pathname;     // Incorrect: /%E4%BD%A0%E5%A5%BD.txt
fileURLToPath('file:///你好.txt');        // Correct:   /你好.txt (POSIX)

new URL('file:///hello world').pathname;  // Incorrect: /hello%20world
fileURLToPath('file:///hello world');     // Correct:   /hello world (POSIX)

```

url.format(URL[, options])

- **URL** <URL> A WHATWG URL object
- **options** <Object>
 - **auth** <boolean> **true** if the serialized URL string should include the username and password, **false** otherwise. **Default:** **true**.
 - **fragment** <boolean> **true** if the serialized URL string should include the fragment, **false** otherwise. **Default:** **true**.
 - **search** <boolean> **true** if the serialized URL string should include the search query, **false** otherwise. **Default:** **true**.
 - **unicode** <boolean> **true** if Unicode characters appearing in the host component of the URL string should be encoded directly as opposed to being Punycode encoded. **Default:** **false**.
- **Returns:** <string>

Returns a customizable serialization of a URL **String** representation of a WHATWG URL object.

The URL object has both a `toString()` method and `href` property that return string serializations of the URL. These are not, however, customizable in any way. The `url.format(URL[, options])` method allows for basic customization of the output.

```

import url from 'url';
const myURL = new URL('https://a:b@測試?abc#foo');

console.log(myURL.href);
// Prints https://a:b@xn--g6w251d/?abc#foo

console.log(myURL.toString());
// Prints https://a:b@xn--g6w251d/?abc#foo

console.log(url.format(myURL, { fragment: false, unicode: true, auth: false }));
// Prints 'https://測試?abc'
const url = require('url');

```

```
const myURL = new URL('https://a:b@測試?abc#foo');

console.log(myURL.href);
// Prints https://a:b@xn--g6w251d/?abc#foo

console.log(myURL.toString());
// Prints https://a:b@xn--g6w251d/?abc#foo

console.log(url.format(myURL, { fragment: false, unicode: true, auth: false }));
// Prints 'https://測試/?abc'
```

url.pathToFileURL(path)

- `path` `<string>` The path to convert to a File URL.
- Returns: `<URL>` The file URL object.

This function ensures that `path` is resolved absolutely, and that the URL control characters are correctly encoded when converting into a File URL.

```
import { pathToFileURL } from 'url';

new URL('/foo#1', 'file:');           // Incorrect: file:///foo#1
pathToFileURL('/foo#1');              // Correct:   file:///foo%231 (POSIX)

new URL('/some/path%.c', 'file:');   // Incorrect: file:///some/path%.c
pathToFileURL('/some/path%.c');       // Correct:   file:///some/path%25.c (POSIX)
const { pathToFileURL } = require('url')

new URL(__filename);                  // Incorrect: throws (POSIX)
new URL(__filename);                  // Incorrect: C:\... (Windows)
pathToFileURL(__filename);             // Correct:   file:///... (POSIX)
pathToFileURL(__filename);             // Correct:   file:///C:/... (Windows)

new URL('/foo#1', 'file:');           // Incorrect: file:///foo#1
pathToFileURL('/foo#1');              // Correct:   file:///foo%231 (POSIX)

new URL('/some/path%.c', 'file:');   // Incorrect: file:///some/path%.c
pathToFileURL('/some/path%.c');       // Correct:   file:///some/path%25.c (POSIX)
```

url.urlToHttpOptions(url)

- `url` `<URL>` The **WHATWG URL** object to convert to an options object.
- Returns: `<Object>` Options object
 - `protocol` `<string>` Protocol to use.
 - `hostname` `<string>` A domain name or IP address of the server to issue the request to.
 - `hash` `<string>` The fragment portion of the URL.
 - `search` `<string>` The serialized query portion of the URL.
 - `pathname` `<string>` The path portion of the URL.
 - `path` `<string>` Request path. Should include query string if any. E.G. `'/index.html?page=12'`. An exception is thrown when the request path contains illegal characters. Currently, only spaces are rejected but that may change in the future.

- `href` `<string>` The serialized URL.
- `port` `<number>` Port of remote server.
- `auth` `<string>` Basic authentication i.e. `'user:password'` to compute an Authorization header.

This utility function converts a URL object into an ordinary options object as expected by the `http.request()` and `https.request()` APIs.

```
import { urlToHttpOptions } from 'url';
const myURL = new URL('https://a:b@xn--g6w251d/?abc#foo');

console.log(urlToHttpOptions(myURL));
/**
{
  protocol: 'https:',
  hostname: 'xn--g6w251d',
  hash: '#foo',
  search: '?abc',
  pathname: '/',
  path: '/?abc',
  href: 'https://a:b@xn--g6w251d/?abc#foo',
  auth: 'a:b'
}
*/const { urlToHttpOptions } = require('url');
const myURL = new URL('https://a:b@xn--g6w251d/?abc#foo');

console.log(urlToHttpOptions(myURL));
/**
{
  protocol: 'https:',
  hostname: 'xn--g6w251d',
  hash: '#foo',
  search: '?abc',
  pathname: '/',
  path: '/?abc',
  href: 'https://a:b@xn--g6w251d/?abc#foo',
  auth: 'a:b'
}
*/
```

Legacy URL API

Stability: 3 - Legacy: Use the WHATWG URL API instead.

Legacy `urlObject`

Stability: 3 - Legacy: Use the WHATWG URL API instead.

The legacy `urlObject` (`require('url').Url` or `import { Url } from 'url'`) is created and returned by the `url.parse()` function.

urlObject.auth

The `auth` property is the username and password portion of the URL, also referred to as *userinfo*. This string subset follows the `protocol` and double slashes (if present) and precedes the `host` component, delimited by `@`. The string is either the username, or it is the username and password separated by `:`.

For example: `'user:pass'`.

urlObject.hash

The `hash` property is the fragment identifier portion of the URL including the leading `#` character.

For example: `'#hash'`.

urlObject.host

The `host` property is the full lower-cased host portion of the URL, including the `port` if specified.

For example: `'sub.example.com:8080'`.

urlObject.hostname

The `hostname` property is the lower-cased host name portion of the `host` component *without* the `port` included.

For example: `'sub.example.com'`.

urlObject.href

The `href` property is the full URL string that was parsed with both the `protocol` and `host` components converted to lower-case.

For example: `'http://user:pass@sub.example.com:8080/p/a/t/h?query=string#hash'`.

urlObject.path

The `path` property is a concatenation of the `pathname` and `search` components.

For example: `'/p/a/t/h?query=string'`.

No decoding of the `path` is performed.

urlObject.pathname

The `pathname` property consists of the entire path section of the URL. This is everything following the `host` (including the `port`) and before the start of the `query` or `hash` components, delimited by either the ASCII question mark (`?`) or hash (`#`) characters.

For example: `'/p/a/t/h'`.

No decoding of the path string is performed.

urlObject.port

The `port` property is the numeric port portion of the `host` component.

For example: `'8080'`.

urlObject.protocol

The `protocol` property identifies the URL's lower-cased protocol scheme.

For example: `'http: '`.

`urlObject.query`

The `query` property is either the query string without the leading ASCII question mark (`?`), or an object returned by the `querystring` module's `parse()` method. Whether the `query` property is a string or object is determined by the `parseQueryString` argument passed to `url.parse()`.

For example: `'query=string'` or `{'query': 'string'}`.

If returned as a string, no decoding of the query string is performed. If returned as an object, both keys and values are decoded.

`urlObject.search`

The `search` property consists of the entire "query string" portion of the URL, including the leading ASCII question mark (`?`) character.

For example: `'?query=string'`.

No decoding of the query string is performed.

`urlObject.slashes`

The `slashes` property is a `boolean` with a value of `true` if two ASCII forward-slash characters (`/`) are required following the colon in the `protocol`.

`url.format(urlObject)`

Stability: 3 - Legacy: Use the WHATWG URL API instead.

- `urlObject` `<Object>` | `<string>` A URL object (as returned by `url.parse()` or constructed otherwise). If a string, it is converted to an object by passing it to `url.parse()`.

The `url.format()` method returns a formatted URL string derived from `urlObject`.

```
const url = require('url');
url.format({
  protocol: 'https',
  hostname: 'example.com',
  pathname: '/some/path',
  query: {
    page: 1,
    format: 'json'
  }
});

// => 'https://example.com/some/path?page=1&format=json'
```

If `urlObject` is not an object or a string, `url.format()` will throw a `TypeError`.

The formatting process operates as follows:

- A new empty string `result` is created.
- If `urlObject.protocol` is a string, it is appended as-is to `result`.

- Otherwise, if `urlObject.protocol` is not `undefined` and is not a string, an `Error` is thrown.
- For all string values of `urlObject.protocol` that *do not end* with an ASCII colon (`:`) character, the literal string `:` will be appended to `result`.
- If either of the following conditions is true, then the literal string `//` will be appended to `result` :
 - `urlObject.slashes` property is true;
 - `urlObject.protocol` begins with `http`, `https`, `ftp`, `gopher`, or `file`;
- If the value of the `urlObject.auth` property is truthy, and either `urlObject.host` or `urlObject.hostname` are not `undefined`, the value of `urlObject.auth` will be coerced into a string and appended to `result` followed by the literal string `@`.
- If the `urlObject.host` property is `undefined` then:
 - If the `urlObject.hostname` is a string, it is appended to `result`.
 - Otherwise, if `urlObject.hostname` is not `undefined` and is not a string, an `Error` is thrown.
 - If the `urlObject.port` property value is truthy, and `urlObject.hostname` is not `undefined` :
 - The literal string `:` is appended to `result`, and
 - The value of `urlObject.port` is coerced to a string and appended to `result`.
- Otherwise, if the `urlObject.host` property value is truthy, the value of `urlObject.host` is coerced to a string and appended to `result`.
- If the `urlObject.pathname` property is a string that is not an empty string:
 - If the `urlObject.pathname` *does not start* with an ASCII forward slash (`/`), then the literal string `/'` is appended to `result`.
 - The value of `urlObject.pathname` is appended to `result`.
- Otherwise, if `urlObject.pathname` is not `undefined` and is not a string, an `Error` is thrown.
- If the `urlObject.search` property is `undefined` and if the `urlObject.query` property is an `Object`, the literal string `?` is appended to `result` followed by the output of calling the `querystring` module's `stringify()` method passing the value of `urlObject.query`.
- Otherwise, if `urlObject.search` is a string:
 - If the value of `urlObject.search` *does not start* with the ASCII question mark (`?`) character, the literal string `?` is appended to `result`.
 - The value of `urlObject.search` is appended to `result`.
- Otherwise, if `urlObject.search` is not `undefined` and is not a string, an `Error` is thrown.
- If the `urlObject.hash` property is a string:
 - If the value of `urlObject.hash` *does not start* with the ASCII hash (`#`) character, the literal string `#` is appended to `result`.
 - The value of `urlObject.hash` is appended to `result`.
- Otherwise, if the `urlObject.hash` property is not `undefined` and is not a string, an `Error` is thrown.
- `result` is returned.

`url.parse(urlString[, parseQueryString[, slashesDenoteHost]])`

Stability: 3 - Legacy: Use the WHATWG URL API instead.

- `urlString` `<string>` The URL string to parse.
- `parseQueryString` `<boolean>` If `true`, the `query` property will always be set to an object returned by the `querystring` module's `parse()` method. If `false`, the `query` property on the returned URL object will be an unparsed, undecoded string. **Default:** `false`.
- `slashesDenoteHost` `<boolean>` If `true`, the first token after the literal string `//` and preceding the next `/` will be interpreted as the `host`. For instance, given `//foo/bar`, the result would be `{host: 'foo', pathname: '/bar'}` rather than `{pathname: '//foo/bar'}`. **Default:** `false`.

The `url.parse()` method takes a URL string, parses it, and returns a URL object.

A `TypeError` is thrown if `urlString` is not a string.

A `URIError` is thrown if the `auth` property is present but cannot be decoded.

Use of the legacy `url.parse()` method is discouraged. Users should use the WHATWG `URL` API. Because the `url.parse()` method uses a lenient, non-standard algorithm for parsing URL strings, security issues can be introduced. Specifically, issues with [host name spoofing](#) and incorrect handling of usernames and passwords have been identified.

`url.resolve(from, to)`

Stability: 3 - Legacy: Use the WHATWG URL API instead.

- `from` `<string>` The Base URL being resolved against.
- `to` `<string>` The HREF URL being resolved.

The `url.resolve()` method resolves a target URL relative to a base URL in a manner similar to that of a Web browser resolving an anchor tag HREF.

```
const url = require('url');
url.resolve('/one/two/three', 'four');           // '/one/two/four'
url.resolve('http://example.com/', 'one');      // 'http://example.com/one'
url.resolve('http://example.com/one', 'two');    // 'http://example.com/two'
```

You can achieve the same result using the WHATWG URL API:

```
function resolve(from, to) {
  const resolvedUrl = new URL(to, new URL(from, 'resolve:'));
  if (resolvedUrl.protocol === 'resolve:') {
    // `from` is a relative URL.
    const { pathname, search, hash } = resolvedUrl;
    return pathname + search + hash;
  }
  return resolvedUrl.toString();
}

resolve('/one/two/three', 'four');           // '/one/two/four'
resolve('http://example.com/', 'one');      // 'http://example.com/one'
resolve('http://example.com/one', 'two');    // 'http://example.com/two'
```

Percent-encoding in URLs

URLs are permitted to only contain a certain range of characters. Any character falling outside of that range must be encoded. How such characters are encoded, and which characters to encode depends entirely on where the character is located within the structure of the URL.

Legacy API

Within the Legacy API, spaces (`' '`) and the following characters will be automatically escaped in the properties of URL objects:

```
< > " \r \n \t { } | \^ '
```

For example, the ASCII space character (' ') is encoded as %20 . The ASCII forward slash (/) character is encoded as %3C .

WHATWG API

The [WHATWG URL Standard](#) uses a more selective and fine grained approach to selecting encoded characters than that used by the Legacy API.

The WHATWG algorithm defines four "percent-encode sets" that describe ranges of characters that must be percent-encoded:

- The *CO control percent-encode set* includes code points in range U+0000 to U+001F (inclusive) and all code points greater than U+007E.
- The *fragment percent-encode set* includes the *CO control percent-encode set* and code points U+0020, U+0022, U+003C, U+003E, and U+0060.
- The *path percent-encode set* includes the *CO control percent-encode set* and code points U+0020, U+0022, U+0023, U+003C, U+003E, U+003F, U+0060, U+007B, and U+007D.
- The *userinfo encode set* includes the *path percent-encode set* and code points U+002F, U+003A, U+003B, U+003D, U+0040, U+005B, U+005C, U+005D, U+005E, and U+007C.

The *userinfo percent-encode set* is used exclusively for username and passwords encoded within the URL. The *path percent-encode set* is used for the path of most URLs. The *fragment percent-encode set* is used for URL fragments. The *CO control percent-encode set* is used for host and path under certain specific conditions, in addition to all other cases.

When non-ASCII characters appear within a host name, the host name is encoded using the [Punycode](#) algorithm. Note, however, that a host name *may* contain *both* Punycode encoded and percent-encoded characters:

```
const myURL = new URL('https://%CF%80.example.com/foo');
console.log(myURL.href);
// Prints https://xn--1xa.example.com/foo
console.log(myURL.origin);
// Prints https://xn--1xa.example.com
```

Util

Stability: 2 - Stable

Source Code: [lib/util.js](#)

The `util` module supports the needs of Node.js internal APIs. Many of the utilities are useful for application and module developers as well. To access it:

```
const util = require('util');
```

util.callbackify(original)

- `original` `<Function>` An async function
- Returns: `<Function>` a callback style function

Takes an `async` function (or a function that returns a `Promise`) and returns a function following the error-first callback style, i.e. taking an `(err, value) => ...` callback as the last argument. In the callback, the first argument will be the rejection reason (or `null` if the `Promise`

resolved), and the second argument will be the resolved value.

```
const util = require('util');

async function fn() {
  return 'hello world';
}

const callbackFunction = util.callbackify(fn);

callbackFunction((err, ret) => {
  if (err) throw err;
  console.log(ret);
});
```

Will print:

```
hello world
```

The callback is executed asynchronously, and will have a limited stack trace. If the callback throws, the process will emit an `'uncaughtException'` event, and if not handled will exit.

Since `null` has a special meaning as the first argument to a callback, if a wrapped function rejects a `Promise` with a falsy value as a reason, the value is wrapped in an `Error` with the original value stored in a field named `reason`.

```
function fn() {
  return Promise.reject(null);
}

const callbackFunction = util.callbackify(fn);

callbackFunction((err, ret) => {
  // When the Promise was rejected with `null` it is wrapped with an Error and
  // the original value is stored in `reason`.
  err && err.hasOwnProperty('reason') && err.reason === null; // true
});
```

util.debuglog(section[, callback])

- `section` `<string>` A string identifying the portion of the application for which the `debuglog` function is being created.
- `callback` `<Function>` A callback invoked the first time the logging function is called with a function argument that is a more optimized logging function.
- Returns: `<Function>` The logging function

The `util.debuglog()` method is used to create a function that conditionally writes debug messages to `stderr` based on the existence of the `NODE_DEBUG` environment variable. If the `section` name appears within the value of that environment variable, then the returned function operates similar to `console.error()`. If not, then the returned function is a no-op.

```
const util = require('util');
const debuglog = util.debuglog('foo');
```

```
debuglog('hello from foo [%d]', 123);
```

If this program is run with `NODE_DEBUG=foo` in the environment, then it will output something like:

```
F00 3245: hello from foo [123]
```

where `3245` is the process id. If it is not run with that environment variable set, then it will not print anything.

The `section` supports wildcard also:

```
const util = require('util');
const debuglog = util.debuglog('foo-bar');

debuglog('hi there, it\'s foo-bar [%d]', 2333);
```

if it is run with `NODE_DEBUG=foo*` in the environment, then it will output something like:

```
F00-BAR 3257: hi there, it's foo-bar [2333]
```

Multiple comma-separated `section` names may be specified in the `NODE_DEBUG` environment variable: `NODE_DEBUG=fs,net,tls`.

The optional `callback` argument can be used to replace the logging function with a different function that doesn't have any initialization or unnecessary wrapping.

```
const util = require('util');
let debuglog = util.debuglog('internals', (debug) => {
  // Replace with a logging function that optimizes out
  // testing if the section is enabled
  debuglog = debug;
});
```

`debuglog().enabled`

- `<boolean>`

The `util.debuglog().enabled` getter is used to create a test that can be used in conditionals based on the existence of the `NODE_DEBUG` environment variable. If the `section` name appears within the value of that environment variable, then the returned value will be `true`. If not, then the returned value will be `false`.

```
const util = require('util');
const enabled = util.debuglog('foo').enabled;
if (enabled) {
  console.log('hello from foo [%d]', 123);
}
```

If this program is run with `NODE_DEBUG=foo` in the environment, then it will output something like:

```
hello from foo [123]
```

util.debug(section)

Alias for `util.debugLog`. Usage allows for readability of that doesn't imply logging when only using `util.debugLog().enabled`.

util.deprecate(fn, msg[, code])

- `fn` `<Function>` The function that is being deprecated.
- `msg` `<string>` A warning message to display when the deprecated function is invoked.
- `code` `<string>` A deprecation code. See the [list of deprecated APIs](#) for a list of codes.
- Returns: `<Function>` The deprecated function wrapped to emit a warning.

The `util.deprecate()` method wraps `fn` (which may be a function or class) in such a way that it is marked as deprecated.

```
const util = require('util');

exports.obsoleteFunction = util.deprecate(() => {
  // Do something here.
}, 'obsoleteFunction() is deprecated. Use newShinyFunction() instead.');
```

When called, `util.deprecate()` will return a function that will emit a `DeprecationWarning` using the `'warning'` event. The warning will be emitted and printed to `stderr` the first time the returned function is called. After the warning is emitted, the wrapped function is called without emitting a warning.

If the same optional `code` is supplied in multiple calls to `util.deprecate()`, the warning will be emitted only once for that `code`.

```
const util = require('util');

const fn1 = util.deprecate(someFunction, someMessage, 'DEP0001');
const fn2 = util.deprecate(someOtherFunction, someOtherMessage, 'DEP0001');
fn1(); // Emits a deprecation warning with code DEP0001
fn2(); // Does not emit a deprecation warning because it has the same code
```

If either the `--no-deprecation` or `--no-warnings` command-line flags are used, or if the `process.noDeprecation` property is set to `true` prior to the first deprecation warning, the `util.deprecate()` method does nothing.

If the `--trace-deprecation` or `--trace-warnings` command-line flags are set, or the `process.traceDeprecation` property is set to `true`, a warning and a stack trace are printed to `stderr` the first time the deprecated function is called.

If the `--throw-deprecation` command-line flag is set, or the `process.throwDeprecation` property is set to `true`, then an exception will be thrown when the deprecated function is called.

The `--throw-deprecation` command-line flag and `process.throwDeprecation` property take precedence over `--trace-deprecation` and `process.traceDeprecation`.

util.format(format[, ...args])

- `format` `<string>` A `printf`-like format string.

The `util.format()` method returns a formatted string using the first argument as a `printf`-like format string which can contain zero or more format specifiers. Each specifier is replaced with the converted value from the corresponding argument. Supported specifiers are:

- `%s`: `String` will be used to convert all values except `BigInt`, `Object` and `-0`. `BigInt` values will be represented with an `n` and Objects that have no user defined `toString` function are inspected using `util.inspect()` with options `{ depth: 0, colors: false, compact: 3 }`.
- `%d`: `Number` will be used to convert all values except `BigInt` and `Symbol`.
- `%i`: `parseInt(value, 10)` is used for all values except `BigInt` and `Symbol`.
- `%f`: `parseFloat(value)` is used for all values except `Symbol`.
- `%j`: JSON. Replaced with the string `'[Circular]'` if the argument contains circular references.
- `%o`: `Object`. A string representation of an object with generic JavaScript object formatting. Similar to `util.inspect()` with options `{ showHidden: true, showProxy: true }`. This will show the full object including non-enumerable properties and proxies.
- `%O`: `Object`. A string representation of an object with generic JavaScript object formatting. Similar to `util.inspect()` without options. This will show the full object not including non-enumerable properties and proxies.
- `%c`: CSS. This specifier is ignored and will skip any CSS passed in.
- `%%`: single percent sign (`'%'`). This does not consume an argument.
- Returns: `<string>` The formatted string

If a specifier does not have a corresponding argument, it is not replaced:

```
util.format('%s:%s', 'foo');  
// Returns: 'foo:%s'
```

Values that are not part of the format string are formatted using `util.inspect()` if their type is not `string`.

If there are more arguments passed to the `util.format()` method than the number of specifiers, the extra arguments are concatenated to the returned string, separated by spaces:

```
util.format('%s:%s', 'foo', 'bar', 'baz');  
// Returns: 'foo:bar baz'
```

If the first argument does not contain a valid format specifier, `util.format()` returns a string that is the concatenation of all arguments separated by spaces:

```
util.format(1, 2, 3);  
// Returns: '1 2 3'
```

If only one argument is passed to `util.format()`, it is returned as it is without any formatting:

```
util.format('%% %s');  
// Returns: '%% %s'
```

`util.format()` is a synchronous method that is intended as a debugging tool. Some input values can have a significant performance overhead that can block the event loop. Use this function with care and never in a hot code path.

util.formatWithOptions(inspectOptions, format[, ...args])

- `inspectOptions` `<Object>`

- `format` `<string>`

This function is identical to `util.format()`, except in that it takes an `inspectOptions` argument which specifies options that are passed along to `util.inspect()`.

```
util.formatWithOptions({ colors: true }, 'See object %O', { foo: 42 });
// Returns 'See object { foo: 42 }', where `42` is colored as a number
// when printed to a terminal.
```

util.getSystemErrorName(err)

- `err` `<number>`
- Returns: `<string>`

Returns the string name for a numeric error code that comes from a Node.js API. The mapping between error codes and error names is platform-dependent. See [Common System Errors](#) for the names of common errors.

```
fs.access('file/that/does/not/exist', (err) => {
  const name = util.getSystemErrorName(err.errno);
  console.error(name); // ENOENT
});
```

util.getSystemErrorMap()

- Returns: `<Map>`

Returns a Map of all system error codes available from the Node.js API. The mapping between error codes and error names is platform-dependent. See [Common System Errors](#) for the names of common errors.

```
fs.access('file/that/does/not/exist', (err) => {
  const errorMap = util.getSystemErrorMap();
  const name = errorMap.get(err.errno);
  console.error(name); // ENOENT
});
```

util.inherits(constructor, superConstructor)

Stability: 3 - Legacy: Use ES2015 class syntax and `extends` keyword instead.

- `constructor` `<Function>`
- `superConstructor` `<Function>`

Usage of `util.inherits()` is discouraged. Please use the ES6 `class` and `extends` keywords to get language level inheritance support. Also note that the two styles are [semantically incompatible](#).

Inherit the prototype methods from one `constructor` into another. The prototype of `constructor` will be set to a new object created from `superConstructor`.

This mainly adds some input validation on top of `Object.setPrototypeOf(constructor.prototype, superConstructor.prototype)`. As an additional convenience, `superConstructor` will be accessible through the `constructor.super_` property.

```
const util = require('util');
const EventEmitter = require('events');

function MyStream() {
  EventEmitter.call(this);
}

util.inherits(MyStream, EventEmitter);

MyStream.prototype.write = function(data) {
  this.emit('data', data);
};

const stream = new MyStream();

console.log(stream instanceof EventEmitter); // true
console.log(MyStream.super_ === EventEmitter); // true

stream.on('data', (data) => {
  console.log(`Received data: "${data}"`);
});

stream.write('It works!'); // Received data: "It works!"
```

ES6 example using `class` and `extends`:

```
const EventEmitter = require('events');

class MyStream extends EventEmitter {
  write(data) {
    this.emit('data', data);
  }
}

const stream = new MyStream();

stream.on('data', (data) => {
  console.log(`Received data: "${data}"`);
});

stream.write('With ES6');
```

`util.inspect(object[, options])`

`util.inspect(object[, showHidden[, depth[, colors]]])`

- `object` `<any>` Any JavaScript primitive or `Object`.

- `options` `<Object>`
 - `showHidden` `<boolean>` If `true`, object's non-enumerable symbols and properties are included in the formatted result. `WeakMap` and `WeakSet` entries are also included as well as user defined prototype properties (excluding method properties). **Default:** `false`.
 - `depth` `<number>` Specifies the number of times to recurse while formatting `object`. This is useful for inspecting large objects. To recurse up to the maximum call stack size pass `Infinity` or `null`. **Default:** `2`.
 - `colors` `<boolean>` If `true`, the output is styled with ANSI color codes. Colors are customizable. See `Customizing util.inspect colors`. **Default:** `false`.
 - `customInspect` `<boolean>` If `false`, `[util.inspect.custom](depth, opts)` functions are not invoked. **Default:** `true`.
 - `showProxy` `<boolean>` If `true`, Proxy inspection includes the `target` and `handler` objects. **Default:** `false`.
 - `maxArrayLength` `<integer>` Specifies the maximum number of `Array`, `TypedArray`, `WeakMap` and `WeakSet` elements to include when formatting. Set to `null` or `Infinity` to show all elements. Set to `0` or negative to show no elements. **Default:** `100`.
 - `maxLength` `<integer>` Specifies the maximum number of characters to include when formatting. Set to `null` or `Infinity` to show all elements. Set to `0` or negative to show no characters. **Default:** `10000`.
 - `breakLength` `<integer>` The length at which input values are split across multiple lines. Set to `Infinity` to format the input as a single line (in combination with `compact` set to `true` or any number `>= 1`). **Default:** `80`.
 - `compact` `<boolean>` | `<integer>` Setting this to `false` causes each object key to be displayed on a new line. It will break on new lines in text that is longer than `breakLength`. If set to a number, the most `n` inner elements are united on a single line as long as all properties fit into `breakLength`. Short array elements are also grouped together. For more information, see the example below. **Default:** `3`.
 - `sorted` `<boolean>` | `<Function>` If set to `true` or a function, all properties of an object, and `Set` and `Map` entries are sorted in the resulting string. If set to `true` the `default sort` is used. If set to a function, it is used as a `compare function`.
 - `getters` `<boolean>` | `<string>` If set to `true`, getters are inspected. If set to `'get'`, only getters without a corresponding setter are inspected. If set to `'set'`, only getters with a corresponding setter are inspected. This might cause side effects depending on the getter function. **Default:** `false`.
- Returns: `<string>` The representation of `object`.

The `util.inspect()` method returns a string representation of `object` that is intended for debugging. The output of `util.inspect` may change at any time and should not be depended upon programmatically. Additional `options` may be passed that alter the result.

`util.inspect()` will use the constructor's name and/or `@@toStringTag` to make an identifiable tag for an inspected value.

```
class Foo {
  get [Symbol.toStringTag]() {
    return 'bar';
  }
}

class Bar {}

const baz = Object.create(null, { [Symbol.toStringTag]: { value: 'foo' } });

util.inspect(new Foo()); // 'Foo [bar] {}'
util.inspect(new Bar()); // 'Bar {}'
util.inspect(baz);       // '[foo] {}'
```

Circular references point to their anchor by using a reference index:

```
const { inspect } = require('util');
```

```

const obj = {};
obj.a = [obj];
obj.b = {};
obj.b.inner = obj.b;
obj.b.obj = obj;

console.log(inspect(obj));
// <ref *1> {
//   a: [ [Circular *1] ],
//   b: <ref *2> { inner: [Circular *2], obj: [Circular *1] }
// }

```

The following example inspects all properties of the `util` object:

```

const util = require('util');

console.log(util.inspect(util, { showHidden: true, depth: null }));

```

The following example highlights the effect of the `compact` option:

```

const util = require('util');

const o = {
  a: [1, 2, [
    'Lorem ipsum dolor sit amet,\nconsectetur adipiscing elit, sed do ' +
    'eiusmod \ntempor incididunt ut labore et dolore magna aliqua.',
    'test',
    'foo']], 4],
  b: new Map([['za', 1], ['zb', 'test']])
};

console.log(util.inspect(o, { compact: true, depth: 5, breakLength: 80 }));

// { a:
//   [ 1,
//     2,
//     [ [ 'Lorem ipsum dolor sit amet,\nconsectetur [...]', // A long line
//         'test',
//         'foo' ] ],
//     4 ],
//   b: Map(2) { 'za' => 1, 'zb' => 'test' } }

// Setting `compact` to false or an integer creates more reader friendly output.
console.log(util.inspect(o, { compact: false, depth: 5, breakLength: 80 }));

// {
//   a: [
//     1,
//     2,
//     [
//       [

```

```
//      'Lorem ipsum dolor sit amet,\n' +
//      'consectetur adipiscing elit, sed do eiusmod \n' +
//      'tempor incididunt ut labore et dolore magna aliqua.',
//      'test',
//      'foo'
//    ]
//  ],
//  4
// ],
// b: Map(2) {
//   'za' => 1,
//   'zb' => 'test'
// }
// }

// Setting `breakLength` to e.g. 150 will print the "Lorem ipsum" text in a
// single line.
```

The `showHidden` option allows `WeakMap` and `WeakSet` entries to be inspected. If there are more entries than `maxLength`, there is no guarantee which entries are displayed. That means retrieving the same `WeakSet` entries twice may result in different output. Furthermore, entries with no remaining strong references may be garbage collected at any time.

```
const { inspect } = require('util');

const obj = { a: 1 };
const obj2 = { b: 2 };
const weakSet = new WeakSet([obj, obj2]);

console.log(inspect(weakSet, { showHidden: true }));
// WeakSet { { a: 1 }, { b: 2 } }
```

The `sorted` option ensures that an object's property insertion order does not impact the result of `util.inspect()`.

```
const { inspect } = require('util');
const assert = require('assert');

const o1 = {
  b: [2, 3, 1],
  a: '`a` comes before `b`',
  c: new Set([2, 3, 1])
};
console.log(inspect(o1, { sorted: true }));
// { a: '`a` comes before `b`', b: [ 2, 3, 1 ], c: Set(3) { 1, 2, 3 } }
console.log(inspect(o1, { sorted: (a, b) => b.localeCompare(a) }));
// { c: Set(3) { 3, 2, 1 }, b: [ 2, 3, 1 ], a: '`a` comes before `b`' }

const o2 = {
  c: new Set([2, 1, 3]),
  a: '`a` comes before `b`',
  b: [2, 3, 1]
```

```
};
assert.strict.equal(
  inspect(o1, { sorted: true }),
  inspect(o2, { sorted: true })
);
```

`util.inspect()` is a synchronous method intended for debugging. Its maximum output length is approximately 128 MB. Inputs that result in longer output will be truncated.

Customizing `util.inspect` colors

Color output (if enabled) of `util.inspect` is customizable globally via the `util.inspect.styles` and `util.inspect.colors` properties.

`util.inspect.styles` is a map associating a style name to a color from `util.inspect.colors`.

The default styles and associated colors are:

- `bigint`: yellow
- `boolean`: yellow
- `date`: magenta
- `module`: underline
- `name`: (no styling)
- `null`: bold
- `number`: yellow
- `regexp`: red
- `special`: cyan (e.g., `Proxies`)
- `string`: green
- `symbol`: green
- `undefined`: grey

Color styling uses ANSI control codes that may not be supported on all terminals. To verify color support use `tty.hasColors()`.

Predefined control codes are listed below (grouped as "Modifiers", "Foreground colors", and "Background colors").

Modifiers

Modifier support varies throughout different terminals. They will mostly be ignored, if not supported.

- `reset` - Resets all (color) modifiers to their defaults
- `bold` - Make text bold
- *italic* - Make text italic
- `underline` - Make text underlined
- `strikethrough` - Puts a horizontal line through the center of the text (Alias: `strikeThrough`, `crossedout`, `crossedOut`)
- `hidden` - Prints the text, but makes it invisible (Alias: `conceal`)
- `dim` - Decreased color intensity (Alias: `faint`)
- `overlined` - Make text overlined
- `blink` - Hides and shows the text in an interval
- `inverse` - Swap foreground and background colors (Alias: `swapcolors`, `swapColors`)
- `doubleunderline` - Make text double underlined (Alias: `doubleUnderLine`)

- `framed` - Draw a frame around the text

Foreground colors

- `black`
- `red`
- `green`
- `yellow`
- `blue`
- `magenta`
- `cyan`
- `white`
- `gray` (alias: `grey`, `blackBright`)
- `redBright`
- `greenBright`
- `yellowBright`
- `blueBright`
- `magentaBright`
- `cyanBright`
- `whiteBright`

Background colors

- `bgBlack`
- `bgRed`
- `bgGreen`
- `bgYellow`
- `bgBlue`
- `bgMagenta`
- `bgCyan`
- `bgWhite`
- `bgGray` (alias: `bgGrey`, `bgBlackBright`)
- `bgRedBright`
- `bgGreenBright`
- `bgYellowBright`
- `bgBlueBright`
- `bgMagentaBright`
- `bgCyanBright`
- `bgWhiteBright`

Custom inspection functions on objects

Objects may also define their own `[util.inspect.custom](depth, opts)` function, which `util.inspect()` will invoke and use the result of when inspecting the object:

```

const util = require('util');

class Box {
  constructor(value) {
    this.value = value;
  }

  [util.inspect.custom](depth, options) {
    if (depth < 0) {
      return options.stylize('[Box]', 'special');
    }

    const newOptions = Object.assign({}, options, {
      depth: options.depth === null ? null : options.depth - 1
    });

    // Five space padding because that's the size of "Box< ".
    const padding = ' '.repeat(5);
    const inner = util.inspect(this.value, newOptions)
      .replace(/\n/g, `\n${padding}`);
    return `${options.stylize('Box', 'special')}< ${inner} >`;
  }
}

const box = new Box(true);

util.inspect(box);
// Returns: "Box< true >"

```

Custom `[util.inspect.custom](depth, opts)` functions typically return a string but may return a value of any type that will be formatted accordingly by `util.inspect()`.

```

const util = require('util');

const obj = { foo: 'this will not show up in the inspect() output' };
obj[util.inspect.custom] = (depth) => {
  return { bar: 'baz' };
};

util.inspect(obj);
// Returns: "{ bar: 'baz' }"

```

util.inspect.custom

- `<symbol>` that can be used to declare custom inspect functions.

In addition to being accessible through `util.inspect.custom`, this symbol is **registered globally** and can be accessed in any environment as `Symbol.for('nodejs.util.inspect.custom')`.

```
const inspect = Symbol.for('nodejs.util.inspect.custom');

class Password {
  constructor(value) {
    this.value = value;
  }

  toString() {
    return 'xxxxxxx';
  }

  [inspect]() {
    return `Password <${this.toString()}>`;
  }
}

const password = new Password('r0sebud');
console.log(password);
// Prints Password <xxxxxxx>
```

See [Custom inspection functions on Objects](#) for more details.

util.inspect.defaultOptions

The `defaultOptions` value allows customization of the default options used by `util.inspect`. This is useful for functions like `console.log` or `util.format` which implicitly call into `util.inspect`. It shall be set to an object containing one or more valid `util.inspect()` options. Setting option properties directly is also supported.

```
const util = require('util');
const arr = Array(101).fill(0);

console.log(arr); // Logs the truncated array
util.inspect.defaultOptions.maxArrayLength = null;
console.log(arr); // logs the full array
```

util.isDeepStrictEqual(val1, val2)

- `val1` `<any>`
- `val2` `<any>`
- Returns: `<boolean>`

Returns `true` if there is deep strict equality between `val1` and `val2`. Otherwise, returns `false`.

See `assert.deepStrictEqual()` for more information about deep strict equality.

util.promisify(original)

- `original` `<Function>`
- Returns: `<Function>`

Takes a function following the common error-first callback style, i.e. taking an `(err, value) => ...` callback as the last argument, and returns a version that returns promises.

```
const util = require('util');
const fs = require('fs');

const stat = util.promisify(fs.stat);
stat('.').then((stats) => {
  // Do something with `stats`
}).catch((error) => {
  // Handle the error.
});
```

Or, equivalently using `async` functions:

```
const util = require('util');
const fs = require('fs');

const stat = util.promisify(fs.stat);

async function callStat() {
  const stats = await stat('.');
  console.log(`This directory is owned by ${stats.uid}`);
}
```

If there is an `original[util.promisify.custom]` property present, `promisify` will return its value, see [Custom promisified functions](#).

`promisify()` assumes that `original` is a function taking a callback as its final argument in all cases. If `original` is not a function, `promisify()` will throw an error. If `original` is a function but its last argument is not an error-first callback, it will still be passed an error-first callback as its last argument.

Using `promisify()` on class methods or other methods that use `this` may not work as expected unless handled specially:

```
const util = require('util');

class Foo {
  constructor() {
    this.a = 42;
  }

  bar(callback) {
    callback(null, this.a);
  }
}

const foo = new Foo();

const naiveBar = util.promisify(foo.bar);
// TypeError: Cannot read property 'a' of undefined
// naiveBar().then(a => console.log(a));
```

```
naiveBar.call(foo).then((a) => console.log(a)); // '42'

const bindBar = naiveBar.bind(foo);
bindBar().then((a) => console.log(a)); // '42'
```

Custom promisified functions

Using the `util.promisify.custom` symbol one can override the return value of `util.promisify()`:

```
const util = require('util');

function doSomething(foo, callback) {
  // ...
}

doSomething[util.promisify.custom] = (foo) => {
  return getPromiseSomehow();
};

const promisified = util.promisify(doSomething);
console.log(promisified === doSomething[util.promisify.custom]);
// prints 'true'
```

This can be useful for cases where the original function does not follow the standard format of taking an error-first callback as the last argument.

For example, with a function that takes in `(foo, onSuccessCallback, onErrorCallback)`:

```
doSomething[util.promisify.custom] = (foo) => {
  return new Promise((resolve, reject) => {
    doSomething(foo, resolve, reject);
  });
};
```

If `promisify.custom` is defined but is not a function, `promisify()` will throw an error.

`util.promisify.custom`

- `<symbol>` that can be used to declare custom promisified variants of functions, see [Custom promisified functions](#).

In addition to being accessible through `util.promisify.custom`, this symbol is [registered globally](#) and can be accessed in any environment as `Symbol.for('nodejs.util.promisify.custom')`.

For example, with a function that takes in `(foo, onSuccessCallback, onErrorCallback)`:

```
const kCustomPromisifiedSymbol = Symbol.for('nodejs.util.promisify.custom');

doSomething[kCustomPromisifiedSymbol] = (foo) => {
  return new Promise((resolve, reject) => {
    doSomething(foo, resolve, reject);
  });
};
```

```
});  
};
```

Class: `util.TextDecoder`

An implementation of the [WHATWG Encoding Standard](#) `TextDecoder` API.

```
const decoder = new TextDecoder('shift_jis');  
let string = '';  
let buffer;  
while (buffer = getNextChunkSomehow()) {  
  string += decoder.decode(buffer, { stream: true });  
}  
string += decoder.decode(); // end-of-stream
```

WHATWG supported encodings

Per the [WHATWG Encoding Standard](#), the encodings supported by the `TextDecoder` API are outlined in the tables below. For each encoding, one or more aliases may be used.

Different Node.js build configurations support different sets of encodings. (see [Internationalization](#))

Encodings supported by default (with full ICU data)

Encoding	Aliases
'ibm866' 'i'	'866', 'cp866', 'csibm866'
'iso-8859-2'	'cisolatin2', 'iso-ir-101', 'iso8859-2', 'iso88592', 'iso_8859-2', 'iso_8859-2:1987', 'l2', 'latin2'
'iso-8859-3'	'cisolatin3', 'iso-ir-109', 'iso8859-3', 'iso88593', 'iso_8859-3', 'iso_8859-3:1988', 'l3', 'latin3'
'iso-8859-4'	'cisolatin4', 'iso-ir-110', 'iso8859-4', 'iso88594', 'iso_8859-4', 'iso_8859-4:1988', 'l4', 'latin4'
'iso-8859-5'	'cisolatincyrillic', 'cyrillic', 'iso-ir-144', 'iso8859-5', 'iso88595', 'iso_8859-5', 'iso_8859-5:1988'
'iso-8859-6'	'arabic', 'asmo-708', 'csiso88596e', 'csiso88596i', 'cisolatinarabic', 'ecma-114', 'iso-8859-6-e', 'iso-8859-6-i', 'iso-ir-127', 'iso8859-6', 'iso88596', 'iso_8859-6', 'iso_8859-6:1987'
'iso-8859-7'	'cisolatingreek', 'ecma-118', 'elot_928', 'greek', 'greek8', 'iso-ir-126', 'iso8859-7', 'iso88597', 'iso_8859-7', 'iso_8859-7:1987', 'sun_eu_greek'
'iso-8859-8'	'csiso88598e', 'cisolatinhebrew', 'hebrew', 'iso-8859-8-e', 'iso-ir-138', 'iso8859-8', 'iso88598', 'iso_8859-8', 'iso_8859-8:1988', 'visual'
'iso-8859-8-i'	'csiso88598i', 'logical'

Encoding	Aliases
'iso-8859-10'	'csisolatin6', 'iso-ir-157', 'iso8859-10', 'iso885910', 'l6', 'latin6'
'iso-8859-13'	'iso8859-13', 'iso885913'
'iso-8859-14'	'iso8859-14', 'iso885914'
'iso-8859-15'	'csisolatin9', 'iso8859-15', 'iso885915', 'iso_8859-15', 'l9'
'koi8-r'	'cskoi8r', 'koi', 'koi8', 'koi8_r'
'koi8-u'	'koi8-ru'
'macintosh'	'csmacintosh', 'mac', 'x-mac-roman'
'windows-874'	'dos-874', 'iso-8859-11', 'iso8859-11', 'iso885911', 'tis-620'
'windows-1250'	'cp1250', 'x-cp1250'
'windows-1251'	'cp1251', 'x-cp1251'
'windows-1252'	'ansi_x3.4-1968', 'ascii', 'cp1252', 'cp819', 'csisolatin1', 'ibm819', 'iso-8859-1', 'iso-ir-100', 'iso8859-1', 'iso88591', 'iso_8859-1', 'iso_8859-1:1987', 'l1', 'latin1', 'us-ascii', 'x-cp1252'
'windows-1253'	'cp1253', 'x-cp1253'
'windows-1254'	'cp1254', 'csisolatin5', 'iso-8859-9', 'iso-ir-148', 'iso8859-9', 'iso88599', 'iso_8859-9', 'iso_8859-9:1989', 'l5', 'latin5', 'x-cp1254'
'windows-1255'	'cp1255', 'x-cp1255'
'windows-1256'	'cp1256', 'x-cp1256'
'windows-1257'	'cp1257', 'x-cp1257'
'windows-1258'	'cp1258', 'x-cp1258'

Encoding	Aliases
'x-mac-cyrillic'	'x-mac-ukrainian'
'gbk'	'chinese', 'csgb2312', 'csiso58gb231280', 'gb2312', 'gb_2312', 'gb_2312-80', 'iso-ir-58', 'x-gbk'
'gb18030'	
'big5'	'big5-hkscs', 'cn-big5', 'csbig5', 'x-x-big5'
'euc-jp'	'cseucpkdfmtjapanese', 'x-euc-jp'
'iso-2022-jp'	'csiso2022jp'
'shift_jis'	'csshiftjis', 'ms932', 'ms_kanji', 'shift-jis', 'sjis', 'windows-31j', 'x-sjis'
'euc-kr'	'cseuckr', 'csksc56011987', 'iso-ir-149', 'korean', 'ks_c_5601-1987', 'ks_c_5601-1989', 'ksc5601', 'ksc_5601', 'windows-949'

Encodings supported when Node.js is built with the `small-icu` option

Encoding	Aliases
'utf-8'	'unicode-1-1-utf-8', 'utf8'
'utf-16le'	'utf-16'
'utf-16be'	

Encodings supported when ICU is disabled

Encoding	Aliases
'utf-8'	'unicode-1-1-utf-8', 'utf8'
'utf-16le'	'utf-16'

The `'iso-8859-16'` encoding listed in the [WHATWG Encoding Standard](#) is not supported.

`new TextDecoder([encoding[, options]])`

- `encoding` `<string>` Identifies the encoding that this `TextDecoder` instance supports. **Default:** `'utf-8'`.
- `options` `<Object>`
 - `fatal` `<boolean>` `true` if decoding failures are fatal. This option is not supported when ICU is disabled (see [Internationalization](#)). **Default:** `false`.
 - `ignoreBOM` `<boolean>` When `true`, the `TextDecoder` will include the byte order mark in the decoded result. When `false`, the byte order mark will be removed from the output. This option is only used when `encoding` is `'utf-8'`, `'utf-16be'` or `'utf-16le'`. **Default:** `false`.

Creates a new `TextDecoder` instance. The `encoding` may specify one of the supported encodings or an alias.

The `TextDecoder` class is also available on the global object.

`textDecoder.decode([input[, options]])`

- `input` `<ArrayBuffer>` | `<DataView>` | `<TypedArray>` An `ArrayBuffer`, `DataView` or `TypedArray` instance containing the encoded data.
- `options` `<Object>`
 - `stream` `<boolean>` `true` if additional chunks of data are expected. **Default:** `false`.
- Returns: `<string>`

Decodes the `input` and returns a string. If `options.stream` is `true`, any incomplete byte sequences occurring at the end of the `input` are buffered internally and emitted after the next call to `textDecoder.decode()`.

If `textDecoder.fatal` is `true`, decoding errors that occur will result in a `TypeError` being thrown.

`textDecoder.encoding`

- `<string>`

The encoding supported by the `TextDecoder` instance.

`textDecoder.fatal`

- `<boolean>`

The value will be `true` if decoding errors result in a `TypeError` being thrown.

`textDecoder.ignoreBOM`

- `<boolean>`

The value will be `true` if the decoding result will include the byte order mark.

Class: `util.TextEncoder`

An implementation of the [WHATWG Encoding Standard](#) `TextEncoder` API. All instances of `TextEncoder` only support UTF-8 encoding.

```
const encoder = new TextEncoder();
const uint8array = encoder.encode('this is some data');
```

The `TextEncoder` class is also available on the global object.

`textEncoder.encode([input])`

- `input` `<string>` The text to encode. **Default:** an empty string.
- Returns: `<Uint8Array>`

UTF-8 encodes the `input` string and returns a `Uint8Array` containing the encoded bytes.

`textEncoder.encodeInto(src, dest)`

- `src` `<string>` The text to encode.
- `dest` `<Uint8Array>` The array to hold the encode result.
- Returns: `<Object>`
 - `read` `<number>` The read Unicode code units of `src`.

- `written` `<number>` The written UTF-8 bytes of `dest`.

UTF-8 encodes the `src` string to the `dest` `Uint8Array` and returns an object containing the read Unicode code units and written UTF-8 bytes.

```
const encoder = new TextEncoder();
const src = 'this is some data';
const dest = new Uint8Array(10);
const { read, written } = encoder.encodeInto(src, dest);
```

textEncoder.encoding

- `<string>`

The encoding supported by the `TextEncoder` instance. Always set to `'utf-8'`.

util.types

`util.types` provides type checks for different kinds of built-in objects. Unlike `instanceof` or `Object.prototype.toString.call(value)`, these checks do not inspect properties of the object that are accessible from JavaScript (like their prototype), and usually have the overhead of calling into C++.

The result generally does not make any guarantees about what kinds of properties or behavior a value exposes in JavaScript. They are primarily useful for addon developers who prefer to do type checking in JavaScript.

The API is accessible via `require('util').types` or `require('util/types')`.

util.types.isAnyArrayBuffer(value)

- `value` `<any>`
- Returns: `<boolean>`

Returns `true` if the value is a built-in `ArrayBuffer` or `SharedArrayBuffer` instance.

See also `util.types.isArrayBuffer()` and `util.types.isSharedArrayBuffer()`.

```
util.types.isAnyArrayBuffer(new ArrayBuffer()); // Returns true
util.types.isAnyArrayBuffer(new SharedArrayBuffer()); // Returns true
```

util.types.isArrayBufferView(value)

- `value` `<any>`
- Returns: `<boolean>`

Returns `true` if the value is an instance of one of the `ArrayBuffer` views, such as typed array objects or `DataView`. Equivalent to `ArrayBuffer.isView()`.

```
util.types.isArrayBufferView(new Int8Array()); // true
util.types.isArrayBufferView(Buffer.from('hello world')); // true
util.types.isArrayBufferView(new DataView(new ArrayBuffer(16))); // true
util.types.isArrayBufferView(new ArrayBuffer()); // false
```

util.types.isArgumentsObject(value)

- `value` `<any>`
- Returns: `<boolean>`

Returns `true` if the value is an `arguments` object.

```
function foo() {  
  util.types.isArgumentsObject(arguments); // Returns true  
}
```

util.types.isArrayBuffer(value)

- `value` `<any>`
- Returns: `<boolean>`

Returns `true` if the value is a built-in `ArrayBuffer` instance. This does *not* include `SharedArrayBuffer` instances. Usually, it is desirable to test for both; See `util.types.isAnyArrayBuffer()` for that.

```
util.types.isArrayBuffer(new ArrayBuffer()); // Returns true  
util.types.isArrayBuffer(new SharedArrayBuffer()); // Returns false
```

util.types.isAsyncFunction(value)

- `value` `<any>`
- Returns: `<boolean>`

Returns `true` if the value is an `async function`. This only reports back what the JavaScript engine is seeing; in particular, the return value may not match the original source code if a transpilation tool was used.

```
util.types.isAsyncFunction(function foo() {}); // Returns false  
util.types.isAsyncFunction(async function foo() {}); // Returns true
```

util.types.isBigInt64Array(value)

- `value` `<any>`
- Returns: `<boolean>`

Returns `true` if the value is a `BigInt64Array` instance.

```
util.types.isBigInt64Array(new BigInt64Array()); // Returns true  
util.types.isBigInt64Array(new BigUint64Array()); // Returns false
```

util.types.isBigUint64Array(value)

- `value` `<any>`
- Returns: `<boolean>`

Returns `true` if the value is a `BigUint64Array` instance.


```
util.types.isBigUint64Array(new BigInt64Array()); // Returns false
util.types.isBigUint64Array(new BigUint64Array()); // Returns true
```

util.types.isBooleanObject(value)

- `value` `<any>`
- Returns: `<boolean>`

Returns `true` if the value is a boolean object, e.g. created by `new Boolean()`.

```
util.types.isBooleanObject(false); // Returns false
util.types.isBooleanObject(true); // Returns false
util.types.isBooleanObject(new Boolean(false)); // Returns true
util.types.isBooleanObject(new Boolean(true)); // Returns true
util.types.isBooleanObject(Boolean(false)); // Returns false
util.types.isBooleanObject(Boolean(true)); // Returns false
```

util.types.isBoxedPrimitive(value)

- `value` `<any>`
- Returns: `<boolean>`

Returns `true` if the value is any boxed primitive object, e.g. created by `new Boolean()`, `new String()` or `Object(Symbol())`.

For example:

```
util.types.isBoxedPrimitive(false); // Returns false
util.types.isBoxedPrimitive(new Boolean(false)); // Returns true
util.types.isBoxedPrimitive(Symbol('foo')); // Returns false
util.types.isBoxedPrimitive(Object(Symbol('foo'))); // Returns true
util.types.isBoxedPrimitive(Object(BigInt(5))); // Returns true
```

util.types.isCryptoKey(value)

- `value` `<Object>`
- Returns: `<boolean>`

Returns `true` if `value` is a `<CryptoKey>`, `false` otherwise.

util.types.isDataView(value)

- `value` `<any>`
- Returns: `<boolean>`

Returns `true` if the value is a built-in `DataView` instance.

```
const ab = new ArrayBuffer(20);
util.types.isDataView(new DataView(ab)); // Returns true
util.types.isDataView(new Float64Array()); // Returns false
```

See also `ArrayBuffer.isView()`.

util.types.isDate(value)

- `value` `<any>`
- Returns: `<boolean>`

Returns `true` if the value is a built-in `Date` instance.

```
util.types.isDate(new Date()); // Returns true
```

util.types.isExternal(value)

- `value` `<any>`
- Returns: `<boolean>`

Returns `true` if the value is a native `External` value.

A native `External` value is a special type of object that contains a raw C++ pointer (`void*`) for access from native code, and has no other properties. Such objects are created either by Node.js internals or native addons. In JavaScript, they are `frozen` objects with a `null` prototype.

```
#include <js_native_api.h>
#include <stdlib.h>

napi_value result;

static napi_value MyNapi(napi_env env, napi_callback_info info) {
    int* raw = (int*) malloc(1024);
    napi_status status = napi_create_external(env, (void*) raw, NULL, NULL, &result);
    if (status != napi_ok) {
        napi_throw_error(env, NULL, "napi_create_external failed");
        return NULL;
    }
    return result;
}

...
DECLARE_NAPI_PROPERTY("myNapi", MyNapi)
...
```

```
const native = require('napi_addon.node');
const data = native.myNapi();
util.types.isExternal(data); // returns true
util.types.isExternal(0); // returns false
util.types.isExternal(new String('foo')); // returns false
```

For further information on `napi_create_external`, refer to `napi_create_external()`.

util.types.isFloat32Array(value)

- `value` `<any>`
- Returns: `<boolean>`

Returns `true` if the value is a built-in `Float32Array` instance.

```
util.types.isFloat32Array(new ArrayBuffer()); // Returns false
util.types.isFloat32Array(new Float32Array()); // Returns true
util.types.isFloat32Array(new Float64Array()); // Returns false
```

util.types.isFloat64Array(value)

- `value` `<any>`
- Returns: `<boolean>`

Returns `true` if the value is a built-in `Float64Array` instance.

```
util.types.isFloat64Array(new ArrayBuffer()); // Returns false
util.types.isFloat64Array(new Uint8Array()); // Returns false
util.types.isFloat64Array(new Float64Array()); // Returns true
```

util.types.isGeneratorFunction(value)

- `value` `<any>`
- Returns: `<boolean>`

Returns `true` if the value is a generator function. This only reports back what the JavaScript engine is seeing; in particular, the return value may not match the original source code if a transpilation tool was used.

```
util.types.isGeneratorFunction(function foo() {}); // Returns false
util.types.isGeneratorFunction(function* foo() {}); // Returns true
```

util.types.isGeneratorObject(value)

- `value` `<any>`
- Returns: `<boolean>`

Returns `true` if the value is a generator object as returned from a built-in generator function. This only reports back what the JavaScript engine is seeing; in particular, the return value may not match the original source code if a transpilation tool was used.

```
function* foo() {}
const generator = foo();
util.types.isGeneratorObject(generator); // Returns true
```

util.types.isInt8Array(value)

- `value` `<any>`
- Returns: `<boolean>`

Returns `true` if the value is a built-in `Int8Array` instance.

```
util.types.isInt8Array(new ArrayBuffer()); // Returns false
util.types.isInt8Array(new Int8Array()); // Returns true
util.types.isInt8Array(new Float64Array()); // Returns false
```

util.types.isInt16Array(value)

- value `<any>`
- Returns: `<boolean>`

Returns `true` if the value is a built-in `Int16Array` instance.

```
util.types.isInt16Array(new ArrayBuffer()); // Returns false
util.types.isInt16Array(new Int16Array()); // Returns true
util.types.isInt16Array(new Float64Array()); // Returns false
```

util.types.isInt32Array(value)

- value `<any>`
- Returns: `<boolean>`

Returns `true` if the value is a built-in `Int32Array` instance.

```
util.types.isInt32Array(new ArrayBuffer()); // Returns false
util.types.isInt32Array(new Int32Array()); // Returns true
util.types.isInt32Array(new Float64Array()); // Returns false
```

util.types.isKeyObject(value)

- value `<Object>`
- Returns: `<boolean>`

Returns `true` if value is a `<KeyObject>`, `false` otherwise.

util.types.isMap(value)

- value `<any>`
- Returns: `<boolean>`

Returns `true` if the value is a built-in `Map` instance.

```
util.types.isMap(new Map()); // Returns true
```

util.types.isMapIterator(value)

- value `<any>`
- Returns: `<boolean>`

Returns `true` if the value is an iterator returned for a built-in `Map` instance.

```
const map = new Map();
util.types.isMapIterator(map.keys()); // Returns true
util.types.isMapIterator(map.values()); // Returns true
util.types.isMapIterator(map.entries()); // Returns true
util.types.isMapIterator(map[Symbol.iterator]()); // Returns true
```

util.types.isModuleNamespaceObject(value)

- `value` `<any>`
- Returns: `<boolean>`

Returns `true` if the value is an instance of a `Module Namespace Object`.

```
import * as ns from './a.js';

util.types.isModuleNamespaceObject(ns); // Returns true
```

util.types.isNativeError(value)

- `value` `<any>`
- Returns: `<boolean>`

Returns `true` if the value is an instance of a built-in `Error` type.

```
util.types.isNativeError(new Error()); // Returns true
util.types.isNativeError(new TypeError()); // Returns true
util.types.isNativeError(new RangeError()); // Returns true
```

util.types.isNumberObject(value)

- `value` `<any>`
- Returns: `<boolean>`

Returns `true` if the value is a number object, e.g. created by `new Number()`.

```
util.types.isNumberObject(0); // Returns false
util.types.isNumberObject(new Number(0)); // Returns true
```

util.types.isPromise(value)

- `value` `<any>`
- Returns: `<boolean>`

Returns `true` if the value is a built-in `Promise`.

```
util.types.isPromise(Promise.resolve(42)); // Returns true
```

util.types.isProxy(value)

- `value` `<any>`
- Returns: `<boolean>`

Returns `true` if the value is a `Proxy` instance.

```
const target = {};
const proxy = new Proxy(target, {});
```

```
util.types.isProxy(target); // Returns false
util.types.isProxy(proxy); // Returns true
```

util.types.isRegExp(value)

- `value` `<any>`
- Returns: `<boolean>`

Returns `true` if the value is a regular expression object.

```
util.types.isRegExp(/abc/); // Returns true
util.types.isRegExp(new RegExp('abc')); // Returns true
```

util.types.isSet(value)

- `value` `<any>`
- Returns: `<boolean>`

Returns `true` if the value is a built-in `Set` instance.

```
util.types.isSet(new Set()); // Returns true
```

util.types.isSetIterator(value)

- `value` `<any>`
- Returns: `<boolean>`

Returns `true` if the value is an iterator returned for a built-in `Set` instance.

```
const set = new Set();
util.types.isSetIterator(set.keys()); // Returns true
util.types.isSetIterator(set.values()); // Returns true
util.types.isSetIterator(set.entries()); // Returns true
util.types.isSetIterator(set[Symbol.iterator]()); // Returns true
```

util.types.isSharedArrayBuffer(value)

- `value` `<any>`
- Returns: `<boolean>`

Returns `true` if the value is a built-in `SharedArrayBuffer` instance. This does *not* include `ArrayBuffer` instances. Usually, it is desirable to test for both; See `util.types.isAnyArrayBuffer()` for that.

```
util.types.isSharedArrayBuffer(new ArrayBuffer()); // Returns false
util.types.isSharedArrayBuffer(new SharedArrayBuffer()); // Returns true
```

util.types.isStringObject(value)

- `value` `<any>`
- Returns: `<boolean>`

Returns `true` if the value is a string object, e.g. created by `new String()`.

```
util.types.isStringObject('foo'); // Returns false
util.types.isStringObject(new String('foo')); // Returns true
```

util.types.isSymbolObject(value)

- `value` `<any>`
- Returns: `<boolean>`

Returns `true` if the value is a symbol object, created by calling `Object()` on a `Symbol` primitive.

```
const symbol = Symbol('foo');
util.types.isSymbolObject(symbol); // Returns false
util.types.isSymbolObject(Object(symbol)); // Returns true
```

util.types.isTypedArray(value)

- `value` `<any>`
- Returns: `<boolean>`

Returns `true` if the value is a built-in `TypedArray` instance.

```
util.types.isTypedArray(new ArrayBuffer()); // Returns false
util.types.isTypedArray(new Uint8Array()); // Returns true
util.types.isTypedArray(new Float64Array()); // Returns true
```

See also `ArrayBuffer.isView()`.

util.types.isUint8Array(value)

- `value` `<any>`
- Returns: `<boolean>`

Returns `true` if the value is a built-in `Uint8Array` instance.

```
util.types.isUint8Array(new ArrayBuffer()); // Returns false
util.types.isUint8Array(new Uint8Array()); // Returns true
util.types.isUint8Array(new Float64Array()); // Returns false
```

util.types.isUint8ClampedArray(value)

- `value` `<any>`
- Returns: `<boolean>`

Returns `true` if the value is a built-in `Uint8ClampedArray` instance.

```
util.types.isUint8ClampedArray(new ArrayBuffer()); // Returns false
util.types.isUint8ClampedArray(new Uint8ClampedArray()); // Returns true
util.types.isUint8ClampedArray(new Float64Array()); // Returns false
```

util.types.isUint16Array(value)

- `value` `<any>`
- Returns: `<boolean>`

Returns `true` if the value is a built-in `Uint16Array` instance.

```
util.types.isUint16Array(new ArrayBuffer()); // Returns false
util.types.isUint16Array(new Uint16Array()); // Returns true
util.types.isUint16Array(new Float64Array()); // Returns false
```

util.types.isUint32Array(value)

- `value` `<any>`
- Returns: `<boolean>`

Returns `true` if the value is a built-in `Uint32Array` instance.

```
util.types.isUint32Array(new ArrayBuffer()); // Returns false
util.types.isUint32Array(new Uint32Array()); // Returns true
util.types.isUint32Array(new Float64Array()); // Returns false
```

util.types.isWeakMap(value)

- `value` `<any>`
- Returns: `<boolean>`

Returns `true` if the value is a built-in `WeakMap` instance.

```
util.types.isWeakMap(new WeakMap()); // Returns true
```

util.types.isWeakSet(value)

- `value` `<any>`
- Returns: `<boolean>`

Returns `true` if the value is a built-in `WeakSet` instance.

```
util.types.isWeakSet(new WeakSet()); // Returns true
```

util.types.isWebAssemblyCompiledModule(value)

Stability: 0 - Deprecated: Use `value instanceof WebAssembly.Module` instead.

- `value` `<any>`
- Returns: `<boolean>`

Returns `true` if the value is a built-in `WebAssembly.Module` instance.


```
const module = new WebAssembly.Module(wasmBuffer);
util.types.isWebAssemblyCompiledModule(module); // Returns true
```

Deprecated APIs

The following APIs are deprecated and should no longer be used. Existing applications and modules should be updated to find alternative approaches.

util._extend(target, source)

Stability: 0 - Deprecated: Use `Object.assign()` instead.

- `target` `<Object>`
- `source` `<Object>`

The `util._extend()` method was never intended to be used outside of internal Node.js modules. The community found and used it anyway.

It is deprecated and should not be used in new code. JavaScript comes with very similar built-in functionality through `Object.assign()`.

util.isArray(object)

Stability: 0 - Deprecated: Use `Array.isArray()` instead.

- `object` `<any>`
- Returns: `<boolean>`

Alias for `Array.isArray()`.

Returns `true` if the given `object` is an `Array`. Otherwise, returns `false`.

```
const util = require('util');

util.isArray([]);
// Returns: true
util.isArray(new Array());
// Returns: true
util.isArray({});
// Returns: false
```

util.isBoolean(object)

Stability: 0 - Deprecated: Use `typeof value === 'boolean'` instead.

- `object` `<any>`
- Returns: `<boolean>`

Returns `true` if the given `object` is a `Boolean`. Otherwise, returns `false`.

```
const util = require('util');

util.isBoolean(1);
// Returns: false
util.isBoolean(0);
// Returns: false
util.isBoolean(false);
// Returns: true
```

util.isBuffer(object)

Stability: 0 - Deprecated: Use `Buffer.isBuffer()` instead.

- `object` `<any>`
- Returns: `<boolean>`

Returns `true` if the given `object` is a `Buffer`. Otherwise, returns `false`.

```
const util = require('util');

util.isBuffer({ length: 0 });
// Returns: false
util.isBuffer([]);
// Returns: false
util.isBuffer(Buffer.from('hello world'));
// Returns: true
```

util.isDate(object)

Stability: 0 - Deprecated: Use `util.types.isDate()` instead.

- `object` `<any>`
- Returns: `<boolean>`

Returns `true` if the given `object` is a `Date`. Otherwise, returns `false`.

```
const util = require('util');

util.isDate(new Date());
// Returns: true
util.isDate(Date());
// false (without 'new' returns a String)
util.isDate({});
// Returns: false
```

util.isError(object)

Stability: 0 - Deprecated: Use `util.types.isNativeError()` instead.

- `object` `<any>`
- Returns: `<boolean>`

Returns `true` if the given `object` is an `Error`. Otherwise, returns `false`.

```
const util = require('util');

util.isError(new Error());
// Returns: true
util.isError(new TypeError());
// Returns: true
util.isError({ name: 'Error', message: 'an error occurred' });
// Returns: false
```

This method relies on `Object.prototype.toString()` behavior. It is possible to obtain an incorrect result when the `object` argument manipulates `@@toStringTag`.

```
const util = require('util');
const obj = { name: 'Error', message: 'an error occurred' };

util.isError(obj);
// Returns: false
obj[Symbol.toStringTag] = 'Error';
util.isError(obj);
// Returns: true
```

util.isFunction(object)

Stability: 0 - Deprecated: Use `typeof value === 'function'` instead.

- `object` `<any>`
- Returns: `<boolean>`

Returns `true` if the given `object` is a `Function`. Otherwise, returns `false`.

```
const util = require('util');

function Foo() {}
const Bar = () => {};

util.isFunction({});
// Returns: false
util.isFunction(Foo);
// Returns: true
```

```
util.isFunction(Bar);  
// Returns: true
```

util.isNull(object)

Stability: 0 - Deprecated: Use `value === null` instead.

- `object` `<any>`
- Returns: `<boolean>`

Returns `true` if the given `object` is strictly `null`. Otherwise, returns `false`.

```
const util = require('util');  
  
util.isNull(0);  
// Returns: false  
util.isNull(undefined);  
// Returns: false  
util.isNull(null);  
// Returns: true
```

util.isNullOrUndefined(object)

Stability: 0 - Deprecated: Use `value === undefined || value === null` instead.

- `object` `<any>`
- Returns: `<boolean>`

Returns `true` if the given `object` is `null` or `undefined`. Otherwise, returns `false`.

```
const util = require('util');  
  
util.isNullOrUndefined(0);  
// Returns: false  
util.isNullOrUndefined(undefined);  
// Returns: true  
util.isNullOrUndefined(null);  
// Returns: true
```

util.isNumber(object)

Stability: 0 - Deprecated: Use `typeof value === 'number'` instead.

- `object` `<any>`
- Returns: `<boolean>`

Returns `true` if the given `object` is a `Number`. Otherwise, returns `false`.

```
const util = require('util');

util.isNumber(false);
// Returns: false
util.isNumber(Infinity);
// Returns: true
util.isNumber(0);
// Returns: true
util.isNumber(NaN);
// Returns: true
```

util.isObject(object)

Stability: 0 - Deprecated: Use `value !== null && typeof value === 'object'` instead.

- `object` `<any>`
- Returns: `<boolean>`

Returns `true` if the given `object` is strictly an `Object` and not a `Function` (even though functions are objects in JavaScript). Otherwise, returns `false`.

```
const util = require('util');

util.isObject(5);
// Returns: false
util.isObject(null);
// Returns: false
util.isObject({});
// Returns: true
util.isObject(() => {});
// Returns: false
```

util.isPrimitive(object)

Stability: 0 - Deprecated: Use `(typeof value !== 'object' && typeof value !== 'function') || value === null` instead.

- `object` `<any>`
- Returns: `<boolean>`

Returns `true` if the given `object` is a primitive type. Otherwise, returns `false`.

```
const util = require('util');

util.isPrimitive(5);
// Returns: true
```

```
util.isPrimitive('foo');
// Returns: true
util.isPrimitive(false);
// Returns: true
util.isPrimitive(null);
// Returns: true
util.isPrimitive(undefined);
// Returns: true
util.isPrimitive({});
// Returns: false
util.isPrimitive(() => {});
// Returns: false
util.isPrimitive(/^$/);
// Returns: false
util.isPrimitive(new Date());
// Returns: false
```

util.isRegExp(object)

Stability: 0 - Deprecated

- `object` `<any>`
- Returns: `<boolean>`

Returns `true` if the given `object` is a `RegExp`. Otherwise, returns `false`.

```
const util = require('util');

util.isRegExp(/some regexp/);
// Returns: true
util.isRegExp(new RegExp('another regexp'));
// Returns: true
util.isRegExp({});
// Returns: false
```

util.isString(object)

Stability: 0 - Deprecated: Use `typeof value === 'string'` instead.

- `object` `<any>`
- Returns: `<boolean>`

Returns `true` if the given `object` is a `string`. Otherwise, returns `false`.

```
const util = require('util');

util.isString('');
```

```
// Returns: true
util.isString('foo');
// Returns: true
util.isString(String('foo'));
// Returns: true
util.isString(5);
// Returns: false
```

util.isSymbol(object)

Stability: 0 - Deprecated: Use `typeof value === 'symbol'` instead.

- `object` `<any>`
- Returns: `<boolean>`

Returns `true` if the given `object` is a `Symbol`. Otherwise, returns `false`.

```
const util = require('util');

util.isSymbol(5);
// Returns: false
util.isSymbol('foo');
// Returns: false
util.isSymbol(Symbol('foo'));
// Returns: true
```

util.isUndefined(object)

Stability: 0 - Deprecated: Use `value === undefined` instead.

- `object` `<any>`
- Returns: `<boolean>`

Returns `true` if the given `object` is `undefined`. Otherwise, returns `false`.

```
const util = require('util');

const foo = undefined;
util.isUndefined(5);
// Returns: false
util.isUndefined(foo);
// Returns: true
util.isUndefined(null);
// Returns: false
```

util.log(string)

Stability: 0 - Deprecated: Use a third party module instead.

- `string` `<string>`

The `util.log()` method prints the given `string` to `stdout` with an included timestamp.

```
const util = require('util');

util.log('Timestamped message.');
```

V8

Source Code: [lib/v8.js](#)

The `v8` module exposes APIs that are specific to the version of `V8` built into the Node.js binary. It can be accessed using:

```
const v8 = require('v8');
```

`v8.cachedDataVersionTag()`

- Returns: `<integer>`

Returns an integer representing a version tag derived from the V8 version, command-line flags, and detected CPU features. This is useful for determining whether a `vm.Script` `cachedData` buffer is compatible with this instance of V8.

```
console.log(v8.cachedDataVersionTag()); // 3947234607
// The value returned by v8.cachedDataVersionTag() is derived from the V8
// version, command-line flags, and detected CPU features. Test that the value
// does indeed update when flags are toggled.
v8.setFlagsFromString('--allow_natives_syntax');
console.log(v8.cachedDataVersionTag()); // 183726201
```

`v8.getHeapCodeStatistics()`

- Returns: `<Object>`

Returns an object with the following properties:

- `code_and_metadata_size` `<number>`
- `bytecode_and_metadata_size` `<number>`
- `external_script_source_size` `<number>`

```
{
  code_and_metadata_size: 212208,
  bytecode_and_metadata_size: 161368,
  external_script_source_size: 1410794
}
```


v8.getHeapSnapshot()

- Returns: `<stream.Readable>` A Readable Stream containing the V8 heap snapshot

Generates a snapshot of the current V8 heap and returns a Readable Stream that may be used to read the JSON serialized representation. This JSON stream format is intended to be used with tools such as Chrome DevTools. The JSON schema is undocumented and specific to the V8 engine. Therefore, the schema may change from one version of V8 to the next.

```
// Print heap snapshot to the console
const v8 = require('v8');
const stream = v8.getHeapSnapshot();
stream.pipe(process.stdout);
```

v8.getHeapSpaceStatistics()

- Returns: `<Object[]>`

Returns statistics about the V8 heap spaces, i.e. the segments which make up the V8 heap. Neither the ordering of heap spaces, nor the availability of a heap space can be guaranteed as the statistics are provided via the V8 `GetHeapSpaceStatistics` function and may change from one V8 version to the next.

The value returned is an array of objects containing the following properties:

- `space_name` `<string>`
- `space_size` `<number>`
- `space_used_size` `<number>`
- `space_available_size` `<number>`
- `physical_space_size` `<number>`

```
[
  {
    "space_name": "new_space",
    "space_size": 2063872,
    "space_used_size": 951112,
    "space_available_size": 80824,
    "physical_space_size": 2063872
  },
  {
    "space_name": "old_space",
    "space_size": 3090560,
    "space_used_size": 2493792,
    "space_available_size": 0,
    "physical_space_size": 3090560
  },
  {
    "space_name": "code_space",
    "space_size": 1260160,
    "space_used_size": 644256,
    "space_available_size": 960,
    "physical_space_size": 1260160
  }
]
```

```

},
{
  "space_name": "map_space",
  "space_size": 1094160,
  "space_used_size": 201608,
  "space_available_size": 0,
  "physical_space_size": 1094160
},
{
  "space_name": "large_object_space",
  "space_size": 0,
  "space_used_size": 0,
  "space_available_size": 1490980608,
  "physical_space_size": 0
}
]

```

v8.getHeapStatistics()

- Returns: `<Object>`

Returns an object with the following properties:

- `total_heap_size` `<number>`
- `total_heap_size_executable` `<number>`
- `total_physical_size` `<number>`
- `total_available_size` `<number>`
- `used_heap_size` `<number>`
- `heap_size_limit` `<number>`
- `malloced_memory` `<number>`
- `peak_malloced_memory` `<number>`
- `does_zap_garbage` `<number>`
- `number_of_native_contexts` `<number>`
- `number_of_detached_contexts` `<number>`

`does_zap_garbage` is a 0/1 boolean, which signifies whether the `--zap_code_space` option is enabled or not. This makes V8 overwrite heap garbage with a bit pattern. The RSS footprint (resident set size) gets bigger because it continuously touches all heap pages and that makes them less likely to get swapped out by the operating system.

`number_of_native_contexts` The value of `native_context` is the number of the top-level contexts currently active. Increase of this number over time indicates a memory leak.

`number_of_detached_contexts` The value of `detached_context` is the number of contexts that were detached and not yet garbage collected. This number being non-zero indicates a potential memory leak.

```

{
  total_heap_size: 7326976,
  total_heap_size_executable: 4194304,
  total_physical_size: 7326976,
  total_available_size: 1152656,

```

```

used_heap_size: 3476208,
heap_size_limit: 1535115264,
malloced_memory: 16384,
peak_malloced_memory: 1127496,
does_zap_garbage: 0,
number_of_native_contexts: 1,
number_of_detached_contexts: 0
}

```

v8.setFlagsFromString(flags)

- `flags` `<string>`

The `v8.setFlagsFromString()` method can be used to programmatically set V8 command-line flags. This method should be used with care. Changing settings after the VM has started may result in unpredictable behavior, including crashes and data loss; or it may simply do nothing.

The V8 options available for a version of Node.js may be determined by running `node --v8-options`.

Usage:

```

// Print GC events to stdout for one minute.
const v8 = require('v8');
v8.setFlagsFromString('--trace_gc');
setTimeout(() => { v8.setFlagsFromString('--notrace_gc'); }, 60e3);

```

v8.stopCoverage()

The `v8.stopCoverage()` method allows the user to stop the coverage collection started by `NODE_V8_COVERAGE`, so that V8 can release the execution count records and optimize code. This can be used in conjunction with `v8.takeCoverage()` if the user wants to collect the coverage on demand.

v8.takeCoverage()

The `v8.takeCoverage()` method allows the user to write the coverage started by `NODE_V8_COVERAGE` to disk on demand. This method can be invoked multiple times during the lifetime of the process. Each time the execution counter will be reset and a new coverage report will be written to the directory specified by `NODE_V8_COVERAGE`.

When the process is about to exit, one last coverage will still be written to disk unless `v8.stopCoverage()` is invoked before the process exits.

v8.writeHeapSnapshot([filename])

- `filename` `<string>` The file path where the V8 heap snapshot is to be saved. If not specified, a file name with the pattern `'Heap-${yyyyymmdd}-${hhmmss}-${pid}-${thread_id}.heapsnapshot'` will be generated, where `{pid}` will be the PID of the Node.js process, `{thread_id}` will be `0` when `writeHeapSnapshot()` is called from the main Node.js thread or the id of a worker thread.
- Returns: `<string>` The filename where the snapshot was saved.

Generates a snapshot of the current V8 heap and writes it to a JSON file. This file is intended to be used with tools such as Chrome DevTools. The JSON schema is undocumented and specific to the V8 engine, and may change from one version of V8 to the next.

A heap snapshot is specific to a single V8 isolate. When using [worker threads](#) , a heap snapshot generated from the main thread will not contain any information about the workers, and vice versa.

```
const { writeHeapSnapshot } = require('v8');
const {
  Worker,
  isMainThread,
  parentPort
} = require('worker_threads');

if (isMainThread) {
  const worker = new Worker(__filename);

  worker.once('message', (filename) => {
    console.log(`worker heapdump: ${filename}`);
    // Now get a heapdump for the main thread.
    console.log(`main thread heapdump: ${writeHeapSnapshot()}`);
  });

  // Tell the worker to create a heapdump.
  worker.postMessage('heapdump');
} else {
  parentPort.once('message', (message) => {
    if (message === 'heapdump') {
      // Generate a heapdump for the worker
      // and return the filename to the parent.
      parentPort.postMessage(writeHeapSnapshot());
    }
  });
}
```

Serialization API

The serialization API provides means of serializing JavaScript values in a way that is compatible with the [HTML structured clone algorithm](#) .

The format is backward-compatible (i.e. safe to store to disk). Equal JavaScript values may result in different serialized output.

v8.serialize(value)

- `value` [<any>](#)
- Returns: [<Buffer>](#)

Uses a [DefaultSerializer](#) to serialize `value` into a buffer.

v8.deserialize(buffer)

- `buffer` [<Buffer>](#) | [<TypedArray>](#) | [<DataView>](#) A buffer returned by `serialize()` .

Uses a [DefaultDeserializer](#) with default options to read a JS value from a buffer.

Class: v8.Serializer

`new Serializer()`

Creates a new `Serializer` object.

`serializer.writeHeader()`

Writes out a header, which includes the serialization format version.

`serializer.writeValue(value)`

- `value` `<any>`

Serializes a JavaScript value and adds the serialized representation to the internal buffer.

This throws an error if `value` cannot be serialized.

`serializer.releaseBuffer()`

- Returns: `<Buffer>`

Returns the stored internal buffer. This serializer should not be used once the buffer is released. Calling this method results in undefined behavior if a previous write has failed.

`serializer.transferArrayBuffer(id, arrayBuffer)`

- `id` `<integer>` A 32-bit unsigned integer.
- `arrayBuffer` `<ArrayBuffer>` An `ArrayBuffer` instance.

Marks an `ArrayBuffer` as having its contents transferred out of band. Pass the corresponding `ArrayBuffer` in the deserializing context to `deserializer.transferArrayBuffer()`.

`serializer.writeUint32(value)`

- `value` `<integer>`

Write a raw 32-bit unsigned integer. For use inside of a custom `serializer._writeHostObject()`.

`serializer.writeUint64(hi, lo)`

- `hi` `<integer>`
- `lo` `<integer>`

Write a raw 64-bit unsigned integer, split into high and low 32-bit parts. For use inside of a custom `serializer._writeHostObject()`.

`serializer.writeDouble(value)`

- `value` `<number>`

Write a JS `number` value. For use inside of a custom `serializer._writeHostObject()`.

`serializer.writeRawBytes(buffer)`

- `buffer` `<Buffer>` | `<TypedArray>` | `<DataView>`

Write raw bytes into the serializer's internal buffer. The deserializer will require a way to compute the length of the buffer. For use inside of a custom `serializer._writeHostObject()`.

`serializer._writeHostObject(object)`

- `object` `<Object>`

This method is called to write some kind of host object, i.e. an object created by native C++ bindings. If it is not possible to serialize `object`, a suitable exception should be thrown.

This method is not present on the `Serializer` class itself but can be provided by subclasses.

`serializer._getDataCloneError(message)`

- `message` `<string>`

This method is called to generate error objects that will be thrown when an object can not be cloned.

This method defaults to the `Error` constructor and can be overridden on subclasses.

`serializer._getSharedArrayBufferId(sharedArrayBuffer)`

- `sharedArrayBuffer` `<SharedArrayBuffer>`

This method is called when the serializer is going to serialize a `SharedArrayBuffer` object. It must return an unsigned 32-bit integer ID for the object, using the same ID if this `SharedArrayBuffer` has already been serialized. When deserializing, this ID will be passed to `deserializer.transferArrayBuffer()`.

If the object cannot be serialized, an exception should be thrown.

This method is not present on the `Serializer` class itself but can be provided by subclasses.

`serializer._setTreatArrayBufferViewsAsHostObjects(flag)`

- `flag` `<boolean>` Default: `false`

Indicate whether to treat `TypedArray` and `DataView` objects as host objects, i.e. pass them to `serializer._writeHostObject()`.

Class: `v8.Deserializer`

`new Deserializer(buffer)`

- `buffer` `<Buffer>` | `<TypedArray>` | `<DataView>` A buffer returned by `serializer.releaseBuffer()`.

Creates a new `Deserializer` object.

`deserializer.readHeader()`

Reads and validates a header (including the format version). May, for example, reject an invalid or unsupported wire format. In that case, an `Error` is thrown.

`deserializer.readValue()`

Deserializes a JavaScript value from the buffer and returns it.

`deserializer.transferArrayBuffer(id, arrayBuffer)`

- `id` `<integer>` A 32-bit unsigned integer.
- `arrayBuffer` `<ArrayBuffer>` | `<SharedArrayBuffer>` An `ArrayBuffer` instance.

Marks an `ArrayBuffer` as having its contents transferred out of band. Pass the corresponding `ArrayBuffer` in the serializing context to `serializer.transferArrayBuffer()` (or return the `id` from `serializer._getSharedArrayBufferId()` in the case of `SharedArrayBuffer`s).

`deserializer.getWireFormatVersion()`

- Returns: `<integer>`

Reads the underlying wire format version. Likely mostly to be useful to legacy code reading old wire format versions. May not be called before `.readHeader()`.

`deserializer.readUint32()`

- Returns: `<integer>`

Read a raw 32-bit unsigned integer and return it. For use inside of a custom `deserializer._readHostObject()`.

`deserializer.readUint64()`

- Returns: `<integer[]>`

Read a raw 64-bit unsigned integer and return it as an array `[hi, lo]` with two 32-bit unsigned integer entries. For use inside of a custom `deserializer._readHostObject()`.

`deserializer.readDouble()`

- Returns: `<number>`

Read a JS `number` value. For use inside of a custom `deserializer._readHostObject()`.

`deserializer.readRawBytes(length)`

- `length` `<integer>`
- Returns: `<Buffer>`

Read raw bytes from the deserializer's internal buffer. The `length` parameter must correspond to the length of the buffer that was passed to `serializer.writeRawBytes()`. For use inside of a custom `deserializer._readHostObject()`.

`deserializer._readHostObject()`

This method is called to read some kind of host object, i.e. an object that is created by native C++ bindings. If it is not possible to deserialize the data, a suitable exception should be thrown.

This method is not present on the `Deserializer` class itself but can be provided by subclasses.

Class: `v8.DefaultSerializer`

A subclass of `Serializer` that serializes `TypedArray` (in particular `Buffer`) and `DataView` objects as host objects, and only stores the part of their underlying `ArrayBuffer`s that they are referring to.

Class: `v8.DefaultDeserializer`

A subclass of `Deserializer` corresponding to the format written by `DefaultSerializer`.

VM (executing JavaScript)

Stability: 2 - Stable

Source Code: [lib/vm.js](#)

The `vm` module enables compiling and running code within V8 Virtual Machine contexts. The `vm` module is not a security mechanism. Do not use it to run untrusted code.

JavaScript code can be compiled and run immediately or compiled, saved, and run later.

A common use case is to run the code in a different V8 Context. This means invoked code has a different global object than the invoking code.

One can provide the context by *contextifying* an object. The invoked code treats any property in the context like a global variable. Any changes to global variables caused by the invoked code are reflected in the context object.

```
const vm = require('vm');

const x = 1;

const context = { x: 2 };
vm.createContext(context); // Contextify the object.

const code = 'x += 40; var y = 17;';
// `x` and `y` are global variables in the context.
// Initially, x has the value 2 because that is the value of context.x.
vm.runInContext(code, context);

console.log(context.x); // 42
console.log(context.y); // 17

console.log(x); // 1; y is not defined.
```

Class: `vm.Script`

Instances of the `vm.Script` class contain precompiled scripts that can be executed in specific contexts.

`new vm.Script(code[, options])`

- `code` `<string>` The JavaScript code to compile.
- `options` `<Object>` | `<string>`
 - `filename` `<string>` Specifies the filename used in stack traces produced by this script. **Default:** `'evalmachine.<anonymous>'`.
 - `lineOffset` `<number>` Specifies the line number offset that is displayed in stack traces produced by this script. **Default:** `0`.
 - `columnOffset` `<number>` Specifies the first-line column number offset that is displayed in stack traces produced by this script. **Default:** `0`.
 - `cachedData` `<Buffer>` | `<TypedArray>` | `<DataView>` Provides an optional `Buffer` or `TypedArray`, or `DataView` with V8's code cache data for the supplied source. When supplied, the `cachedDataRejected` value will be set to either `true` or `false` depending on acceptance of the data by V8.
 - `produceCachedData` `<boolean>` When `true` and no `cachedData` is present, V8 will attempt to produce code cache data for `code`. Upon success, a `Buffer` with V8's code cache data will be produced and stored in the `cachedData` property of the returned `vm.Script` instance. The `cachedDataProduced` value will be set to either `true` or `false` depending on whether code cache data is produced successfully. This option is **deprecated** in favor of `script.createCachedData()`. **Default:** `false`.
 - `importModuleDynamically` `<Function>` Called during evaluation of this module when `import()` is called. If this option is not specified, calls to `import()` will reject with `ERR_VM_DYNAMIC_IMPORT_CALLBACK_MISSING`. This option is part of the experimental modules API. We do not recommend using it in a production environment.
 - `specifier` `<string>` specifier passed to `import()`
 - `script` `<vm.Script>`
 - Returns: `<Module Namespace Object>` | `<vm.Module>` Returning a `vm.Module` is recommended in order to take advantage of error tracking, and to avoid issues with namespaces that contain `then` function exports.

If `options` is a string, then it specifies the filename.

Creating a new `vm.Script` object compiles `code` but does not run it. The compiled `vm.Script` can be run later multiple times. The `code` is not bound to any global object; rather, it is bound before each run, just for that run.

`script.createCachedData()`

- Returns: `<Buffer>`

Creates a code cache that can be used with the `Script` constructor's `cachedData` option. Returns a `Buffer`. This method may be called at any time and any number of times.

```
const script = new vm.Script(`
function add(a, b) {
  return a + b;
}

const x = add(1, 2);
`);

const cacheWithoutX = script.createCachedData();

script.runInThisContext();

const cacheWithX = script.createCachedData();
```

`script.runInContext(contextifiedObject[, options])`

- `contextifiedObject` `<Object>` A `contextified` object as returned by the `vm.createContext()` method.
- `options` `<Object>`
 - `displayErrors` `<boolean>` When `true`, if an `Error` occurs while compiling the `code`, the line of code causing the error is attached to the stack trace. **Default:** `true`.
 - `timeout` `<integer>` Specifies the number of milliseconds to execute `code` before terminating execution. If execution is terminated, an `Error` will be thrown. This value must be a strictly positive integer.
 - `breakOnSigint` `<boolean>` If `true`, receiving `SIGINT` (`Ctrl` + `C`) will terminate execution and throw an `Error`. Existing handlers for the event that have been attached via `process.on('SIGINT')` are disabled during script execution, but continue to work after that. **Default:** `false`.
- Returns: `<any>` the result of the very last statement executed in the script.

Runs the compiled code contained by the `vm.Script` object within the given `contextifiedObject` and returns the result. Running code does not have access to local scope.

The following example compiles code that increments a global variable, sets the value of another global variable, then execute the code multiple times. The globals are contained in the `context` object.

```
const vm = require('vm');

const context = {
  animal: 'cat',
  count: 2
};

const script = new vm.Script('count += 1; name = "kitty";');
```

```
vm.createContext(context);
for (let i = 0; i < 10; ++i) {
  script.runInContext(context);
}

console.log(context);
// Prints: { animal: 'cat', count: 12, name: 'kitty' }
```

Using the `timeout` or `breakOnSigint` options will result in new event loops and corresponding threads being started, which have a non-zero performance overhead.

`script.runInNewContext([contextObject[, options]])`

- `contextObject` `<Object>` An object that will be `contextified`. If `undefined`, a new object will be created.
- `options` `<Object>`
 - `displayErrors` `<boolean>` When `true`, if an `Error` occurs while compiling the `code`, the line of code causing the error is attached to the stack trace. **Default:** `true`.
 - `timeout` `<integer>` Specifies the number of milliseconds to execute `code` before terminating execution. If execution is terminated, an `Error` will be thrown. This value must be a strictly positive integer.
 - `breakOnSigint` `<boolean>` If `true`, receiving `SIGINT` (`Ctrl + C`) will terminate execution and throw an `Error`. Existing handlers for the event that have been attached via `process.on('SIGINT')` are disabled during script execution, but continue to work after that. **Default:** `false`.
 - `contextName` `<string>` Human-readable name of the newly created context. **Default:** `'VM Context i'`, where `i` is an ascending numerical index of the created context.
 - `contextOrigin` `<string>` `Origin` corresponding to the newly created context for display purposes. The origin should be formatted like a URL, but with only the scheme, host, and port (if necessary), like the value of the `url.origin` property of a `URL` object. Most notably, this string should omit the trailing slash, as that denotes a path. **Default:** `''`.
 - `contextCodeGeneration` `<Object>`
 - `strings` `<boolean>` If set to `false` any calls to `eval` or function constructors (`Function`, `GeneratorFunction`, etc) will throw an `EvalError`. **Default:** `true`.
 - `wasm` `<boolean>` If set to `false` any attempt to compile a WebAssembly module will throw a `WebAssembly.CompileError`. **Default:** `true`.
 - `microtaskMode` `<string>` If set to `afterEvaluate`, microtasks (tasks scheduled through `Promise`s and `async function`s) will be run immediately after the script has run. They are included in the `timeout` and `breakOnSigint` scopes in that case.
- Returns: `<any>` the result of the very last statement executed in the script.

First contextifies the given `contextObject`, runs the compiled code contained by the `vm.Script` object within the created context, and returns the result. Running code does not have access to local scope.

The following example compiles code that sets a global variable, then executes the code multiple times in different contexts. The globals are set on and contained within each individual `context`.

```
const vm = require('vm');

const script = new vm.Script('globalVar = "set"');

const contexts = [{}, {}, {}];
contexts.forEach((context) => {
  script.runInNewContext(context);
});
```

```
console.log(contexts);
// Prints: [{ globalVar: 'set' }, { globalVar: 'set' }, { globalVar: 'set' }]
```

script.runInThisContext([options])

- **options** `<Object>`
 - **displayErrors** `<boolean>` When `true`, if an `Error` occurs while compiling the `code`, the line of code causing the error is attached to the stack trace. **Default:** `true`.
 - **timeout** `<integer>` Specifies the number of milliseconds to execute `code` before terminating execution. If execution is terminated, an `Error` will be thrown. This value must be a strictly positive integer.
 - **breakOnSigint** `<boolean>` If `true`, receiving `SIGINT` (`Ctrl + C`) will terminate execution and throw an `Error`. Existing handlers for the event that have been attached via `process.on('SIGINT')` are disabled during script execution, but continue to work after that. **Default:** `false`.
- **Returns:** `<any>` the result of the very last statement executed in the script.

Runs the compiled code contained by the `vm.Script` within the context of the current `global` object. Running code does not have access to local scope, but *does* have access to the current `global` object.

The following example compiles code that increments a `global` variable then executes that code multiple times:

```
const vm = require('vm');

global.globalVar = 0;

const script = new vm.Script('globalVar += 1', { filename: 'myfile.vm' });

for (let i = 0; i < 1000; ++i) {
  script.runInThisContext();
}

console.log(globalVar);

// 1000
```

Class: `vm.Module`

Stability: 1 - Experimental

This feature is only available with the `--experimental-vm-modules` command flag enabled.

The `vm.Module` class provides a low-level interface for using ECMAScript modules in VM contexts. It is the counterpart of the `vm.Script` class that closely mirrors `ModuleRecord`s as defined in the ECMAScript specification.

Unlike `vm.Script` however, every `vm.Module` object is bound to a context from its creation. Operations on `vm.Module` objects are intrinsically asynchronous, in contrast with the synchronous nature of `vm.Script` objects. The use of 'async' functions can help with manipulating `vm.Module` objects.

Using a `vm.Module` object requires three distinct steps: creation/parsing, linking, and evaluation. These three steps are illustrated in the following example.

This implementation lies at a lower level than the [ECMAScript Module loader](#). There is also no way to interact with the Loader yet, though support is planned.

```
import vm from 'vm';

const contextifiedObject = vm.createContext({
  secret: 42,
  print: console.log,
});

// Step 1
//
// Create a Module by constructing a new `vm.SourceTextModule` object. This
// parses the provided source text, throwing a `SyntaxError` if anything goes
// wrong. By default, a Module is created in the top context. But here, we
// specify `contextifiedObject` as the context this Module belongs to.
//
// Here, we attempt to obtain the default export from the module "foo", and
// put it into local binding "secret".

const bar = new vm.SourceTextModule(`
  import s from 'foo';
  s;
  print(s);
`, { context: contextifiedObject });

// Step 2
//
// "Link" the imported dependencies of this Module to it.
//
// The provided linking callback (the "linker") accepts two arguments: the
// parent module (`bar` in this case) and the string that is the specifier of
// the imported module. The callback is expected to return a Module that
// corresponds to the provided specifier, with certain requirements documented
// in `module.link()`.
//
// If linking has not started for the returned Module, the same linker
// callback will be called on the returned Module.
//
// Even top-level Modules without dependencies must be explicitly linked. The
// callback provided would never be called, however.
//
// The link() method returns a Promise that will be resolved when all the
// Promises returned by the linker resolve.
//
// Note: This is a contrived example in that the linker function creates a new
// "foo" module every time it is called. In a full-fledged module system, a
// cache would probably be used to avoid duplicated modules.
```

```

async function linker(specifier, referencingModule) {
  if (specifier === 'foo') {
    return new vm.SourceTextModule(`
      // The "secret" variable refers to the global variable we added to
      // "contextifiedObject" when creating the context.
      export default secret;
    `, { context: referencingModule.context });

    // Using `contextifiedObject` instead of `referencingModule.context`
    // here would work as well.
  }
  throw new Error(`Unable to resolve dependency: ${specifier}`);
}

await bar.link(linker);

// Step 3
//
// Evaluate the Module. The evaluate() method returns a promise which will
// resolve after the module has finished evaluating.

// Prints 42.
await bar.evaluate();const vm = require('vm');

const contextifiedObject = vm.createContext({
  secret: 42,
  print: console.log,
});

(async () => {
  // Step 1
  //
  // Create a Module by constructing a new `vm.SourceTextModule` object. This
  // parses the provided source text, throwing a `SyntaxError` if anything goes
  // wrong. By default, a Module is created in the top context. But here, we
  // specify `contextifiedObject` as the context this Module belongs to.
  //
  // Here, we attempt to obtain the default export from the module "foo", and
  // put it into local binding "secret".

  const bar = new vm.SourceTextModule(`
    import s from 'foo';
    s;
    print(s);
  `, { context: contextifiedObject });

  // Step 2
  //
  // "Link" the imported dependencies of this Module to it.
  //
  // The provided linking callback (the "linker") accepts two arguments: the

```

```

// parent module (`bar` in this case) and the string that is the specifier of
// the imported module. The callback is expected to return a Module that
// corresponds to the provided specifier, with certain requirements documented
// in `module.link()`.
//
// If linking has not started for the returned Module, the same linker
// callback will be called on the returned Module.
//
// Even top-level Modules without dependencies must be explicitly linked. The
// callback provided would never be called, however.
//
// The link() method returns a Promise that will be resolved when all the
// Promises returned by the linker resolve.
//
// Note: This is a contrived example in that the linker function creates a new
// "foo" module every time it is called. In a full-fledged module system, a
// cache would probably be used to avoid duplicated modules.

async function linker(specifier, referencingModule) {
  if (specifier === 'foo') {
    return new vm.SourceTextModule(`
      // The "secret" variable refers to the global variable we added to
      // "contextifiedObject" when creating the context.
      export default secret;
    `, { context: referencingModule.context });

    // Using `contextifiedObject` instead of `referencingModule.context`
    // here would work as well.
  }
  throw new Error(`Unable to resolve dependency: ${specifier}`);
}
await bar.link(linker);

// Step 3
//
// Evaluate the Module. The evaluate() method returns a promise which will
// resolve after the module has finished evaluating.

// Prints 42.
await bar.evaluate();
})();

```

module.dependencySpecifiers

- `<string[]>`

The specifiers of all dependencies of this module. The returned array is frozen to disallow any changes to it.

Corresponds to the `[[RequestedModules]]` field of [Cyclic Module Record](#)s in the ECMAScript specification.

module.error

- `<any>`

If the `module.status` is `'errored'`, this property contains the exception thrown by the module during evaluation. If the status is anything else, accessing this property will result in a thrown exception.

The value `undefined` cannot be used for cases where there is not a thrown exception due to possible ambiguity with `throw undefined;`.

Corresponds to the `[[EvaluationError]]` field of `Cyclic Module Record`s in the ECMAScript specification.

`module.evaluate([options])`

- `options` `<Object>`
 - `timeout` `<integer>` Specifies the number of milliseconds to evaluate before terminating execution. If execution is interrupted, an `Error` will be thrown. This value must be a strictly positive integer.
 - `breakOnSigint` `<boolean>` If `true`, receiving `SIGINT` (`Ctrl + C`) will terminate execution and throw an `Error`. Existing handlers for the event that have been attached via `process.on('SIGINT')` are disabled during script execution, but continue to work after that. **Default:** `false`.
- Returns: `<Promise>` Fulfills with `undefined` upon success.

Evaluate the module.

This must be called after the module has been linked; otherwise it will reject. It could be called also when the module has already been evaluated, in which case it will either do nothing if the initial evaluation ended in success (`module.status` is `'evaluated'`) or it will re-throw the exception that the initial evaluation resulted in (`module.status` is `'errored'`).

This method cannot be called while the module is being evaluated (`module.status` is `'evaluating'`).

Corresponds to the `Evaluate()` concrete method field of `Cyclic Module Record`s in the ECMAScript specification.

`module.identifier`

- `<string>`

The identifier of the current module, as set in the constructor.

`module.link(linker)`

- `linker` `<Function>`
 - `specifier` `<string>` The specifier of the requested module:

```
import foo from 'foo';
//          ^^^^^ the module specifier
```

- `extra` `<Object>`
 - `assert` `<Object>` The data from the assertion:

```
import foo from 'foo' assert { name: 'value' };
//          ^^^^^^^^^^^^^^^^^ the assertion
```

Per ECMA-262, hosts are expected to ignore assertions that they do not support, as opposed to, for example, triggering an error if an unsupported assertion is present.

- `referencingModule` `<vm.Module>` The `Module` object `link()` is called on.
- Returns: `<vm.Module>` | `<Promise>`

- Returns: `<Promise>`

Link module dependencies. This method must be called before evaluation, and can only be called once per module.

The function is expected to return a `Module` object or a `Promise` that eventually resolves to a `Module` object. The returned `Module` must satisfy the following two invariants:

- It must belong to the same context as the parent `Module`.
- Its `status` must not be `'errored'`.

If the returned `Module`'s `status` is `'unlinked'`, this method will be recursively called on the returned `Module` with the same provided `linker` function.

`link()` returns a `Promise` that will either get resolved when all linking instances resolve to a valid `Module`, or rejected if the linker function either throws an exception or returns an invalid `Module`.

The linker function roughly corresponds to the implementation-defined `HostResolveImportedModule` abstract operation in the ECMAScript specification, with a few key differences:

- The linker function is allowed to be asynchronous while `HostResolveImportedModule` is synchronous.

The actual `HostResolveImportedModule` implementation used during module linking is one that returns the modules linked during linking. Since at that point all modules would have been fully linked already, the `HostResolveImportedModule` implementation is fully synchronous per specification.

Corresponds to the `Link() concrete method` field of `Cyclic Module Record`s in the ECMAScript specification.

module.namespace

- `<Object>`

The namespace object of the module. This is only available after linking (`module.link()`) has completed.

Corresponds to the `GetModuleNamespace` abstract operation in the ECMAScript specification.

module.status

- `<string>`

The current status of the module. Will be one of:

- `'unlinked'`: `module.link()` has not yet been called.
- `'linking'`: `module.link()` has been called, but not all Promises returned by the linker function have been resolved yet.
- `'linked'`: The module has been linked successfully, and all of its dependencies are linked, but `module.evaluate()` has not yet been called.
- `'evaluating'`: The module is being evaluated through a `module.evaluate()` on itself or a parent module.
- `'evaluated'`: The module has been successfully evaluated.
- `'errored'`: The module has been evaluated, but an exception was thrown.

Other than `'errored'`, this status string corresponds to the specification's `Cyclic Module Record`'s `[[Status]]` field. `'errored'` corresponds to `'evaluated'` in the specification, but with `[[EvaluationError]]` set to a value that is not `undefined`.

Class: vm.SourceTextModule

This feature is only available with the `--experimental-vm-modules` command flag enabled.

- Extends: `<vm.Module>`

The `vm.SourceTextModule` class provides the `SourceTextModuleRecord` as defined in the ECMAScript specification.

`new vm.SourceTextModule(code[, options])`

- `code` `<string>` JavaScript Module code to parse
- `options`
 - `identifier` `<string>` String used in stack traces. **Default:** `'vm:module(i)'` where `i` is a context-specific ascending index.
 - `cachedData` `<Buffer>` | `<TypedArray>` | `<DataView>` Provides an optional `Buffer` or `TypedArray`, or `DataView` with V8's code cache data for the supplied source. The `code` must be the same as the module from which this `cachedData` was created.
 - `context` `<Object>` The `contextified` object as returned by the `vm.createContext()` method, to compile and evaluate this Module in.
 - `lineOffset` `<integer>` Specifies the line number offset that is displayed in stack traces produced by this Module. **Default:** `0`.
 - `columnOffset` `<integer>` Specifies the first-line column number offset that is displayed in stack traces produced by this Module. **Default:** `0`.
 - `initializeImportMeta` `<Function>` Called during evaluation of this Module to initialize the `import.meta`.
 - `meta` `<import.meta>`
 - `module` `<vm.SourceTextModule>`
 - `importModuleDynamically` `<Function>` Called during evaluation of this module when `import()` is called. If this option is not specified, calls to `import()` will reject with `ERR_VM_DYNAMIC_IMPORT_CALLBACK_MISSING`.
 - `specifier` `<string>` specifier passed to `import()`
 - `module` `<vm.Module>`
 - Returns: `<Module Namespace Object>` | `<vm.Module>` Returning a `vm.Module` is recommended in order to take advantage of error tracking, and to avoid issues with namespaces that contain `then` function exports.

Creates a new `SourceTextModule` instance.

Properties assigned to the `import.meta` object that are objects may allow the module to access information outside the specified `context`. Use `vm.runInContext()` to create objects in a specific context.

```
import vm from 'vm';

const contextifiedObject = vm.createContext({ secret: 42 });

const module = new vm.SourceTextModule(
  'Object.getPrototypeOf(import.meta.prop).secret = secret;',
  {
    initializeImportMeta(meta) {
      // Note: this object is created in the top context. As such,
      // Object.getPrototypeOf(import.meta.prop) points to the
      // Object.prototype in the top context rather than that in
      // the contextified object.
      meta.prop = {};
    }
  }
);
```

```

});
// Since module has no dependencies, the linker function will never be called.
await module.link(() => {});
await module.evaluate();

// Now, Object.prototype.secret will be equal to 42.
//
// To fix this problem, replace
//     meta.prop = {};
// above with
//     meta.prop = vm.runInContext('{}', contextifiedObject);
const vm = require('vm');
const contextifiedObject = vm.createContext({ secret: 42 });
(async () => {
  const module = new vm.SourceTextModule(
    'Object.getPrototypeOf(import.meta.prop).secret = secret;',
    {
      initializeImportMeta(meta) {
        // Note: this object is created in the top context. As such,
        // Object.getPrototypeOf(import.meta.prop) points to the
        // Object.prototype in the top context rather than that in
        // the contextified object.
        meta.prop = {};
      }
    }
  );
  // Since module has no dependencies, the linker function will never be called.
  await module.link(() => {});
  await module.evaluate();
  // Now, Object.prototype.secret will be equal to 42.
  //
  // To fix this problem, replace
  //     meta.prop = {};
  // above with
  //     meta.prop = vm.runInContext('{}', contextifiedObject);
})();

```

sourceTextModule.createCachedData()

- Returns: `<Buffer>`

Creates a code cache that can be used with the `SourceTextModule` constructor's `cachedData` option. Returns a `Buffer`. This method may be called any number of times before the module has been evaluated.

```

// Create an initial module
const module = new vm.SourceTextModule('const a = 1;');

// Create cached data from this module
const cachedData = module.createCachedData();

// Create a new module using the cached data. The code must be the same.
const module2 = new vm.SourceTextModule('const a = 1;', { cachedData });

```

Class: `vm.SyntheticModule`

Stability: 1 - Experimental

This feature is only available with the `--experimental-vm-modules` command flag enabled.

- Extends: `<vm.Module>`

The `vm.SyntheticModule` class provides the `Synthetic Module Record` as defined in the WebIDL specification. The purpose of synthetic modules is to provide a generic interface for exposing non-JavaScript sources to ECMAScript module graphs.

```
const vm = require('vm');

const source = '{ "a": 1 }';
const module = new vm.SyntheticModule(['default'], function() {
  const obj = JSON.parse(source);
  this.setExport('default', obj);
});

// Use `module` in linking...
```

`new vm.SyntheticModule(exportNames, evaluateCallback[, options])`

- `exportNames` `<string[]>` Array of names that will be exported from the module.
- `evaluateCallback` `<Function>` Called when the module is evaluated.
- `options`
 - `identifier` `<string>` String used in stack traces.**Default:** `'vm:module(i)'` where `i` is a context-specific ascending index.
 - `context` `<Object>` The `contextified` object as returned by the `vm.createContext()` method, to compile and evaluate this Module in.

Creates a new `SyntheticModule` instance.

Objects assigned to the exports of this instance may allow importers of the module to access information outside the specified `context`. Use `vm.runInContext()` to create objects in a specific context.

`syntheticModule.setExport(name, value)`

- `name` `<string>` Name of the export to set.
- `value` `<any>` The value to set the export to.

This method is used after the module is linked to set the values of exports. If it is called before the module is linked, an `ERR_VM_MODULE_STATUS` error will be thrown.

```
import vm from 'vm';

const m = new vm.SyntheticModule(['x'], () => {
  m.setExport('x', 1);
});

await m.link(() => {});
```

```

await m.evaluate();

assert.strictEqual(m.namespace.x, 1);const vm = require('vm');
(async () => {
  const m = new vm.SyntheticModule(['x'], () => {
    m.setExport('x', 1);
  });
  await m.link(() => {});
  await m.evaluate();
  assert.strictEqual(m.namespace.x, 1);
})();

```

vm.compileFunction(code[, params[, options]])

- **code** `<string>` The body of the function to compile.
- **params** `<string[]>` An array of strings containing all parameters for the function.
- **options** `<Object>`
 - **filename** `<string>` Specifies the filename used in stack traces produced by this script. **Default:** `''`.
 - **lineOffset** `<number>` Specifies the line number offset that is displayed in stack traces produced by this script. **Default:** `0`.
 - **columnOffset** `<number>` Specifies the first-line column number offset that is displayed in stack traces produced by this script. **Default:** `0`.
 - **cachedData** `<Buffer> | <TypedArray> | <DataView>` Provides an optional `Buffer` or `TypedArray`, or `DataView` with V8's code cache data for the supplied source.
 - **produceCachedData** `<boolean>` Specifies whether to produce new cache data. **Default:** `false`.
 - **parsingContext** `<Object>` The `contextified` object in which the said function should be compiled in.
 - **contextExtensions** `<Object[]>` An array containing a collection of context extensions (objects wrapping the current scope) to be applied while compiling. **Default:** `[]`.
 - **importModuleDynamically** `<Function>` Called during evaluation of this module when `import()` is called. If this option is not specified, calls to `import()` will reject with `ERR_VM_DYNAMIC_IMPORT_CALLBACK_MISSING`. This option is part of the experimental modules API, and should not be considered stable.
 - **specifier** `<string>` specifier passed to `import()`
 - **function** `<Function>`
 - **Returns:** `<Module Namespace Object> | <vm.Module>` Returning a `vm.Module` is recommended in order to take advantage of error tracking, and to avoid issues with namespaces that contain `then` function exports.
- **Returns:** `<Function>`

Compiles the given code into the provided context (if no context is supplied, the current context is used), and returns it wrapped inside a function with the given `params`.

vm.createContext([contextObject[, options]])

- **contextObject** `<Object>`
- **options** `<Object>`
 - **name** `<string>` Human-readable name of the newly created context. **Default:** `'VM Context i'`, where `i` is an ascending numerical index of the created context.
 - **origin** `<string>` `Origin` corresponding to the newly created context for display purposes. The origin should be formatted like a URL, but with only the scheme, host, and port (if necessary), like the value of the `url.origin` property of a `URL` object. Most notably, this string should omit the trailing slash, as that denotes a path. **Default:** `''`.

- `codeGeneration` `<Object>`
 - `strings` `<boolean>` If set to false any calls to `eval` or function constructors (`Function` , `GeneratorFunction` , etc) will throw an `EvalError` . **Default:** `true` .
 - `wasn` `<boolean>` If set to false any attempt to compile a WebAssembly module will throw a `WebAssembly.CompileError` . **Default:** `true` .
- `microtaskMode` `<string>` If set to `afterEvaluate` , microtasks (tasks scheduled through `Promise` s and `async` function s) will be run immediately after a script has run through `script.runInContext()` . They are included in the `timeout` and `breakOnSigint` scopes in that case.
- Returns: `<Object>` contextified object.

If given a `contextObject` , the `vm.createContext()` method will `prepare that object` so that it can be used in calls to `vm.runInContext()` or `script.runInContext()` . Inside such scripts, the `contextObject` will be the global object, retaining all of its existing properties but also having the built-in objects and functions any standard `global object` has. Outside of scripts run by the `vm` module, global variables will remain unchanged.

```
const vm = require('vm');

global.globalVar = 3;

const context = { globalVar: 1 };
vm.createContext(context);

vm.runInContext('globalVar *= 2;', context);

console.log(context);
// Prints: { globalVar: 2 }

console.log(global.globalVar);
// Prints: 3
```

If `contextObject` is omitted (or passed explicitly as `undefined`), a new, empty `contextified` object will be returned.

The `vm.createContext()` method is primarily useful for creating a single context that can be used to run multiple scripts. For instance, if emulating a web browser, the method can be used to create a single context representing a window's global object, then run all `<script>` tags together within that context.

The provided `name` and `origin` of the context are made visible through the Inspector API.

vm.isContext(object)

- `object` `<Object>`
- Returns: `<boolean>`

Returns `true` if the given `object` object has been `contextified` using `vm.createContext()` .

vm.measureMemory([options])

Stability: 1 - Experimental

Measure the memory known to V8 and used by all contexts known to the current V8 isolate, or the main context.

- `options` `<Object>` Optional.
 - `mode` `<string>` Either `'summary'` or `'detailed'`. In summary mode, only the memory measured for the main context will be returned. In detailed mode, the measure measured for all contexts known to the current V8 isolate will be returned. **Default:** `'summary'`
 - `execution` `<string>` Either `'default'` or `'eager'`. With default execution, the promise will not resolve until after the next scheduled garbage collection starts, which may take a while (or never if the program exits before the next GC). With eager execution, the GC will be started right away to measure the memory. **Default:** `'default'`
- Returns: `<Promise>` If the memory is successfully measured the promise will resolve with an object containing information about the memory usage.

The format of the object that the returned Promise may resolve with is specific to the V8 engine and may change from one version of V8 to the next.

The returned result is different from the statistics returned by `v8.getHeapSpaceStatistics()` in that `vm.measureMemory()` measure the memory reachable by each V8 specific contexts in the current instance of the V8 engine, while the result of `v8.getHeapSpaceStatistics()` measure the memory occupied by each heap space in the current V8 instance.

```
const vm = require('vm');
// Measure the memory used by the main context.
vm.measureMemory({ mode: 'summary' })
  // This is the same as vm.measureMemory()
  .then((result) => {
    // The current format is:
    // {
    //   total: {
    //     jsMemoryEstimate: 2418479, jsMemoryRange: [ 2418479, 2745799 ]
    //   }
    // }
    console.log(result);
  });

const context = vm.createContext({ a: 1 });
vm.measureMemory({ mode: 'detailed', execution: 'eager' })
  .then((result) => {
    // Reference the context here so that it won't be GC'ed
    // until the measurement is complete.
    console.log(context.a);
    // {
    //   total: {
    //     jsMemoryEstimate: 2574732,
    //     jsMemoryRange: [ 2574732, 2904372 ]
    //   },
    //   current: {
    //     jsMemoryEstimate: 2438996,
    //     jsMemoryRange: [ 2438996, 2768636 ]
    //   },
    //   other: [
    //     {
    //       jsMemoryEstimate: 135736,
    //       jsMemoryRange: [ 135736, 465376 ]
    //     }
    //   ]
    // }
```

```
//    }
//  ]
// }
console.log(result);
});
```

vm.runInContext(code, contextifiedObject[, options])

- **code** `<string>` The JavaScript code to compile and run.
- **contextifiedObject** `<Object>` The `contextified` object that will be used as the `global` when the `code` is compiled and run.
- **options** `<Object>` | `<string>`
 - **filename** `<string>` Specifies the filename used in stack traces produced by this script. **Default:** `'evalmachine.<anonymous>'`.
 - **lineOffset** `<number>` Specifies the line number offset that is displayed in stack traces produced by this script. **Default:** `0`.
 - **columnOffset** `<number>` Specifies the first-line column number offset that is displayed in stack traces produced by this script. **Default:** `0`.
 - **displayErrors** `<boolean>` When `true`, if an `Error` occurs while compiling the `code`, the line of code causing the error is attached to the stack trace. **Default:** `true`.
 - **timeout** `<integer>` Specifies the number of milliseconds to execute `code` before terminating execution. If execution is terminated, an `Error` will be thrown. This value must be a strictly positive integer.
 - **breakOnSigint** `<boolean>` If `true`, receiving `SIGINT` (`Ctrl` + `C`) will terminate execution and throw an `Error`. Existing handlers for the event that have been attached via `process.on('SIGINT')` are disabled during script execution, but continue to work after that. **Default:** `false`.
 - **cachedData** `<Buffer>` | `<TypedArray>` | `<DataView>` Provides an optional `Buffer` or `TypedArray`, or `DataView` with V8's code cache data for the supplied source. When supplied, the `cachedDataRejected` value will be set to either `true` or `false` depending on acceptance of the data by V8.
 - **produceCachedData** `<boolean>` When `true` and no `cachedData` is present, V8 will attempt to produce code cache data for `code`. Upon success, a `Buffer` with V8's code cache data will be produced and stored in the `cachedData` property of the returned `vm.Script` instance. The `cachedDataProduced` value will be set to either `true` or `false` depending on whether code cache data is produced successfully. This option is **deprecated** in favor of `script.createCachedData()`. **Default:** `false`.
 - **importModuleDynamically** `<Function>` Called during evaluation of this module when `import()` is called. If this option is not specified, calls to `import()` will reject with `ERR_VM_DYNAMIC_IMPORT_CALLBACK_MISSING`. This option is part of the experimental modules API. We do not recommend using it in a production environment.
 - **specifier** `<string>` specifier passed to `import()`
 - **script** `<vm.Script>`
 - **Returns:** `<Module Namespace Object>` | `<vm.Module>` Returning a `vm.Module` is recommended in order to take advantage of error tracking, and to avoid issues with namespaces that contain `then` function exports.
- **Returns:** `<any>` the result of the very last statement executed in the script.

The `vm.runInContext()` method compiles `code`, runs it within the context of the `contextifiedObject`, then returns the result. Running code does not have access to the local scope. The `contextifiedObject` object *must* have been previously `contextified` using the `vm.createContext()` method.

If `options` is a string, then it specifies the filename.

The following example compiles and executes different scripts using a single `contextified` object:

```
const vm = require('vm');

const contextObject = { globalVar: 1 };
```

```
vm.createContext(contextObject);

for (let i = 0; i < 10; ++i) {
  vm.runInContext('globalVar *= 2;', contextObject);
}

console.log(contextObject);
// Prints: { globalVar: 1024 }
```

vm.runInNewContext(code[, contextObject[, options]])

- **code** `<string>` The JavaScript code to compile and run.
- **contextObject** `<Object>` An object that will be `contextified`. If `undefined`, a new object will be created.
- **options** `<Object> | <string>`
 - **filename** `<string>` Specifies the filename used in stack traces produced by this script. **Default:** `'evalmachine.<anonymous>'`.
 - **lineOffset** `<number>` Specifies the line number offset that is displayed in stack traces produced by this script. **Default:** `0`.
 - **columnOffset** `<number>` Specifies the first-line column number offset that is displayed in stack traces produced by this script. **Default:** `0`.
 - **displayErrors** `<boolean>` When `true`, if an `Error` occurs while compiling the `code`, the line of code causing the error is attached to the stack trace. **Default:** `true`.
 - **timeout** `<integer>` Specifies the number of milliseconds to execute `code` before terminating execution. If execution is terminated, an `Error` will be thrown. This value must be a strictly positive integer.
 - **breakOnSigint** `<boolean>` If `true`, receiving `SIGINT` (`Ctrl + C`) will terminate execution and throw an `Error`. Existing handlers for the event that have been attached via `process.on('SIGINT')` are disabled during script execution, but continue to work after that. **Default:** `false`.
 - **contextName** `<string>` Human-readable name of the newly created context. **Default:** `'VM Context i'`, where `i` is an ascending numerical index of the created context.
 - **contextOrigin** `<string>` `Origin` corresponding to the newly created context for display purposes. The origin should be formatted like a URL, but with only the scheme, host, and port (if necessary), like the value of the `url.origin` property of a `URL` object. Most notably, this string should omit the trailing slash, as that denotes a path. **Default:** `''`.
 - **contextCodeGeneration** `<Object>`
 - **strings** `<boolean>` If set to `false` any calls to `eval` or function constructors (`Function`, `GeneratorFunction`, etc) will throw an `EvalError`. **Default:** `true`.
 - **wasm** `<boolean>` If set to `false` any attempt to compile a WebAssembly module will throw a `WebAssembly.CompileError`. **Default:** `true`.
 - **cachedData** `<Buffer> | <TypedArray> | <DataView>` Provides an optional `Buffer` or `TypedArray`, or `DataView` with V8's code cache data for the supplied source. When supplied, the `cachedDataRejected` value will be set to either `true` or `false` depending on acceptance of the data by V8.
 - **produceCachedData** `<boolean>` When `true` and no `cachedData` is present, V8 will attempt to produce code cache data for `code`. Upon success, a `Buffer` with V8's code cache data will be produced and stored in the `cachedData` property of the returned `vm.Script` instance. The `cachedDataProduced` value will be set to either `true` or `false` depending on whether code cache data is produced successfully. This option is **deprecated** in favor of `script.createCachedData()`. **Default:** `false`.
 - **importModuleDynamically** `<Function>` Called during evaluation of this module when `import()` is called. If this option is not specified, calls to `import()` will reject with `ERR_VM_DYNAMIC_IMPORT_CALLBACK_MISSING`. This option is part of the experimental modules API. We do not recommend using it in a production environment.
 - **specifier** `<string>` specifier passed to `import()`
 - **script** `<vm.Script>`
 - **Returns:** `<Module Namespace Object> | <vm.Module>` Returning a `vm.Module` is recommended in order to take advantage of error tracking, and to avoid issues with namespaces that contain `then` function exports.

- `microtaskMode` `<string>` If set to `afterEvaluate`, microtasks (tasks scheduled through `Promise`s and `async function`s) will be run immediately after the script has run. They are included in the `timeout` and `breakOnSigint` scopes in that case.
- Returns: `<any>` the result of the very last statement executed in the script.

The `vm.runInNewContext()` first contextifies the given `contextObject` (or creates a new `contextObject` if passed as `undefined`), compiles the `code`, runs it within the created context, then returns the result. Running code does not have access to the local scope.

If `options` is a string, then it specifies the filename.

The following example compiles and executes code that increments a global variable and sets a new one. These globals are contained in the `contextObject`.

```
const vm = require('vm');

const contextObject = {
  animal: 'cat',
  count: 2
};

vm.runInNewContext('count += 1; name = "kitty"', contextObject);
console.log(contextObject);
// Prints: { animal: 'cat', count: 3, name: 'kitty' }
```

vm.runInThisContext(code[, options])

- `code` `<string>` The JavaScript code to compile and run.
- `options` `<Object>` | `<string>`
 - `filename` `<string>` Specifies the filename used in stack traces produced by this script. **Default:** `'evalmachine.<anonymous>'`.
 - `lineOffset` `<number>` Specifies the line number offset that is displayed in stack traces produced by this script. **Default:** `0`.
 - `columnOffset` `<number>` Specifies the first-line column number offset that is displayed in stack traces produced by this script. **Default:** `0`.
 - `displayErrors` `<boolean>` When `true`, if an `Error` occurs while compiling the `code`, the line of code causing the error is attached to the stack trace. **Default:** `true`.
 - `timeout` `<integer>` Specifies the number of milliseconds to execute `code` before terminating execution. If execution is terminated, an `Error` will be thrown. This value must be a strictly positive integer.
 - `breakOnSigint` `<boolean>` If `true`, receiving `SIGINT` (`Ctrl + C`) will terminate execution and throw an `Error`. Existing handlers for the event that have been attached via `process.on('SIGINT')` are disabled during script execution, but continue to work after that. **Default:** `false`.
 - `cachedData` `<Buffer>` | `<TypedArray>` | `<DataView>` Provides an optional `Buffer` or `TypedArray`, or `DataView` with V8's code cache data for the supplied source. When supplied, the `cachedDataRejected` value will be set to either `true` or `false` depending on acceptance of the data by V8.
 - `produceCachedData` `<boolean>` When `true` and no `cachedData` is present, V8 will attempt to produce code cache data for `code`. Upon success, a `Buffer` with V8's code cache data will be produced and stored in the `cachedData` property of the returned `vm.Script` instance. The `cachedDataProduced` value will be set to either `true` or `false` depending on whether code cache data is produced successfully. This option is **deprecated** in favor of `script.createCachedData()`. **Default:** `false`.
 - `importModuleDynamically` `<Function>` Called during evaluation of this module when `import()` is called. If this option is not specified, calls to `import()` will reject with `ERR_VM_DYNAMIC_IMPORT_CALLBACK_MISSING`. This option is part of the experimental modules API. We do not recommend using it in a production environment.
 - `specifier` `<string>` specifier passed to `import()`

- `script` `<vm.Script>`
- Returns: `<Module Namespace Object> | <vm.Module>` Returning a `vm.Module` is recommended in order to take advantage of error tracking, and to avoid issues with namespaces that contain `then` function exports.
- Returns: `<any>` the result of the very last statement executed in the script.

`vm.runInThisContext()` compiles `code`, runs it within the context of the current `global` and returns the result. Running code does not have access to local scope, but does have access to the current `global` object.

If `options` is a string, then it specifies the filename.

The following example illustrates using both `vm.runInThisContext()` and the JavaScript `eval()` function to run the same code:

```
const vm = require('vm');
let localVar = 'initial value';

const vmResult = vm.runInThisContext('localVar = "vm";');
console.log(`vmResult: '${vmResult}', localVar: '${localVar}'`);
// Prints: vmResult: 'vm', localVar: 'initial value'

const evalResult = eval('localVar = "eval";');
console.log(`evalResult: '${evalResult}', localVar: '${localVar}'`);
// Prints: evalResult: 'eval', localVar: 'eval'
```

Because `vm.runInThisContext()` does not have access to the local scope, `localVar` is unchanged. In contrast, `eval()` *does* have access to the local scope, so the value `localVar` is changed. In this way `vm.runInThisContext()` is much like an *indirect eval() call*, e.g. `(0, eval)('code')`.

Example: Running an HTTP server within a VM

When using either `script.runInThisContext()` or `vm.runInThisContext()`, the code is executed within the current V8 global context. The code passed to this VM context will have its own isolated scope.

In order to run a simple web server using the `http` module the code passed to the context must either call `require('http')` on its own, or have a reference to the `http` module passed to it. For instance:

```
'use strict';
const vm = require('vm');

const code = `
((require) => {
  const http = require('http');

  http.createServer((request, response) => {
    response.writeHead(200, { 'Content-Type': 'text/plain' });
    response.end('Hello World\\n');
  }).listen(8124);

  console.log('Server running at http://127.0.0.1:8124/');
})`;

vm.runInThisContext(code)(require);
```

The `require()` in the above case shares the state with the context it is passed from. This may introduce risks when untrusted code is executed, e.g. altering objects in the context in unwanted ways.

What does it mean to "contextify" an object?

All JavaScript executed within Node.js runs within the scope of a "context". According to the [V8 Embedder's Guide](#) :

In V8, a context is an execution environment that allows separate, unrelated, JavaScript applications to run in a single instance of V8. You must explicitly specify the context in which you want any JavaScript code to be run.

When the method `vm.createContext()` is called, the `contextObject` argument (or a newly-created object if `contextObject` is `undefined`) is associated internally with a new instance of a V8 Context. This V8 Context provides the `code` run using the `vm` module's methods with an isolated global environment within which it can operate. The process of creating the V8 Context and associating it with the `contextObject` is what this document refers to as "contextifying" the object.

Timeout interactions with asynchronous tasks and Promises

`Promise`s and `async` functions can schedule tasks run by the JavaScript engine asynchronously. By default, these tasks are run after all JavaScript functions on the current stack are done executing. This allows escaping the functionality of the `timeout` and `breakOnSigint` options.

For example, the following code executed by `vm.runInNewContext()` with a timeout of 5 milliseconds schedules an infinite loop to run after a promise resolves. The scheduled loop is never interrupted by the timeout:

```
const vm = require('vm');

function loop() {
  console.log('entering loop');
  while (1) console.log(Date.now());
}

vm.runInNewContext(
  'Promise.resolve().then(() => loop());',
  { loop, console },
  { timeout: 5 }
);
// This is printed *before* 'entering loop' (!)
console.log('done executing');
```

This can be addressed by passing `microtaskMode: 'afterEvaluate'` to the code that creates the `Context` :

```
const vm = require('vm');

function loop() {
  while (1) console.log(Date.now());
}

vm.runInNewContext(
  'Promise.resolve().then(() => loop());',
  { loop, console },
  { microtaskMode: 'afterEvaluate' }
```

```
{ timeout: 5, microtaskMode: 'afterEvaluate' }  
};
```

In this case, the microtask scheduled through `promise.then()` will be run before returning from `vm.runInNewContext()`, and will be interrupted by the `timeout` functionality. This applies only to code running in a `vm.Context`, so e.g. `vm.runInThisContext()` does not take this option.

Promise callbacks are entered into the microtask queue of the context in which they were created. For example, if `() => loop()` is replaced with just `loop` in the above example, then `loop` will be pushed into the global microtask queue, because it is a function from the outer (main) context, and thus will also be able to escape the timeout.

If asynchronous scheduling functions such as `process.nextTick()`, `queueMicrotask()`, `setTimeout()`, `setImmediate()`, etc. are made available inside a `vm.Context`, functions passed to them will be added to global queues, which are shared by all contexts. Therefore, callbacks passed to those functions are not controllable through the timeout either.

WebAssembly System Interface (WASI)

Stability: 1 - Experimental

Source Code: [lib/wasi.js](#)

The WASI API provides an implementation of the [WebAssembly System Interface](#) specification. WASI gives sandboxed WebAssembly applications access to the underlying operating system via a collection of POSIX-like functions.

```
import fs from 'fs';  
import { WASI } from 'wasi';  
import { argv, env } from 'process';  
  
const wasi = new WASI({  
  args: argv,  
  env,  
  preopens: {  
    '/sandbox': '/some/real/path/that/wasm/can/access'  
  }  
});  
  
const importObject = { wasi_snapshot_preview1: wasi.wasiImport };  
  
const wasm = await WebAssembly.compile(fs.readFileSync('./demo.wasm'));  
const instance = await WebAssembly.instantiate(wasm, importObject);  
  
wasi.start(instance); 'use strict';  
const fs = require('fs');  
const { WASI } = require('wasi');  
const { argv, env } = require('process');  
  
const wasi = new WASI({  
  args: argv,  
  env,  
  preopens: {
```

```

    '/sandbox': '/some/real/path/that/wasm/can/access'
  }
});
const importObject = { wasi_snapshot_preview1: wasi.wasiImport };

(async () => {
  const wasm = await WebAssembly.compile(fs.readFileSync('./demo.wasm'));
  const instance = await WebAssembly.instantiate(wasm, importObject);

  wasi.start(instance);
})();

```

To run the above example, create a new WebAssembly text format file named `demo.wat` :

```

(module
  ;; Import the required fd_write WASI function which will write the given io vectors to stdout
  ;; The function signature for fd_write is:
  ;; (File Descriptor, *iovs, iovs_len, nwritten) -> Returns number of bytes written
  (import "wasi_snapshot_preview1" "fd_write" (func $fd_write (param i32 i32 i32 i32) (result i32)))

  (memory 1)
  (export "memory" (memory 0))

  ;; Write 'hello world\n' to memory at an offset of 8 bytes
  ;; Note the trailing newline which is required for the text to appear
  (data (i32.const 8) "hello world\n")

  (func $main (export "_start")
    ;; Creating a new io vector within linear memory
    (i32.store (i32.const 0) (i32.const 8)) ;; iov.iov_base - This is a pointer to the start of the 'hello world\n'
    (i32.store (i32.const 4) (i32.const 12)) ;; iov.iov_len - The length of the 'hello world\n' string

    (call $fd_write
      (i32.const 1) ;; file_descriptor - 1 for stdout
      (i32.const 0) ;; *iovs - The pointer to the iov array, which is stored at memory location 0
      (i32.const 1) ;; iovs_len - We're printing 1 string stored in an iov - so one.
      (i32.const 20) ;; nwritten - A place in memory to store the number of bytes written
    )
    drop ;; Discard the number of bytes written from the top of the stack
  )
)

```

Use `wabt` to compile `.wat` to `.wasm`

```
$ wat2wasm demo.wat
```

The `--experimental-wasi-unstable-preview1` CLI argument is needed for this example to run.

Class: WASI

The `WASI` class provides the WASI system call API and additional convenience methods for working with WASI-based applications. Each `WASI` instance represents a distinct sandbox environment. For security purposes, each `WASI` instance must have its command-line arguments, environment variables, and sandbox directory structure configured explicitly.

`new WASI([options])`

- `options` `<Object>`
 - `args` `<Array>` An array of strings that the WebAssembly application will see as command-line arguments. The first argument is the virtual path to the WASI command itself. **Default:** `[]`.
 - `env` `<Object>` An object similar to `process.env` that the WebAssembly application will see as its environment. **Default:** `{}`.
 - `preopens` `<Object>` This object represents the WebAssembly application's sandbox directory structure. The string keys of `preopens` are treated as directories within the sandbox. The corresponding values in `preopens` are the real paths to those directories on the host machine.
 - `returnOnExit` `<boolean>` By default, WASI applications terminate the Node.js process via the `__wasi_proc_exit()` function. Setting this option to `true` causes `wasi.start()` to return the exit code rather than terminate the process. **Default:** `false`.
 - `stdin` `<integer>` The file descriptor used as standard input in the WebAssembly application. **Default:** `0`.
 - `stdout` `<integer>` The file descriptor used as standard output in the WebAssembly application. **Default:** `1`.
 - `stderr` `<integer>` The file descriptor used as standard error in the WebAssembly application. **Default:** `2`.

`wasi.start(instance)`

- `instance` `<WebAssembly.Instance>`

Attempt to begin execution of `instance` as a WASI command by invoking its `_start()` export. If `instance` does not contain a `_start()` export, or if `instance` contains an `_initialize()` export, then an exception is thrown.

`start()` requires that `instance` exports a `WebAssembly.Memory` named `memory`. If `instance` does not have a `memory` export an exception is thrown.

If `start()` is called more than once, an exception is thrown.

`wasi.initialize(instance)`

- `instance` `<WebAssembly.Instance>`

Attempt to initialize `instance` as a WASI reactor by invoking its `_initialize()` export, if it is present. If `instance` contains a `_start()` export, then an exception is thrown.

`initialize()` requires that `instance` exports a `WebAssembly.Memory` named `memory`. If `instance` does not have a `memory` export an exception is thrown.

If `initialize()` is called more than once, an exception is thrown.

`wasi.wasiImport`

- `<Object>`

`wasiImport` is an object that implements the WASI system call API. This object should be passed as the `wasi_snapshot_preview1` import during the instantiation of a `WebAssembly.Instance`.

Web Crypto API

Node.js provides an implementation of the standard [Web Crypto API](#).

Use `require('crypto').webcrypto` to access this module.

```
const { subtle } = require('crypto').webcrypto;

(async function() {

  const key = await subtle.generateKey({
    name: 'HMAC',
    hash: 'SHA-256',
    length: 256
  }, true, ['sign', 'verify']);

  const digest = await subtle.sign({
    name: 'HMAC'
  }, key, 'I love cupcakes');

})();
```

Examples

Generating keys

The `<SubtleCrypto>` class can be used to generate symmetric (secret) keys or asymmetric key pairs (public key and private key).

AES keys

```
const { subtle } = require('crypto').webcrypto;

async function generateAesKey(length = 256) {
  const key = await subtle.generateKey({
    name: 'AES-CBC',
    length
  }, true, ['encrypt', 'decrypt']);

  return key;
}
```

Elliptic curve key pairs

```
const { subtle } = require('crypto').webcrypto;

async function generateEcKey(namedCurve = 'P-521') {
  const {
    publicKey,
```

```

    privateKey
  } = await subtle.generateKey({
    name: 'ECDSA',
    namedCurve,
  }, true, ['sign', 'verify']);

  return { publicKey, privateKey };
}

```

ED25519/ED448/X25519/X448 Elliptic curve key pairs

```

const { subtle } = require('crypto').webcrypto;

async function generateEd25519Key() {
  return subtle.generateKey({
    name: 'NODE-ED25519',
    namedCurve: 'NODE-ED25519',
  }, true, ['sign', 'verify']);
}

async function generateX25519Key() {
  return subtle.generateKey({
    name: 'ECDH',
    namedCurve: 'NODE-X25519',
  }, true, ['deriveKey']);
}

```

HMAC keys

```

const { subtle } = require('crypto').webcrypto;

async function generateHmacKey(hash = 'SHA-256') {
  const key = await subtle.generateKey({
    name: 'HMAC',
    hash
  }, true, ['sign', 'verify']);

  return key;
}

```

RSA key pairs

```

const { subtle } = require('crypto').webcrypto;
const publicExponent = new Uint8Array([1, 0, 1]);

async function generateRsaKey(modulusLength = 2048, hash = 'SHA-256') {
  const {
    publicKey,
    privateKey
  } = await subtle.generateKey({
    name: 'RSA',
    modulusLength,
    publicExponent,
    hash
  }, true, ['sign', 'verify']);

  return { publicKey, privateKey };
}

```



```

} = await subtle.generateKey({
  name: 'RSASSA-PKCS1-v1_5',
  modulusLength,
  publicExponent,
  hash,
}, true, ['sign', 'verify']);

return { publicKey, privateKey };
}

```

Encryption and decryption

```

const { subtle, getRandomValues } = require('crypto').webcrypto;

async function aesEncrypt(plaintext) {
  const ec = new TextEncoder();
  const key = await generateAesKey();
  const iv = getRandomValues(new Uint8Array(16));

  const ciphertext = await subtle.encrypt({
    name: 'AES-CBC',
    iv,
  }, key, ec.encode(plaintext));

  return {
    key,
    iv,
    ciphertext
  };
}

async function aesDecrypt(ciphertext, key, iv) {
  const dec = new TextDecoder();
  const plaintext = await subtle.decrypt({
    name: 'AES-CBC',
    iv,
  }, key, ciphertext);

  return dec.decode(plaintext);
}

```

Exporting and importing keys

```

const { subtle } = require('crypto').webcrypto;

async function generateAndExportHmacKey(format = 'jwk', hash = 'SHA-512') {
  const key = await subtle.generateKey({
    name: 'HMAC',
    hash
  }, true, ['sign', 'verify']);
}

```

```

    return subtle.exportKey(format, key);
}

async function importHmacKey(keyData, format = 'jwk', hash = 'SHA-512') {
  const key = await subtle.importKey(format, keyData, {
    name: 'HMAC',
    hash
  }, true, ['sign', 'verify']);

  return key;
}

```

Wrapping and unwrapping keys

```

const { subtle } = require('crypto').webcrypto;

async function generateAndWrapHmacKey(format = 'jwk', hash = 'SHA-512') {
  const [
    key,
    wrappingKey,
  ] = await Promise.all([
    subtle.generateKey({
      name: 'HMAC', hash
    }, true, ['sign', 'verify']),
    subtle.generateKey({
      name: 'AES-KW',
      length: 256
    }, true, ['wrapKey', 'unwrapKey']),
  ]);

  const wrappedKey = await subtle.wrapKey(format, key, wrappingKey, 'AES-KW');

  return wrappedKey;
}

async function unwrapHmacKey(
  wrappedKey,
  wrappingKey,
  format = 'jwk',
  hash = 'SHA-512') {
  const key = await subtle.unwrapKey(
    format,
    wrappedKey,
    unwrappingKey,
    'AES-KW',
    { name: 'HMAC', hash },
    true,
    ['sign', 'verify']);
}

```

```
    return key;
  }
```

Sign and verify

```
const { subtle } = require('crypto').webcrypto;

async function sign(key, data) {
  const ec = new TextEncoder();
  const signature =
    await subtle.sign('RSASSA-PKCS1-v1_5', key, ec.encode(data));
  return signature;
}

async function verify(key, signature, data) {
  const ec = new TextEncoder();
  const verified =
    await subtle.verify(
      'RSASSA-PKCS1-v1_5',
      key,
      signature,
      ec.encode(data));
  return verified;
}
```

Deriving bits and keys

```
const { subtle } = require('crypto').webcrypto;

async function pbkdf2(pass, salt, iterations = 1000, length = 256) {
  const ec = new TextEncoder();
  const key = await subtle.importKey(
    'raw',
    ec.encode(pass),
    'PBKDF2',
    false,
    ['deriveBits']);
  const bits = await subtle.deriveBits({
    name: 'PBKDF2',
    hash: 'SHA-512',
    salt: ec.encode(salt),
    iterations
  }, key, length);
  return bits;
}

async function pbkdf2Key(pass, salt, iterations = 1000, length = 256) {
  const ec = new TextEncoder();
  const keyMaterial = await subtle.importKey(
```


Algorithm	generateKey	exportKey	importKey	encrypt	decrypt	wrapKey	unwrapKey	deriveBits	deriveKey
'AES-GCM'	✓	✓	✓	✓	✓	✓	✓		
'AES-KW'	✓	✓	✓			✓	✓		
'HMAC'	✓	✓	✓						
'HKDF'		✓	✓					✓	✓
'PBKDF2'		✓	✓					✓	✓
'SHA-1'									
'SHA-256'									
'SHA-384'									
'SHA-512'									
'NODE-DSA' ¹	✓	✓	✓						
'NODE-DH' ¹	✓	✓	✓					✓	✓
'NODE-ED25519' ¹	✓	✓	✓						
'NODE-ED448' ¹	✓	✓	✓						

¹ Node.js-specific extension

Class: `Crypto`

Calling `require('crypto').webcrypto` returns an instance of the `Crypto` class. `Crypto` is a singleton that provides access to the remainder of the crypto API.

`crypto.subtle`

- Type: `<SubtleCrypto>`

Provides access to the `SubtleCrypto` API.

`crypto.getRandomValues(typedArray)`

- `typedArray` `<Buffer>` | `<TypedArray>` | `<DataView>` | `<ArrayBuffer>`
- Returns: `<Buffer>` | `<TypedArray>` | `<DataView>` | `<ArrayBuffer>` Returns `typedArray`.

Generates cryptographically strong random values. The given `typedArray` is filled with random values, and a reference to `typedArray` is returned.

An error will be thrown if the given `typedArray` is larger than 65,536 bytes.

`crypto.randomUUID()`

- Returns: `<string>`

Generates a random [RFC 4122](#) version 4 UUID. The UUID is generated using a cryptographic pseudorandom number generator.

Class: `CryptoKey`

`cryptoKey.algorithm`

- Type: `<AesKeyGenParams> | <RsaHashedKeyGenParams> | <EcKeyGenParams> | <HmacKeyGenParams> | <NodeDsaKeyGenParams> | <NodeDhKeyGenParams>`

An object detailing the algorithm for which the key can be used along with additional algorithm-specific parameters.

Read-only.

`cryptoKey.extractable`

- Type: `<boolean>`

When `true`, the `<CryptoKey>` can be extracted using either `subtleCrypto.exportKey()` or `subtleCrypto.wrapKey()`.

Read-only.

`cryptoKey.type`

- Type: `<string>` One of `'secret'`, `'private'`, or `'public'`.

A string identifying whether the key is a symmetric (`'secret'`) or asymmetric (`'private'` or `'public'`) key.

`cryptoKey.usages`

- Type: `<string[]>`

An array of strings identifying the operations for which the key may be used.

The possible usages are:

- `'encrypt'` - The key may be used to encrypt data.
- `'decrypt'` - The key may be used to decrypt data.
- `'sign'` - The key may be used to generate digital signatures.
- `'verify'` - The key may be used to verify digital signatures.
- `'deriveKey'` - The key may be used to derive a new key.
- `'deriveBits'` - The key may be used to derive bits.
- `'wrapKey'` - The key may be used to wrap another key.
- `'unwrapKey'` - The key may be used to unwrap another key.

Valid key usages depend on the key algorithm (identified by `cryptokey.algorithm.name`).

Key Type	'encrypt'	'decrypt'	'sign'	'verify'	'deriveKey'	'deriveBits'	'wrapKey'	'unwrapKey'
----------	-----------	-----------	--------	----------	-------------	--------------	-----------	-------------

Key Type	'encrypt'	'decrypt'	'sign'	'verify'	'deriveKey'	'deriveBits'	'wrapKey'	'unwrapKey'
'AES-CBC'	✓	✓					✓	✓
'AES-CTR'	✓	✓					✓	✓
'AES-GCM'	✓	✓					✓	✓
'AES-KW'							✓	✓
'ECDH'					✓	✓		
'ECDSA'			✓	✓				
'HKDF'					✓	✓		
'HMAC'			✓	✓				
'PBKDF2'					✓	✓		
'RSA-OAEP'	✓	✓					✓	✓
'RSA-PSS'			✓	✓				
'RSASSA-PKCS1-v1_5'			✓	✓				
'NODE-DSA' ¹			✓	✓				
'NODE-DH' ¹					✓	✓		
'NODE-SCRYPT' ¹					✓	✓		
'NODE-ED25519' ¹			✓	✓				
'NODE-ED448' ¹			✓	✓				

¹ Node.js-specific extension.

Class: `CryptoKeyPair`

The `CryptoKeyPair` is a simple dictionary object with `publicKey` and `privateKey` properties, representing an asymmetric key pair.

`cryptoKeyPair.privateKey`

- Type: `<CryptoKey>` A `<CryptoKey>` whose `type` will be `'private'`.

`cryptoKeyPair.publicKey`

- Type: `<CryptoKey>` A `<CryptoKey>` whose `type` will be `'public'`.

Class: `SubtleCrypto`

subtle.decrypt(algorithm, key, data)

- `algorithm`: `<RsaOaepParams>` | `<AesCtrParams>` | `<AesCbcParams>` | `<AesGcmParams>`
- `key`: `<CryptoKey>`
- `data`: `<ArrayBuffer>` | `<TypedArray>` | `<DataView>` | `<Buffer>`
- Returns: `<Promise>` containing `<ArrayBuffer>`

Using the method and parameters specified in `algorithm` and the keying material provided by `key`, `subtle.decrypt()` attempts to decipher the provided `data`. If successful, the returned promise will be resolved with an `<ArrayBuffer>` containing the plaintext result.

The algorithms currently supported include:

- `'RSA-OAEP'`
- `'AES-CTR'`
- `'AES-CBC'`
- `'AES-GCM'`

subtle.deriveBits(algorithm, baseKey, length)

- `algorithm`: `<EcdhKeyDeriveParams>` | `<HkdfParams>` | `<Pbkdf2Params>` | `<NodeDhDeriveBitsParams>` | `<NodeScryptParams>`
- `baseKey`: `<CryptoKey>`
- `length`: `<number>`
- Returns: `<Promise>` containing `<ArrayBuffer>`

Using the method and parameters specified in `algorithm` and the keying material provided by `baseKey`, `subtle.deriveBits()` attempts to generate `length` bits. The Node.js implementation requires that `length` is a multiple of 8. If successful, the returned promise will be resolved with an `<ArrayBuffer>` containing the generated data.

The algorithms currently supported include:

- `'ECDH'`
- `'HKDF'`
- `'PBKDF2'`
- `'NODE-DH'`¹
- `'NODE-SCRYPT'`¹

¹ Node.js-specific extension

subtle.deriveKey(algorithm, baseKey, derivedKeyAlgorithm, extractable, keyUsages)

- `algorithm`: `<EcdhKeyDeriveParams>` | `<HkdfParams>` | `<Pbkdf2Params>` | `<NodeDhDeriveBitsParams>` | `<NodeScryptParams>`
- `baseKey`: `<CryptoKey>`
- `derivedKeyAlgorithm`: `<HmacKeyGenParams>` | `<AesKeyGenParams>`
- `extractable`: `<boolean>`
- `keyUsages`: `<string[]>` See [Key usages](#).
- Returns: `<Promise>` containing `<CryptoKey>`

Using the method and parameters specified in `algorithm`, and the keying material provided by `baseKey`, `subtle.deriveKey()` attempts to generate a new `<CryptoKey>` based on the method and parameters in `derivedKeyAlgorithm`.

Calling `subtle.deriveKey()` is equivalent to calling `subtle.deriveBits()` to generate raw keying material, then passing the result into the `subtle.importKey()` method using the `deriveKeyAlgorithm`, `extractable`, and `keyUsages` parameters as input.

The algorithms currently supported include:

- `'ECDH'`
- `'HKDF'`
- `'PBKDF2'`
- `'NODE-DH'`¹
- `'NODE-SCRYPT'`¹

¹ Node.js-specific extension

`subtle.digest(algorithm, data)`

- `algorithm`: `<string>` | `<Object>`
- `data`: `<ArrayBuffer>` | `<TypedArray>` | `<DataView>` | `<Buffer>`
- Returns: `<Promise>` containing `<ArrayBuffer>`

Using the method identified by `algorithm`, `subtle.digest()` attempts to generate a digest of `data`. If successful, the returned promise is resolved with an `<ArrayBuffer>` containing the computed digest.

If `algorithm` is provided as a `<string>`, it must be one of:

- `'SHA-1'`
- `'SHA-256'`
- `'SHA-384'`
- `'SHA-512'`

If `algorithm` is provided as an `<Object>`, it must have a `name` property whose value is one of the above.

`subtle.encrypt(algorithm, key, data)`

- `algorithm`: `<RsaOaepParams>` | `<AesCtrParams>` | `<AesCbcParams>` | `<AesGcmParams>`
- `key`: `<CryptoKey>`
- Returns: `<Promise>` containing `<ArrayBuffer>`

Using the method and parameters specified by `algorithm` and the keying material provided by `key`, `subtle.encrypt()` attempts to encipher `data`. If successful, the returned promise is resolved with an `<ArrayBuffer>` containing the encrypted result.

The algorithms currently supported include:

- `'RSA-OAEP'`
- `'AES-CTR'`
- `'AES-CBC'`
- `'AES-GCM'`

`subtle.exportKey(format, key)`

- `format`: `<string>` Must be one of `'raw'`, `'pkcs8'`, `'spki'`, `'jwk'`, or `'node.keyObject'`.
- `key`: `<CryptoKey>`
- Returns: `<Promise>` containing `<ArrayBuffer>`, or, if `format` is `'node.keyObject'`, a `<KeyObject>`.

Exports the given key into the specified format, if supported.

If the `<CryptoKey>` is not extractable, the returned promise will reject.

When `format` is either `'pkcs8'` or `'spki'` and the export is successful, the returned promise will be resolved with an `<ArrayBuffer>` containing the exported key data.

When `format` is `'jwk'` and the export is successful, the returned promise will be resolved with a JavaScript object conforming to the [JSON Web Key](#) specification.

The special `'node.keyObject'` value for `format` is a Node.js-specific extension that allows converting a `<CryptoKey>` into a Node.js `<KeyObject>`.

Key Type	'spki'	'pkcs8'	'jwk'	'raw'
'AES-CBC'			✓	✓
'AES-CTR'			✓	✓
'AES-GCM'			✓	✓
'AES-KW'			✓	✓
'ECDH'	✓	✓	✓	✓
'ECDSA'	✓	✓	✓	✓
'HDKF'				
'HMAC'			✓	✓
'PBKDF2'				
'RSA-OAEP'	✓	✓	✓	
'RSA-PSS'	✓	✓	✓	
'RSASSA-PKCS1-v1_5'	✓	✓	✓	
'NODE-DSA' ¹	✓	✓		
'NODE-DH' ¹	✓	✓		
'NODE-SCRYPT' ¹				
'NODE-ED25519' ¹	✓	✓	✓	✓
'NODE-ED448' ¹	✓	✓	✓	✓

¹ Node.js-specific extension

subtle.generateKey(algorithm, extractable, keyUsages)

- `algorithm`: `<RsaHashedKeyGenParams>` | `<EcKeyGenParams>` | `<HmacKeyGenParams>` | `<AesKeyGenParams>` | `<NodeDsaKeyGenParams>` | `<NodeDhKeyGenParams>` | `<NodeEdKeyGenParams>`
- `extractable`: `<boolean>`
- `keyUsages`: `<string[]>` See [Key usages](#).

- Returns: `<Promise>` containing `<CryptoKey>` | `<CryptoKeyPair>`

Using the method and parameters provided in `algorithm`, `subtle.generateKey()` attempts to generate new keying material. Depending the method used, the method may generate either a single `<CryptoKey>` or a `<CryptoKeyPair>`.

The `<CryptoKeyPair>` (public and private key) generating algorithms supported include:

- `'RSASSA-PKCS1-v1_5'`
- `'RSA-PSS'`
- `'RSA-OAEP'`
- `'ECDSA'`
- `'ECDH'`
- `'NODE-DSA'` ¹
- `'NODE-DH'` ¹
- `'NODE-ED25519'` ¹
- `'NODE-ED448'` ¹

The `<CryptoKey>` (secret key) generating algorithms supported include:

- `'HMAC'`
- `'AES-CTR'`
- `'AES-CBC'`
- `'AES-GCM'`
- `'AES-KW'`

¹ Non-standard Node.js extension

subtle.importKey(format, keyData, algorithm, extractable, keyUsages)

- `format`: `<string>` Must be one of `'raw'`, `'pkcs8'`, `'spki'`, `'jwk'`, or `'node.keyObject'`.
- `keyData`: `<ArrayBuffer>` | `<TypedArray>` | `<DataView>` | `<Buffer>` | `<KeyObject>`
- `algorithm`: `<RsaHashedImportParams>` | `<EcKeyImportParams>` | `<HmacImportParams>` | `<AesImportParams>` | `<Pbkdf2ImportParams>` | `<NodeDsaImportParams>` | `<NodeDhImportParams>` | `<NodeScriptImportParams>` | `<NodeEdKeyImportParams>`
- `extractable`: `<boolean>`
- `keyUsages`: `<string[]>` See [Key usages](#).
- Returns: `<Promise>` containing `<CryptoKey>`

The `subtle.importKey()` method attempts to interpret the provided `keyData` as the given `format` to create a `<CryptoKey>` instance using the provided `algorithm`, `extractable`, and `keyUsages` arguments. If the import is successful, the returned promise will be resolved with the created `<CryptoKey>`.

The special `'node.keyObject'` value for `format` is a Node.js-specific extension that allows converting a Node.js `<KeyObject>` into a `<CryptoKey>`.

If importing a `'PBKDF2'` key, `extractable` must be `false`.

The algorithms currently supported include:

Key Type	<code>'spki'</code>	<code>'pkcs8'</code>	<code>'jwk'</code>	<code>'raw'</code>
----------	---------------------	----------------------	--------------------	--------------------

Key Type	'spki'	'pkcs8'	'jwk'	'raw'
'AES-CBC'			✓	✓
'AES-CTR'			✓	✓
'AES-GCM'			✓	✓
'AES-KW'			✓	✓
'ECDH'	✓	✓	✓	✓
'ECDSA'	✓	✓	✓	✓
'HDKF'				✓
'HMAC'			✓	✓
'PBKDF2'				✓
'RSA-OAEP'	✓	✓	✓	
'RSA-PSS'	✓	✓	✓	
'RSASSA-PKCS1-v1_5'	✓	✓	✓	
'NODE-DSA' ¹	✓	✓		
'NODE-DH' ¹	✓	✓		
'NODE-SCRYPT' ¹				✓
'NODE-ED25519' ¹	✓	✓	✓	✓
'NODE-ED448' ¹	✓	✓	✓	✓

¹ Node.js-specific extension

subtle.sign(algorithm, key, data)

- `algorithm`: `<RsaSignParams>` | `<RsaPssParams>` | `<EcdsaParams>` | `<HmacParams>` | `<NodeDsaSignParams>`
- `key`: `<CryptoKey>`
- `data`: `<ArrayBuffer>` | `<TypedArray>` | `<DataView>` | `<Buffer>`
- Returns: `<Promise>` containing `<ArrayBuffer>`

Using the method and parameters given by `algorithm` and the keying material provided by `key`, `subtle.sign()` attempts to generate a cryptographic signature of `data`. If successful, the returned promise is resolved with an `<ArrayBuffer>` containing the generated signature.

The algorithms currently supported include:

- 'RSASSA-PKCS1-v1_5'
- 'RSA-PSS'
- 'ECDSA'
- 'HMAC'
- 'NODE-DSA'¹

- 'NODE-ED25519'¹
- 'NODE-ED448'¹

¹ Non-standard Node.js extension

subtle.unwrapKey(format, wrappedKey, unwrappingKey, unwrapAlgo, unwrappedKeyAlgo, extractable, keyUsages)

- `format`: `<string>` Must be one of 'raw', 'pkcs8', 'spki', or 'jwk'.
- `wrappedKey`: `<ArrayBuffer>` | `<TypedArray>` | `<DataView>` | `<Buffer>`
- `unwrappingKey`: `<CryptoKey>`
- `unwrapAlgo`: `<RsaOaepParams>` | `<AesCtrParams>` | `<AesCbcParams>` | `<AesGcmParams>` | `<AesKwParams>`
- `unwrappedKeyAlgo`: `<RsaHashedImportParams>` | `<EcKeyImportParams>` | `<HmacImportParams>` | `<AesImportParams>`
- `extractable`: `<boolean>`
- `keyUsages`: `<string[]>` See [Key usages](#).
- Returns: `<Promise>` containing `<CryptoKey>`

In cryptography, "wrapping a key" refers to exporting and then encrypting the keying material. The `subtle.unwrapKey()` method attempts to decrypt a wrapped key and create a `<CryptoKey>` instance. It is equivalent to calling `subtle.decrypt()` first on the encrypted key data (using the `wrappedKey`, `unwrapAlgo`, and `unwrappingKey` arguments as input) then passing the results in to the `subtle.importKey()` method using the `unwrappedKeyAlgo`, `extractable`, and `keyUsages` arguments as inputs. If successful, the returned promise is resolved with a `<CryptoKey>` object.

The wrapping algorithms currently supported include:

- 'RSA-OAEP'
- 'AES-CTR'¹
- 'AES-CBC'¹
- 'AES-GCM'¹
- 'AES-KW'¹

The unwrapped key algorithms supported include:

- 'RSASSA-PKCS1-v1_5'
- 'RSA-PSS'
- 'RSA-OAEP'
- 'ECDSA'
- 'ECDH'
- 'HMAC'
- 'AES-CTR'
- 'AES-CBC'
- 'AES-GCM'
- 'AES-KW'
- 'NODE-DSA'¹
- 'NODE-DH'¹

¹ Non-standard Node.js extension

subtle.verify(algorithm, key, signature, data)

- `algorithm`: `<RsaSignParams>` | `<RsaPssParams>` | `<EcdsaParams>` | `<HmacParams>` | `<NodeDsaSignParams>`
- `key`: `<CryptoKey>`
- `signature`: `<ArrayBuffer>` | `<TypedArray>` | `<DataView>` | `<Buffer>`
- `data`: `<ArrayBuffer>` | `<TypedArray>` | `<DataView>` | `<Buffer>`
- Returns: `<Promise>` containing `<boolean>`

Using the method and parameters given in `algorithm` and the keying material provided by `key`, `subtle.verify()` attempts to verify that `signature` is a valid cryptographic signature of `data`. The returned promise is resolved with either `true` or `false`.

The algorithms currently supported include:

- `'RSASSA-PKCS1-v1_5'`
- `'RSA-PSS'`
- `'ECDSA'`
- `'HMAC'`
- `'NODE-DSA'`¹
- `'NODE-ED25519'`¹
- `'NODE-ED448'`¹

¹ Non-standard Node.js extension

subtle.wrapKey(format, key, wrappingKey, wrapAlgo)

- `format`: `<string>` Must be one of `'raw'`, `'pkcs8'`, `'spki'`, or `'jwk'`.
- `key`: `<CryptoKey>`
- `wrappingKey`: `<CryptoKey>`
- `wrapAlgo`: `<RsaOaepParams>` | `<AesCtrParams>` | `<AesCbcParams>` | `<AesGcmParams>` | `<AesKwParams>`
- Returns: `<Promise>` containing `<ArrayBuffer>`

In cryptography, "wrapping a key" refers to exporting and then encrypting the keying material. The `subtle.wrapKey()` method exports the keying material into the format identified by `format`, then encrypts it using the method and parameters specified by `wrapAlgo` and the keying material provided by `wrappingKey`. It is the equivalent to calling `subtle.exportKey()` using `format` and `key` as the arguments, then passing the result to the `subtle.encrypt()` method using `wrappingKey` and `wrapAlgo` as inputs. If successful, the returned promise will be resolved with an `<ArrayBuffer>` containing the encrypted key data.

The wrapping algorithms currently supported include:

- `'RSA-OAEP'`
- `'AES-CTR'`
- `'AES-CBC'`
- `'AES-GCM'`
- `'AES-KW'`

Algorithm Parameters

The algorithm parameter objects define the methods and parameters used by the various `<SubtleCrypto>` methods. While described here as "classes", they are simple JavaScript dictionary objects.

Class: AesCbcParams

aesCbcParams.iv

- Type: `<ArrayBuffer>` | `<TypedArray>` | `<DataView>` | `<Buffer>`

Provides the initialization vector. It must be exactly 16-bytes in length and should be unpredictable and cryptographically random.

aesCbcParams.name

- Type: `<string>` Must be `'AES-CBC'`.

Class: AesCtrParams

aesCtrParams.counter

- Type: `<ArrayBuffer>` | `<TypedArray>` | `<DataView>` | `<Buffer>`

The initial value of the counter block. This must be exactly 16 bytes long.

The `AES-CTR` method uses the rightmost `length` bits of the block as the counter and the remaining bits as the nonce.

aesCtrParams.length

- Type: `<number>` The number of bits in the `aesCtrParams.counter` that are to be used as the counter.

aesCtrParams.name

- Type: `<string>` Must be `'AES-CTR'`.

Class: AesGcmParams

aesGcmParams.additionalData

- Type: `<ArrayBuffer>` | `<TypedArray>` | `<DataView>` | `<Buffer>` | `<undefined>`

With the AES-GCM method, the `additionalData` is extra input that is not encrypted but is included in the authentication of the data. The use of `additionalData` is optional.

aesGcmParams.iv

- Type: `<ArrayBuffer>` | `<TypedArray>` | `<DataView>` | `<Buffer>`

The initialization vector must be unique for every encryption operation using a given key. It is recommended by the AES-GCM specification that this contain at least 12 random bytes.

aesGcmParams.name

- Type: `<string>` Must be `'AES-GCM'`.

aesGcmParams.tagLength

- Type: `<number>` The size in bits of the generated authentication tag. This values must be one of `32`, `64`, `96`, `104`, `112`, `120`, or `128`.
Default: `128`.

Class: AesImportParams

aesImportParams.name

- Type: `<string>` Must be one of `'AES-CTR'`, `'AES-CBC'`, `'AES-GCM'`, or `'AES-KW'`.

Class: AesKeyGenParams

aesKeyGenParams.length

- Type: `<number>`

The length of the AES key to be generated. This must be either `128` , `192` , or `256` .

aesKeyGenParams.name

- Type: `<string>` Must be one of `'AES-CBC'` , `'AES-CTR'` , `'AES-GCM'` , or `'AES-KW'`

Class: AesKwParams

aesKwParams.name

- Type: `<string>` Must be `'AES-KW'` .

Class: EcdhKeyDeriveParams

ecdhKeyDeriveParams.name

- Type: `<string>` Must be `'ECDH'` .

ecdhKeyDeriveParams.public

- Type: `<CryptoKey>`

ECDH key derivation operates by taking as input one parties private key and another parties public key -- using both to generate a common shared secret. The `ecdhKeyDeriveParams.public` property is set to the other parties public key.

Class: EcdsaParams

ecdsaParams.hash

- Type: `<string>` | `<Object>`

If represented as a `<string>` , the value must be one of:

- `'SHA-1'`
- `'SHA-256'`
- `'SHA-384'`
- `'SHA-512'`

If represented as an `<Object>` , the object must have a `name` property whose value is one of the above listed values.

ecdsaParams.name

- Type: `<string>` Must be `'ECDSA'` .

Class: EcKeyGenParams

ecKeyGenParams.name

- Type: `<string>` Must be one of `'ECDSA'` or `'ECDH'` .

ecKeyGenParams.namedCurve

- Type: `<string>` Must be one of `'P-256'`, `'P-384'`, `'P-521'`, `'NODE-ED25519'`, `'NODE-ED448'`, `'NODE-X25519'`, or `'NODE-X448'`.

Class: `EcKeyImportParams`

`ecKeyImportParams.name`

- Type: `<string>` Must be one of `'ECDSA'` or `'ECDH'`.

`ecKeyImportParams.namedCurve`

- Type: `<string>` Must be one of `'P-256'`, `'P-384'`, `'P-521'`, `'NODE-ED25519'`, `'NODE-ED448'`, `'NODE-X25519'`, or `'NODE-X448'`.

Class: `HkdfParams`

`hkdfParams.hash`

- Type: `<string>` | `<Object>`

If represented as a `<string>`, the value must be one of:

- `'SHA-1'`
- `'SHA-256'`
- `'SHA-384'`
- `'SHA-512'`

If represented as an `<Object>`, the object must have a `name` property whose value is one of the above listed values.

`hkdfParams.info`

- Type: `<ArrayBuffer>` | `<TypedArray>` | `<DataView>` | `<Buffer>`

Provides application-specific contextual input to the HKDF algorithm. This can be zero-length but must be provided.

`hkdfParams.name`

- Type: `<string>` Must be `'HKDF'`.

`hkdfParams.salt`

- Type: `<ArrayBuffer>` | `<TypedArray>` | `<DataView>` | `<Buffer>`

The salt value significantly improves the strength of the HKDF algorithm. It should be random or pseudorandom and should be the same length as the output of the digest function (for instance, if using `'SHA-256'` as the digest, the salt should be 256-bits of random data).

Class: `HmacImportParams`

`'hmacImportParams.hash'`

- Type: `<string>` | `<Object>`

If represented as a `<string>`, the value must be one of:

- `'SHA-1'`
- `'SHA-256'`
- `'SHA-384'`

- 'SHA-512'

If represented as an `<Object>`, the object must have a `name` property whose value is one of the above listed values.

hmacImportParams.length

- Type: `<number>`

The optional number of bits in the HMAC key. This is optional and should be omitted for most cases.

hmacImportParams.name

- Type: `<string>` Must be 'HMAC'.

Class: HmacKeyGenParams

hmacKeyGenParams.hash

- Type: `<string>` | `<Object>`

If represented as a `<string>`, the value must be one of:

- 'SHA-1'
- 'SHA-256'
- 'SHA-384'
- 'SHA-512'

If represented as an `<Object>`, the object must have a `name` property whose value is one of the above listed values.

hmacKeyGenParams.length

- Type: `<number>`

The number of bits to generate for the HMAC key. If omitted, the length will be determined by the hash algorithm used. This is optional and should be omitted for most cases.

hmacKeyGenParams.name

- Type: `<string>` Must be 'HMAC'.

Class: HmacParams

hmacParams.name

- Type: `<string>` Must be 'HMAC'.

Class: Pbkdf2ImportParams

pbkdf2ImportParams.name

- Type: `<string>` Must be 'PBKDF2'

Class: Pbkdf2Params

pbkdb2Params.hash

- Type: `<string>` | `<Object>`

If represented as a `<string>`, the value must be one of:

- 'SHA-1'
- 'SHA-256'
- 'SHA-384'
- 'SHA-512'

If represented as an `<Object>`, the object must have a `name` property whose value is one of the above listed values.

pbkdf2Params.iterations

- Type: `<number>`

The number of iterations the PBKDF2 algorithm should make when deriving bits.

pbkdf2Params.name

- Type: `<string>` Must be 'PBKDF2'.

pbkdf2Params.salt

- Type: `<ArrayBuffer>` | `<TypedArray>` | `<DataView>` | `<Buffer>`

Should be at least 16 random or pseudorandom bytes.

Class: RsaHashedImportParams

rsaHashedImportParams.hash

- Type: `<string>` | `<Object>`

If represented as a `<string>`, the value must be one of:

- 'SHA-1'
- 'SHA-256'
- 'SHA-384'
- 'SHA-512'

If represented as an `<Object>`, the object must have a `name` property whose value is one of the above listed values.

rsaHashedImportParams.name

- Type: `<string>` Must be one of 'RSASSA-PKCS1-v1_5', 'RSA-PSS', or 'RSA-OAEP'.

Class: RsaHashedKeyGenParams

rsaHashedKeyGenParams.hash

- Type: `<string>` | `<Object>`

If represented as a `<string>`, the value must be one of:

- 'SHA-1'
- 'SHA-256'
- 'SHA-384'
- 'SHA-512'

If represented as an `<Object>`, the object must have a `name` property whose value is one of the above listed values.

rsaHashedKeyGenParams.modulusLength

- Type: `<number>`

The length in bits of the RSA modulus. As a best practice, this should be at least `2048`.

rsaHashedKeyGenParams.name

- Type: `<string>` Must be one of `'RSASSA-PKCS1-v1_5'`, `'RSA-PSS'`, or `'RSA-OAEP'`.

rsaHashedKeyGenParams.publicExponent

- Type: `<Uint8Array>`

The RSA public exponent. This must be a `<Uint8Array>` containing a big-endian, unsigned integer that must fit within 32-bits. The `<Uint8Array>` may contain an arbitrary number of leading zero-bits. The value must be a prime number. Unless there is reason to use a different value, use `new Uint8Array([1, 0, 1])` (65537) as the public exponent.

Class: RsaOaepParams

rsaOaepParams.label

- Type: `<ArrayBuffer>` | `<TypedArray>` | `<DataView>` | `<Buffer>`

An additional collection of bytes that will not be encrypted, but will be bound to the generated ciphertext.

The `rsaOaepParams.label` parameter is optional.

rsaOaepParams.name

- Type: `<string>` must be `'RSA-OAEP'`.

Class: RsaPssParams

rsaPssParams.name

- Type: `<string>` Must be `'RSA-PSS'`.

rsaPssParams.saltLength

- Type: `<number>`

The length (in bytes) of the random salt to use.

Class: RsaSignParams

rsaSignParams.name

- Type: `<string>` Must be `'RSASSA-PKCS1-v1_5'`

Node.js-specific extensions

The Node.js Web Crypto API extends various aspects of the Web Crypto API. These extensions are consistently identified by prepending names with the `node.` prefix. For instance, the `'node.keyObject'` key format can be used with the `subtle.exportKey()` and `subtle.importKey()` methods to convert between a WebCrypto `<CryptoKey>` object and a Node.js `<KeyObject>`.

Care should be taken when using Node.js-specific extensions as they are not supported by other WebCrypto implementations and reduce the portability of code to other environments.

NODE-DH Algorithm

The `NODE-DH` algorithm is the common implementation of Diffie-Hellman key agreement.

Class: `NodeDhImportParams`

`nodeDhImportParams.name`

- Type: `<string>` Must be `'NODE-DH'`.

Class: `NodeDhKeyGenParams`

`nodeDhKeyGenParams.generator`

- Type: `<number>` A custom generator.

`nodeDhKeyGenParams.group`

- Type: `<string>` The Diffie-Hellman group name.

`nodeDhKeyGenParams.prime`

- Type: `<Buffer>` The prime parameter.

`nodeDhKeyGenParams.primeLength`

- Type: `<number>` The length in bits of the prime.

Class: `NodeDhDeriveBitsParams`

`nodeDhDeriveBitsParams.public`

- Type: `<CryptoKey>` The other parties public key.

NODE-DSA Algorithm

The `NODE-DSA` algorithm is the common implementation of the DSA digital signature algorithm.

Class: `NodeDsaImportParams`

`nodeDsaImportParams.hash`

- Type: `<string> | <Object>`

If represented as a `<string>`, the value must be one of:

- `'SHA-1'`
- `'SHA-256'`
- `'SHA-384'`
- `'SHA-512'`

If represented as an `<Object>`, the object must have a `name` property whose value is one of the above listed values.

`nodeDsaImportParams.name`

- Type: `<string>` Must be `'NODE-DSA'`.

Class: `NodeDsaKeyGenParams`

nodeDsaKeyGenParams.divisorLength

- Type: `<number>`

The optional length in bits of the DSA divisor.

nodeDsaKeyGenParams.hash

- Type: `<string>` | `<Object>`

If represented as a `<string>`, the value must be one of:

- `'SHA-1'`
- `'SHA-256'`
- `'SHA-384'`
- `'SHA-512'`

If represented as an `<Object>`, the object must have a `name` property whose value is one of the above listed values.

nodeDsaKeyGenParams.modulusLength

- Type: `<number>`

The length in bits of the DSA modulus. As a best practice, this should be at least `2048`.

nodeDsaKeyGenParams.name

- Type: `<string>` Must be `'NODE-DSA'`.

Class: NodeDsaSignParams

nodeDsaSignParams.name

- Type: `<string>` Must be `'NODE-DSA'`

NODE-ED25519 and NODE-ED448 Algorithms

Class: NodeEdKeyGenParams

nodeEdKeyGenParams.name

- Type: `<string>` Must be one of `'NODE-ED25519'`, `'NODE-ED448'` or `'ECDH'`.

nodeEdKeyGenParams.namedCurve

- Type: `<string>` Must be one of `'NODE-ED25519'`, `'NODE-ED448'`, `'NODE-X25519'`, or `'NODE-X448'`.

Class: NodeEdKeyImportParams

nodeEdKeyImportParams.name

- Type: `<string>` Must be one of `'NODE-ED25519'` or `'NODE-ED448'` if importing an `Ed25519` or `Ed448` key, or `'ECDH'` if importing an `X25519` or `X448` key.

nodeEdKeyImportParams.namedCurve

- Type: `<string>` Must be one of `'NODE-ED25519'`, `'NODE-ED448'`, `'NODE-X25519'`, or `'NODE-X448'`.

nodeEdKeyImportParams.public

- Type: `<boolean>`

The `public` parameter is used to specify that the `'raw'` format key is to be interpreted as a public key. **Default:** `false` .

NODE-SCRIPT Algorithm

The `NODE-SCRIPT` algorithm is the common implementation of the script key derivation algorithm.

Class: NodeScriptImportParams

nodeScriptImportParams.name

- Type: `<string>` Must be `'NODE-SCRIPT'` .

Class: NodeScriptParams

nodeScriptParams.encoding

- Type: `<string>` The string encoding when `salt` is a string.

nodeScriptParams.maxmem

- Type: `<number>` Memory upper bound. It is an error when (approximately) $127 * N * r > \text{maxmem}$. **Default:** `32 * 1024 * 1024` .

nodeScriptParams.N

- Type: `<number>` The CPU/memory cost parameter. Must e a power of two greater than 1. **Default:** `16384` .

nodeScriptParams.p

- Type: `<number>` Parallelization parameter. **Default:** `1` .

nodeScriptParams.r

- Type: `<number>` Block size parameter. **Default:** `8` .

nodeScriptParams.salt

- Type: `<string>` | `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>`

Web Streams API

Stability: 1 - Experimental

An implementation of the [WHATWG Streams Standard](#) .

```
import {
  ReadableStream,
  WritableStream,
  TransformStream,
} from 'node:stream/web';const {
  ReadableStream,
```

```
WritableStream,  
TransformStream,  
} = require('stream/web');
```

Overview

The **WHATWG Streams Standard** (or "web streams") defines an API for handling streaming data. It is similar to the Node.js **Streams** API but emerged later and has become the "standard" API for streaming data across many JavaScript environments.

There are three primary types of objects

- **ReadableStream** - Represents a source of streaming data.
- **WritableStream** - Represents a destination for streaming data.
- **TransformStream** - Represents an algorithm for transforming streaming data.

Example **ReadableStream**

This example creates a simple **ReadableStream** that pushes the current `performance.now()` timestamp once every second forever. An `async iterable` is used to read the data from the stream.

```
import {  
  ReadableStream  
} from 'node:stream/web';  
  
import {  
  setInterval as every  
} from 'node:timers/promises';  
  
import {  
  performance  
} from 'node:perf_hooks';  
  
const SECOND = 1000;  
  
const stream = new ReadableStream({  
  async start(controller) {  
    for await (const _ of every(SECOND))  
      controller.enqueue(performance.now());  
  }  
});  
  
for await (const value of stream)  
  console.log(value);const {  
  ReadableStream  
} = require('stream/web');  
  
const {  
  setInterval: every  
} = require('timers/promises');
```



```

const {
  performance
} = require('perf_hooks');

const SECOND = 1000;

const stream = new ReadableStream({
  async start(controller) {
    for await (const _ of every(SECOND))
      controller.enqueue(performance.now());
  }
});

(async () => {
  for await (const value of stream)
    console.log(value);
})();

```

API

Class: ReadableStream

new ReadableStream([underlyingSource [, strategy]])

- **underlyingSource** `<Object>`
 - **start** `<Function>` A user-defined function that is invoked immediately when the `ReadableStream` is created.
 - **controller** `<ReadableStreamDefaultController> | <ReadableByteStreamController>`
 - Returns: `undefined` or a promise fulfilled with `undefined`.
 - **pull** `<Function>` A user-defined function that is called repeatedly when the `ReadableStream` internal queue is not full. The operation may be sync or async. If async, the function will not be called again until the previously returned promise is fulfilled.
 - **controller** `<ReadableStreamDefaultController> | <ReadableByteStreamController>`
 - Returns: A promise fulfilled with `undefined`.
 - **cancel** `<Function>` A user-defined function that is called when the `ReadableStream` is canceled.
 - **reason** `<any>`
 - Returns: A promise fulfilled with `undefined`.
 - **type** `<string>` Must be `'bytes'` or `undefined`.
 - **autoAllocateChunkSize** `<number>` Used only when **type** is equal to `'bytes'`.
- **strategy** `<Object>`
 - **highWaterMark** `<number>` The maximum internal queue size before backpressure is applied.
 - **size** `<Function>` A user-defined function used to identify the size of each chunk of data.
 - **chunk** `<any>`
 - Returns: `<number>`

readableStream.locked

- Type: `<boolean>` Set to `true` if there is an active reader for this `<ReadableStream>`.

The `readableStream.locked` property is `false` by default, and is switch to `true` while there is an active reader consuming the stream's data.

readableStream.cancel([reason])

- `reason` `<any>`
- Returns: A promise fulfilled with `undefined` once cancelation has been completed.

readableStream.getReader([options])

- `options` `<Object>`
 - `mode` `<string>` `'byob'` or `undefined`
- Returns: `<ReadableStreamDefaultReader>` | `<ReadableStreamBYOBReader>`

```
import { ReadableStream } from 'node:stream/web';

const stream = new ReadableStream();

const reader = stream.getReader();

console.log(await reader.read());const { ReadableStream } = require('stream/web');

const stream = new ReadableStream();

const reader = stream.getReader();

reader.read().then(console.log);
```

Causes the `readableStream.locked` to be `true`.

readableStream.pipeThrough(transform[, options])

- `transform` `<Object>`
 - `readable` `<ReadableStream>` The `ReadableStream` to which `transform.writable` will push the potentially modified data is receives from this `ReadableStream`.
 - `writable` `<WritableStream>` The `WritableStream` to which this `ReadableStream`'s data will be written.
- `options` `<Object>`
 - `preventAbort` `<boolean>` When `true`, errors in this `ReadableStream` will not cause `transform.writable` to be aborted.
 - `preventCancel` `<boolean>` When `true`, errors in the destination `transform.writable` is not cause this `ReadableStream` to be canceled.
 - `preventClose` `<boolean>` When `true`, closing this `ReadableStream` will no cause `transform.writable` to be closed.
 - `signal` `<AbortSignal>` Allows the transfer of data to be canceled using an `<AbortController>`.
- Returns: `<ReadableStream>` From `transform.readable`.

Connects this `<ReadableStream>` to the pair of `<ReadableStream>` and `<WritableStream>` provided in the `transform` argument such that the data from this `<ReadableStream>` is written in to `transform.writable`, possibly transformed, then pushed to `transform.readable`. Once the pipeline is configured, `transform.readable` is returned.

Causes the `readableStream.locked` to be `true` while the pipe operation is active.

```
import {
  ReadableStream,
  TransformStream,
} from 'node:stream/web';
```

```

const stream = new ReadableStream({
  start(controller) {
    controller.enqueue('a');
  },
});

const transform = new TransformStream({
  transform(chunk, controller) {
    controller.enqueue(chunk.toUpperCase());
  }
});

const transformedStream = stream.pipeThrough(transform);

for await (const chunk of transformedStream)
  console.log(chunk);const {
  ReadableStream,
  TransformStream,
} = require('stream/web');

const stream = new ReadableStream({
  start(controller) {
    controller.enqueue('a');
  },
});

const transform = new TransformStream({
  transform(chunk, controller) {
    controller.enqueue(chunk.toUpperCase());
  }
});

const transformedStream = stream.pipeThrough(transform);

(async () => {
  for await (const chunk of transformedStream)
    console.log(chunk);
})();

```

readableStream.pipeTo(destination, options)

- **destination** `<WritableStream>` A `<WritableStream>` to which this `ReadableStream`'s data will be written.
- **options** `<Object>`
 - **preventAbort** `<boolean>` When `true`, errors in this `ReadableStream` will not cause `transform.writable` to be aborted.
 - **preventCancel** `<boolean>` When `true`, errors in the destination `transform.writable` is not cause this `ReadableStream` to be canceled.
 - **preventClose** `<boolean>` When `true`, closing this `ReadableStream` will no cause `transform.writable` to be closed.
 - **signal** `<AbortSignal>` Allows the transfer of data to be canceled using an `<AbortController>`.
- Returns: A promise fulfilled with `undefined`

Causes the `readableStream.locked` to be `true` while the pipe operation is active.

`readableStream.tee()`

- Returns: `<ReadableStream[]>`

Returns a pair of new `<ReadableStream>` instances to which this `ReadableStream`'s data will be forwarded. Each will receive the same data.

Causes the `readableStream.locked` to be `true`.

`readableStream.values([options])`

- `options` `<Object>`
 - `preventCancel` `<boolean>` When `true`, prevents the `<ReadableStream>` from being closed when the async iterator abruptly terminates. **Defaults:** `false`

Creates and returns an async iterator usable for consuming this `ReadableStream`'s data.

Causes the `readableStream.locked` to be `true` while the async iterator is active.

```
import { Buffer } from 'node:buffer';

const stream = new ReadableStream(getSomeSource());

for await (const chunk of stream.values({ preventCancel: true }))
  console.log(Buffer.from(chunk).toString());
```

Async Iteration

The `<ReadableStream>` object supports the async iterator protocol using `for await` syntax.

```
import { Buffer } from 'buffer';

const stream = new ReadableStream(getSomeSource());

for await (const chunk of stream)
  console.log(Buffer.from(chunk).toString());
```

The async iterator will consume the `<ReadableStream>` until it terminates.

By default, if the async iterator exits early (via either a `break`, `return`, or a `throw`), the `<ReadableStream>` will be closed. To prevent automatic closing of the `<ReadableStream>`, use the `readableStream.values()` method to acquire the async iterator and set the `preventCancel` option to `true`.

The `<ReadableStream>` must not be locked (that is, it must not have an existing active reader). During the async iteration, the `<ReadableStream>` will be locked.

Transferring with `postMessage()`

A `<ReadableStream>` instance can be transferred using a `<MessagePort>`.

```
const stream = new ReadableStream(getReadableSourceSomehow());

const { port1, port2 } = new MessageChannel();
```

```
port1.onmessage = ({ data }) => {
  data.getReader().read().then((chunk) => {
    console.log(chunk);
  });
};

port2.postMessage(stream, [stream]);
```

Class: ReadableStreamDefaultReader

By default, calling `readableStream.getReader()` with no arguments will return an instance of `ReadableStreamDefaultReader`. The default reader treats the chunks of data passed through the stream as opaque values, which allows the `<ReadableStream>` to work with generally any JavaScript value.

`new ReadableStreamDefaultReader(stream)`

- `stream` `<ReadableStream>`

Creates a new `<ReadableStreamDefaultReader>` that is locked to the given `<ReadableStream>`.

`readableStreamDefaultReader.cancel([reason])`

- `reason` `<any>`
- Returns: A promise fulfilled with `undefined`.

Cancels the `<ReadableStream>` and returns a promise that is fulfilled when the underlying stream has been canceled.

`readableStreamDefaultReader.closed`

- Type: `<Promise>` Fulfilled with `undefined` when the associated `<ReadableStream>` is closed or this reader's lock is released.

`readableStreamDefaultReader.read()`

- Returns: A promise fulfilled with an object:
 - `value` `<ArrayBuffer>`
 - `done` `<boolean>`

Requests the next chunk of data from the underlying `<ReadableStream>` and returns a promise that is fulfilled with the data once it is available.

`readableStreamDefaultReader.releaseLock()`

Releases this reader's lock on the underlying `<ReadableStream>`.

Class: ReadableStreamBYOBReader

The `ReadableStreamBYOBReader` is an alternative consumer for byte-oriented `<ReadableStream>`'s (those that are created with `underlyingSource.type` set equal to `'bytes'` when the `ReadableStream` was created).

The `BYOB` is short for "bring your own buffer". This is a pattern that allows for more efficient reading of byte-oriented data that avoids extraneous copying.

```
import {
  open
} from 'node:fs/promises';
```

```

import {
  ReadableStream
} from 'node:stream/web';

import { Buffer } from 'node:buffer';

class Source {
  type = 'bytes';
  autoAllocateChunkSize = 1024;

  async start(controller) {
    this.file = await open(new URL(import.meta.url));
    this.controller = controller;
  }

  async pull(controller) {
    const view = controller.byobRequest?.view;
    const {
      bytesRead,
    } = await this.file.read({
      buffer: view,
      offset: view.byteOffset,
      length: view.byteLength
    });

    if (bytesRead === 0) {
      await this.file.close();
      this.controller.close();
    }
    controller.byobRequest.respond(bytesRead);
  }
}

const stream = new ReadableStream(new Source());

async function read(stream) {
  const reader = stream.getReader({ mode: 'byob' });

  const chunks = [];
  let result;
  do {
    result = await reader.read(Buffer.alloc(100));
    if (result.value !== undefined)
      chunks.push(Buffer.from(result.value));
  } while (!result.done);

  return Buffer.concat(chunks);
}

```

```
const data = await read(stream);
console.log(Buffer.from(data).toString());
```

new ReadableStreamBYOBReader(stream)

- `stream` `<ReadableStream>`

Creates a new `ReadableStreamBYOBReader` that is locked to the given `<ReadableStream>`.

readableStreamBYOBReader.cancel([reason])

- `reason` `<any>`
- Returns: A promise fulfilled with `undefined`.

Cancels the `<ReadableStream>` and returns a promise that is fulfilled when the underlying stream has been canceled.

readableStreamBYOBReader.closed

- Type: `<Promise>` Fulfilled with `undefined` when the associated `<ReadableStream>` is closed or this reader's lock is released.

readableStreamBYOBReader.read(view)

- `view` `<Buffer>` | `<TypedArray>` | `<DataView>`
- Returns: A promise fulfilled with an object:
 - `value` `<ArrayBuffer>`
 - `done` `<boolean>`

Requests the next chunk of data from the underlying `<ReadableStream>` and returns a promise that is fulfilled with the data once it is available.

Do not pass a pooled `<Buffer>` object instance in to this method. Pooled `Buffer` objects are created using `Buffer.allocUnsafe()`, or `Buffer.from()`, or are often returned by various `fs` module callbacks. These types of `Buffer`s use a shared underlying `<ArrayBuffer>` object that contains all of the data from all of the pooled `Buffer` instances. When a `Buffer`, `<TypedArray>`, or `<DataView>` is passed in to `readableStreamBYOBReader.read()`, the view's underlying `ArrayBuffer` is *detached*, invalidating all existing views that may exist on that `ArrayBuffer`. This can have disastrous consequences for your application.

readableStreamBYOBReader.releaseLock()

Releases this reader's lock on the underlying `<ReadableStream>`.

Class: ReadableStreamDefaultController

Every `<ReadableStream>` has a controller that is responsible for the internal state and management of the stream's queue. The `ReadableStreamDefaultController` is the default controller implementation for `ReadableStream`s that are not byte-oriented.

readableStreamDefaultController.close()

Closes the `<ReadableStream>` to which this controller is associated.

readableStreamDefaultController.desiredSize

- Type: `<number>`

Returns the amount of data remaining to fill the `<ReadableStream>`'s queue.

readableStreamDefaultController.enqueue(chunk)

- `chunk` `<any>`

Appends a new chunk of data to the `<ReadableStream>` 's queue.

`readableStreamDefaultController.error(error)`

- `error` `<any>`

Signals an error that causes the `<ReadableStream>` to error and close.

Class: `ReadableByteStreamController`

Every `<ReadableStream>` has a controller that is responsible for the internal state and management of the stream's queue. The `ReadableByteStreamController` is for byte-oriented `ReadableStream`s.

`readableByteStreamController.byobRequest`

- Type: `<ReadableStreamBYOBRequest>`

`readableByteStreamController.close()`

Closes the `<ReadableStream>` to which this controller is associated.

`readableByteStreamController.desiredSize`

- Type: `<number>`

Returns the amount of data remaining to fill the `<ReadableStream>` 's queue.

`readableByteStreamController.enqueue(chunk)`

- `chunk` : `<Buffer>` | `<TypedArray>` | `<DataView>`

Appends a new chunk of data to the `<ReadableStream>` 's queue.

`readableByteStreamController.error(error)`

- `error` `<any>`

Signals an error that causes the `<ReadableStream>` to error and close.

Class: `ReadableStreamBYOBRequest`

When using `ReadableByteStreamController` in byte-oriented streams, and when using the `ReadableStreamBYOBReader`, the `readableByteStreamController.byobRequest` property provides access to a `ReadableStreamBYOBRequest` instance that represents the current read request. The object is used to gain access to the `ArrayBuffer` / `TypedArray` that has been provided for the read request to fill, and provides methods for signaling that the data has been provided.

`readableStreamBYOBRequest.respond(bytesWritten)`

- `bytesWritten` `<number>`

Signals that a `bytesWritten` number of bytes have been written to `readableStreamBYOBRequest.view`.

`readableStreamBYOBRequest.respondWithNewView(view)`

- `view` `<Buffer>` | `<TypedArray>` | `<DataView>`

Signals that the request has been fulfilled with bytes written to a new `Buffer`, `TypedArray`, or `DataView`.

`readableStreamBYOBRequest.view`

- Type: `<Buffer>` | `<TypedArray>` | `<DataView>`

Class: WritableStream

The `WritableStream` is a destination to which stream data is sent.

```
import {
  WritableStream
} from 'node:stream/web';

const stream = new WritableStream({
  write(chunk) {
    console.log(chunk);
  }
});

await stream.getWriter().write('Hello World');
```

new WritableStream([underlyingSink[, strategy]])

- `underlyingSink` `<Object>`
 - `start` `<Function>` A user-defined function that is invoked immediately when the `WritableStream` is created.
 - `controller` `<WritableStreamDefaultController>`
 - Returns: `undefined` or a promise fulfilled with `undefined`.
 - `write` `<Function>` A user-defined function that is invoked when a chunk of data has been written to the `WritableStream`.
 - `chunk` `<any>`
 - `controller` `<WritableStreamDefaultController>`
 - Returns: A promise fulfilled with `undefined`.
 - `close` `<Function>` A user-defined function that is called when the `WritableStream` is closed.
 - Returns: A promise fulfilled with `undefined`.
 - `abort` `<Function>` A user-defined function that is called to abruptly close the `WritableStream`.
 - `reason` `<any>`
 - Returns: A promise fulfilled with `undefined`.
 - `type` `<any>` The `type` option is reserved for future use and *must* be `undefined`.
- `strategy` `<Object>`
 - `highWaterMark` `<number>` The maximum internal queue size before backpressure is applied.
 - `size` `<Function>` A user-defined function used to identify the size of each chunk of data.
 - `chunk` `<any>`
 - Returns: `<number>`

writableStream.abort([reason])

- `reason` `<any>`
- Returns: A promise fulfilled with `undefined`.

Abruptly terminates the `WritableStream`. All queued writes will be canceled with their associated promises rejected.

writableStream.close()

- Returns: A promise fulfilled with `undefined`.

Closes the `WritableStream` when no additional writes are expected.

`writableStream.getWriter()`

- Returns: `<WritableStreamDefaultWriter>`

Creates and creates a new writer instance that can be used to write data into the `WritableStream`.

`writableStream.locked`

- Type: `<boolean>`

The `writableStream.locked` property is `false` by default, and is switched to `true` while there is an active writer attached to this `WritableStream`.

Transferring with `postMessage()`

A `<WritableStream>` instance can be transferred using a `<MessagePort>`.

```
const stream = new WritableStream(getWritableSinkSomehow());

const { port1, port2 } = new MessageChannel();

port1.onmessage = ({ data }) => {
  data.getWriter().write('hello');
};

port2.postMessage(stream, [stream]);
```

Class: `WritableStreamDefaultWriter`

`new WritableStreamDefaultWriter(stream)`

- `stream` `<WritableStream>`

Creates a new `WritableStreamDefaultWriter` that is locked to the given `WritableStream`.

`writableStreamDefaultWriter.abort([reason])`

- `reason` `<any>`
- Returns: A promise fulfilled with `undefined`.

Abruptly terminates the `WritableStream`. All queued writes will be canceled with their associated promises rejected.

`writableStreamDefaultWriter.close()`

- Returns: A promise fulfilled with `undefined`.

Closes the `WritableStream` when no additional writes are expected.

`writableStreamDefaultWriter.closed`

- Type: A promise that is fulfilled with `undefined` when the associated `<WritableStream>` is closed or this writer's lock is released.

`writableStreamDefaultWriter.desiredSize`

- Type: `<number>`

The amount of data required to fill the `<WritableStream>`'s queue.

`writableStreamDefaultWriter.ready`

- type: A promise that is fulfilled with `undefined` when the writer is ready to be used.

`writableStreamDefaultWriter.releaseLock()`

Releases this writer's lock on the underlying `<ReadableStream>`.

`writableStreamDefaultWriter.write([chunk])`

- `chunk` : `<any>`
- Returns: A promise fulfilled with `undefined`.

Appends a new chunk of data to the `<WritableStream>`'s queue.

Class: `WritableStreamDefaultController`

The `WritableStreamDefaultController` manage's the `<WritableStream>`'s internal state.

`writableStreamDefaultController.abortReason`

- Type: `<any>` The `reason` value passed to `writableStream.abort()`.

`writableStreamDefaultController.error(error)`

- `error` `<any>`

Called by user-code to signal that an error has occurred while processing the `WritableStream` data. When called, the `<WritableStream>` will be aborted, with currently pending writes canceled.

`writableStreamDefaultController.signal`

- Type: `<AbortSignal>` An `AbortSignal` that can be used to cancel pending write or close operations when a `<WritableStream>` is aborted.

Class: `TransformStream`

A `TransformStream` consists of a `<ReadableStream>` and a `<WritableStream>` that are connected such that the data written to the `WritableStream` is received, and potentially transformed, before being pushed into the `ReadableStream`'s queue.

```
import {
  TransformStream
} from 'node:stream/web';

const transform = new TransformStream({
  transform(chunk, controller) {
    controller.enqueue(chunk.toUpperCase());
  }
});

await Promise.all([
  transform.writable.getWriter().write('A'),
  transform.readable.getReader().read(),
]);
```

`new TransformStream([transformer[, writableStrategy[, readableStrategy]])`

- `transformer` `<Object>`
 - `start` `<Function>` A user-defined function that is invoked immediately when the `TransformStream` is created.
 - `controller` `<TransformStreamDefaultController>`
 - Returns: `undefined` or a promise fulfilled with `undefined`
 - `transform` `<Function>` A user-defined function that receives, and potentially modifies, a chunk of data written to `transformStream.writable`, before forwarding that on to `transformStream.readable`.
 - `chunk` `<any>`
 - `controller` `<TransformStreamDefaultController>`
 - Returns: A promise fulfilled with `undefined`.
 - `flush` `<Function>` A user-defined function that is called immediately before the writable side of the `TransformStream` is closed, signaling the end of the transformation process.
 - `controller` `<TransformStreamDefaultController>`
 - Returns: A promise fulfilled with `undefined`.
 - `readableType` `<any>` the `readableType` option is reserved for future use and *must* be `undefined`.
 - `writableType` `<any>` the `writableType` option is reserved for future use and *must* be `undefined`.
- `writableStrategy` `<Object>`
 - `highWaterMark` `<number>` The maximum internal queue size before backpressure is applied.
 - `size` `<Function>` A user-defined function used to identify the size of each chunk of data.
 - `chunk` `<any>`
 - Returns: `<number>`
- `readableStrategy` `<Object>`
 - `highWaterMark` `<number>` The maximum internal queue size before backpressure is applied.
 - `size` `<Function>` A user-defined function used to identify the size of each chunk of data.
 - `chunk` `<any>`
 - Returns: `<number>`

`transformStream.readable`

- Type: `<ReadableStream>`

`transformStream.writable`

- Type: `<WritableStream>`

Transferring with `postMessage()`

A `<TransformStream>` instance can be transferred using a `<MessagePort>`.

```
const stream = new TransformStream();

const { port1, port2 } = new MessageChannel();

port1.onmessage = ({ data }) => {
  const { writable, readable } = data;
  // ...
};
```

```
port2.postMessage(stream, [stream]);
```

Class: TransformStreamDefaultController

The `TransformStreamDefaultController` manages the internal state of the `TransformStream`.

`transformStreamDefaultController.desiredSize`

- Type: `<number>`

The amount of data required to fill the readable side's queue.

`transformStreamDefaultController.enqueue([chunk])`

- `chunk` `<any>`

Appends a chunk of data to the readable side's queue.

`transformStreamDefaultController.error([reason])`

- `reason` `<any>`

Signals to both the readable and writable side that an error has occurred while processing the transform data, causing both sides to be abruptly closed.

`transformStreamDefaultController.terminate()`

Closes the readable side of the transport and causes the writable side to be abruptly closed with an error.

Class: ByteLengthQueuingStrategy

`new ByteLengthQueuingStrategy(options)`

- `options` `<Object>`
 - `highWaterMark` `<number>`

`byteLengthQueuingStrategy.highWaterMark`

- Type: `<number>`

`byteLengthQueuingStrategy.size`

- Type: `<Function>`
 - `chunk` `<any>`
 - Returns: `<number>`

Class: CountQueuingStrategy

`new CountQueuingStrategy(options)`

- `options` `<Object>`
 - `highWaterMark` `<number>`

`countQueuingStrategy.highWaterMark`

- Type: `<number>`
-

countQueuingStrategy.size

- Type: `<Function>`
 - `chunk` `<any>`
 - Returns: `<number>`

Class: TextEncoderStream

new TextEncoderStream()

Creates a new `TextEncoderStream` instance.

textEncoderStream.encoding

- Type: `<string>`

The encoding supported by the `TextEncoderStream` instance.

textEncoderStream.readable

- Type: `<ReadableStream>`

textEncoderStream.writable

- Type: `<WritableStream>`

Class: TextDecoderStream

new TextDecoderStream([encoding[, options]])

- `encoding` `<string>` Identifies the encoding that this `TextDecoder` instance supports. **Default:** `'utf-8'`.
- `options` `<Object>`
 - `fatal` `<boolean>` `true` if decoding failures are fatal.
 - `ignoreBOM` `<boolean>` When `true`, the `TextDecoderStream` will include the byte order mark in the decoded result. When `false`, the byte order mark will be removed from the output. This option is only used when `encoding` is `'utf-8'`, `'utf-16be'` or `'utf-16le'`. **Default:** `false`.

Creates a new `TextDecoderStream` instance.

textDecoderStream.encoding

- Type: `<string>`

The encoding supported by the `TextDecoderStream` instance.

textDecoderStream.fatal

- Type: `<boolean>`

The value will be `true` if decoding errors result in a `TypeError` being thrown.

textDecoderStream.ignoreBOM

- Type: `<boolean>`

The value will be `true` if the decoding result will include the byte order mark.

textDecoderStream.readable

- Type: `<ReadableStream>`

`textDecoderStream.writable`

- Type: `<WritableStream>`

Class: `CompressionStream`

`new CompressionStream(format)`

- `format` `<string>` One of either `'deflate'` or `'gzip'`.

`compressionStream.readable`

- Type: `<ReadableStream>`

`compressionStream.writable`

- Type: `<WritableStream>`

Class: `DecompressionStream`

`new DecompressionStream(format)`

- `format` `<string>` One of either `'deflate'` or `'gzip'`.

`decompressionStream.readable`

- Type: `<ReadableStream>`

`decompressionStream.writable`

- Type: `<WritableStream>`

Utility Consumers

The utility consumer functions provide common options for consuming streams.

They are accessed using:

```
import {
  arrayBuffer,
  blob,
  json,
  text,
} from 'node:stream/consumers'; const {
  arrayBuffer,
  blob,
  json,
  text,
} = require('stream/consumers');
```

`streamConsumers.arrayBuffer(stream)`

- `stream` `<ReadableStream>` | `<stream.Readable>` | `<AsyncIterator>`

- Returns: `<Promise>` Fulfills with an `ArrayBuffer` containing the full contents of the stream.

`streamConsumers.blob(stream)`

- `stream` `<ReadableStream>` | `<stream.Readable>` | `<AsyncIterator>`
- Returns: `<Promise>` Fulfills with a `<Blob>` containing the full contents of the stream.

`streamConsumers.buffer(stream)`

- `stream` `<ReadableStream>` | `<stream.Readable>` | `<AsyncIterator>`
- Returns: `<Promise>` Fulfills with a `<Buffer>` containing the full contents of the stream.

`streamConsumers.json(stream)`

- `stream` `<ReadableStream>` | `<stream.Readable>` | `<AsyncIterator>`
- Returns: `<Promise>` Fulfills with the contents of the stream parsed as a UTF-8 encoded string that is then passed through `JSON.parse()`.

`streamConsumers.text(stream)`

- `stream` `<ReadableStream>` | `<stream.Readable>` | `<AsyncIterator>`
- Returns: `<Promise>` Fulfills with the contents of the stream parsed as a UTF-8 encoded string.

Worker threads

Stability: 2 - Stable

Source Code: [lib/worker_threads.js](#)

The `worker_threads` module enables the use of threads that execute JavaScript in parallel. To access it:

```
const worker = require('worker_threads');
```

Workers (threads) are useful for performing CPU-intensive JavaScript operations. They do not help much with I/O-intensive work. The Node.js built-in asynchronous I/O operations are more efficient than Workers can be.

Unlike `child_process` or `cluster`, `worker_threads` can share memory. They do so by transferring `ArrayBuffer` instances or sharing `SharedArrayBuffer` instances.

```
const {
  Worker, isMainThread, parentPort, workerData
} = require('worker_threads');

if (isMainThread) {
  module.exports = function parseJSAsync(script) {
    return new Promise((resolve, reject) => {
      const worker = new Worker(__filename, {
        workerData: script
      });
      worker.on('message', resolve);
    });
  };
}
```



```

worker.on('error', reject);
worker.on('exit', (code) => {
  if (code !== 0)
    reject(new Error(`Worker stopped with exit code ${code}`));
});
});
};
} else {
  const { parse } = require('some-js-parsing-library');
  const script = workerData;
  parentPort.postMessage(parse(script));
}

```

The above example spawns a Worker thread for each `parse()` call. In actual practice, use a pool of Workers for these kinds of tasks. Otherwise, the overhead of creating Workers would likely exceed their benefit.

When implementing a worker pool, use the [AsyncResource](#) API to inform diagnostic tools (e.g. to provide asynchronous stack traces) about the correlation between tasks and their outcomes. See "Using AsyncResource for a Worker thread pool" in the [async_hooks](#) documentation for an example implementation.

Worker threads inherit non-process-specific options by default. Refer to [Worker constructor options](#) to know how to customize worker thread options, specifically `argv` and `execArgv` options.

worker.getEnvironmentData(key)

Stability: 1 - Experimental

- `key` [<any>](#) Any arbitrary, cloneable JavaScript value that can be used as a [<Map>](#) key.
- Returns: [<any>](#)

Within a worker thread, `worker.getEnvironmentData()` returns a clone of data passed to the spawning thread's `worker.setEnvironmentData()`. Every new `Worker` receives its own copy of the environment data automatically.

```

const {
  Worker,
  isMainThread,
  setEnvironmentData,
  getEnvironmentData,
} = require('worker_threads');

if (isMainThread) {
  setEnvironmentData('Hello', 'World!');
  const worker = new Worker(__filename);
} else {
  console.log(getEnvironmentData('Hello')); // Prints 'World!'.
}

```

worker.isMainThread

- `<boolean>`

Is `true` if this code is not running inside of a `Worker` thread.

```
const { Worker, isMainThread } = require('worker_threads');

if (isMainThread) {
  // This re-loads the current file inside a Worker instance.
  new Worker(__filename);
} else {
  console.log('Inside Worker!');
  console.log(isMainThread); // Prints 'false'.
}
```

worker.markAsUntransferable(object)

Mark an object as not transferable. If `object` occurs in the transfer list of a `port.postMessage()` call, it is ignored.

In particular, this makes sense for objects that can be cloned, rather than transferred, and which are used by other objects on the sending side. For example, Node.js marks the `ArrayBuffer` s it uses for its `Buffer` pool with this.

This operation cannot be undone.

```
const { MessageChannel, markAsUntransferable } = require('worker_threads');

const pooledBuffer = new ArrayBuffer(8);
const typedArray1 = new Uint8Array(pooledBuffer);
const typedArray2 = new Float64Array(pooledBuffer);

markAsUntransferable(pooledBuffer);

const { port1 } = new MessageChannel();
port1.postMessage(typedArray1, [ typedArray1.buffer ]);

// The following line prints the contents of typedArray1 -- it still owns
// its memory and has been cloned, not transferred. Without
// `markAsUntransferable()`, this would print an empty Uint8Array.
// typedArray2 is intact as well.
console.log(typedArray1);
console.log(typedArray2);
```

There is no equivalent to this API in browsers.

worker.moveMessagePortToContext(port, contextifiedSandbox)

- `port` `<MessagePort>` The message port to transfer.
- `contextifiedSandbox` `<Object>` A `contextified` object as returned by the `vm.createContext()` method.
- Returns: `<MessagePort>`

Transfer a `MessagePort` to a different `vm` Context. The original `port` object is rendered unusable, and the returned `MessagePort` instance takes its place.

The returned `MessagePort` is an object in the target context and inherits from its global `Object` class. Objects passed to the `port.onmessage()` listener are also created in the target context and inherit from its global `Object` class.

However, the created `MessagePort` no longer inherits from `EventTarget`, and only `port.onmessage()` can be used to receive events using it.

worker.parentPort

- `<null> | <MessagePort>`

If this thread is a `Worker`, this is a `MessagePort` allowing communication with the parent thread. Messages sent using `parentPort.postMessage()` are available in the parent thread using `worker.on('message')`, and messages sent from the parent thread using `worker.postMessage()` are available in this thread using `parentPort.on('message')`.

```
const { Worker, isMainThread, parentPort } = require('worker_threads');

if (isMainThread) {
  const worker = new Worker(__filename);
  worker.once('message', (message) => {
    console.log(message); // Prints 'Hello, world!'.
  });
  worker.postMessage('Hello, world!');
} else {
  // When a message from the parent thread is received, send it back:
  parentPort.once('message', (message) => {
    parentPort.postMessage(message);
  });
}
```

worker.receiveMessageOnPort(port)

- `port <MessagePort> | <BroadcastChannel>`
- Returns: `<Object> | <undefined>`

Receive a single message from a given `MessagePort`. If no message is available, `undefined` is returned, otherwise an object with a single `message` property that contains the message payload, corresponding to the oldest message in the `MessagePort`'s queue.

```
const { MessageChannel, receiveMessageOnPort } = require('worker_threads');
const { port1, port2 } = new MessageChannel();
port1.postMessage({ hello: 'world' });

console.log(receiveMessageOnPort(port2));
// Prints: { message: { hello: 'world' } }
console.log(receiveMessageOnPort(port2));
// Prints: undefined
```

When this function is used, no `'message'` event is emitted and the `onmessage` listener is not invoked.

worker.resourceLimits

- `<Object>`
 - `maxYoungGenerationSizeMb` `<number>`
 - `maxOldGenerationSizeMb` `<number>`
 - `codeRangeSizeMb` `<number>`
 - `stackSizeMb` `<number>`

Provides the set of JS engine resource constraints inside this Worker thread. If the `resourceLimits` option was passed to the `Worker` constructor, this matches its values.

If this is used in the main thread, its value is an empty object.

worker.SHARE_ENV

- `<symbol>`

A special value that can be passed as the `env` option of the `Worker` constructor, to indicate that the current thread and the Worker thread should share read and write access to the same set of environment variables.

```
const { Worker, SHARE_ENV } = require('worker_threads');
new Worker('process.env.SET_IN_WORKER = "foo"', { eval: true, env: SHARE_ENV })
  .on('exit', () => {
    console.log(process.env.SET_IN_WORKER); // Prints 'foo'.
  });
```

worker.setEnvironmentData(key[, value])

Stability: 1 - Experimental

- `key` `<any>` Any arbitrary, cloneable JavaScript value that can be used as a `<Map>` key.
- `value` `<any>` Any arbitrary, cloneable JavaScript value that will be cloned and passed automatically to all new `Worker` instances. If `value` is passed as `undefined`, any previously set value for the `key` will be deleted.

The `worker.setEnvironmentData()` API sets the content of `worker.getEnvironmentData()` in the current thread and all new `Worker` instances spawned from the current context.

worker.threadId

- `<integer>`

An integer identifier for the current thread. On the corresponding worker object (if there is any), it is available as `worker.threadId`. This value is unique for each `Worker` instance inside a single process.

worker.workerData

An arbitrary JavaScript value that contains a clone of the data passed to this thread's `Worker` constructor.

The data is cloned as if using `postMessage()`, according to the [HTML structured clone algorithm](#).

```
const { Worker, isMainThread, workerData } = require('worker_threads');

if (isMainThread) {
  const worker = new Worker(__filename, { workerData: 'Hello, world!' });
} else {
  console.log(workerData); // Prints 'Hello, world!'.
}
```

Class: BroadcastChannel extends EventTarget

Stability: 1 - Experimental

Instances of `BroadcastChannel` allow asynchronous one-to-many communication with all other `BroadcastChannel` instances bound to the same channel name.

```
'use strict';

const {
  isMainThread,
  BroadcastChannel,
  Worker
} = require('worker_threads');

const bc = new BroadcastChannel('hello');

if (isMainThread) {
  let c = 0;
  bc.onmessage = (event) => {
    console.log(event.data);
    if (++c === 10) bc.close();
  };
  for (let n = 0; n < 10; n++)
    new Worker(__filename);
} else {
  bc.postMessage('hello from every worker');
  bc.close();
}
```

new BroadcastChannel(name)

- `name` [<any>](#) The name of the channel to connect to. Any JavaScript value that can be converted to a string using `${name}` is permitted.

broadcastChannel.close()

Closes the `BroadcastChannel` connection.

broadcastChannel.onmessage

- Type: `<Function>` Invoked with a single `MessageEvent` argument when a message is received.

broadcastChannel.onmessageerror

- Type: `<Function>` Invoked with a received message cannot be deserialized.

broadcastChannel.postMessage(message)

- `message` `<any>` Any cloneable JavaScript value.

broadcastChannel.ref()

Opposite of `unref()`. Calling `ref()` on a previously `unref()` ed `BroadcastChannel` does *not* let the program exit if it's the only active handle left (the default behavior). If the port is `ref()` ed, calling `ref()` again has no effect.

broadcastChannel.unref()

Calling `unref()` on a `BroadcastChannel` allows the thread to exit if this is the only active handle in the event system. If the `BroadcastChannel` is already `unref()` ed calling `unref()` again has no effect.

Class: MessageChannel

Instances of the `worker.MessageChannel` class represent an asynchronous, two-way communications channel. The `MessageChannel` has no methods of its own. `new MessageChannel()` yields an object with `port1` and `port2` properties, which refer to linked `MessagePort` instances.

```
const { MessageChannel } = require('worker_threads');

const { port1, port2 } = new MessageChannel();
port1.on('message', (message) => console.log('received', message));
port2.postMessage({ foo: 'bar' });
// Prints: received { foo: 'bar' } from the `port1.on('message')` listener
```

Class: MessagePort

- Extends: `<EventTarget>`

Instances of the `worker.MessagePort` class represent one end of an asynchronous, two-way communications channel. It can be used to transfer structured data, memory regions and other `MessagePort` s between different `Worker` s.

This implementation matches `browser MessagePort` s.

Event: 'close'

The `'close'` event is emitted once either side of the channel has been disconnected.

```
const { MessageChannel } = require('worker_threads');
const { port1, port2 } = new MessageChannel();

// Prints:
//   foobar
```

```
// closed!
port2.on('message', (message) => console.log(message));
port2.on('close', () => console.log('closed!'));

port1.postMessage('foobar');
port1.close();
```

Event: 'message'

- `value` `<any>` The transmitted value

The 'message' event is emitted for any incoming message, containing the cloned input of `port.postMessage()`.

Listeners on this event receive a clone of the `value` parameter as passed to `postMessage()` and no further arguments.

Event: 'messageerror'

- `error` `<Error>` An Error object

The 'messageerror' event is emitted when deserializing a message failed.

Currently, this event is emitted when there is an error occurring while instantiating the posted JS object on the receiving end. Such situations are rare, but can happen, for instance, when certain Node.js API objects are received in a `vm.Context` (where Node.js APIs are currently unavailable).

port.close()

Disables further sending of messages on either side of the connection. This method can be called when no further communication will happen over this `MessagePort`.

The 'close' event is emitted on both `MessagePort` instances that are part of the channel.

port.postMessage(value[, transferList])

- `value` `<any>`
- `transferList` `<Object[]>`

Sends a JavaScript value to the receiving side of this channel. `value` is transferred in a way which is compatible with the [HTML structured clone algorithm](#).

In particular, the significant differences to `JSON` are:

- `value` may contain circular references.
- `value` may contain instances of builtin JS types such as `RegExp` s, `BigInt` s, `Map` s, `Set` s, etc.
- `value` may contain typed arrays, both using `ArrayBuffer` s and `SharedArrayBuffer` s.
- `value` may contain `WebAssembly.Module` instances.
- `value` may not contain native (C++-backed) objects other than:
 - `<CryptoKey>` s,
 - `<FileHandle>` s,
 - `<Histogram>` s,
 - `<KeyObject>` s,
 - `<MessagePort>` s,
 - `<net.BlockList>` s,

- `<net.SocketAddress>` es,
- `<X509Certificate>` s.

```
const { MessageChannel } = require('worker_threads');
const { port1, port2 } = new MessageChannel();

port1.on('message', (message) => console.log(message));

const circularData = {};
circularData.foo = circularData;
// Prints: { foo: [Circular] }
port2.postMessage(circularData);
```

`transferList` may be a list of `ArrayBuffer`, `MessagePort` and `FileHandle` objects. After transferring, they are not usable on the sending side of the channel anymore (even if they are not contained in `value`). Unlike with `child processes`, transferring handles such as network sockets is currently not supported.

If `value` contains `SharedArrayBuffer` instances, those are accessible from either thread. They cannot be listed in `transferList`.

`value` may still contain `ArrayBuffer` instances that are not in `transferList`; in that case, the underlying memory is copied rather than moved.

```
const { MessageChannel } = require('worker_threads');
const { port1, port2 } = new MessageChannel();

port1.on('message', (message) => console.log(message));

const uint8Array = new Uint8Array([ 1, 2, 3, 4 ]);
// This posts a copy of `uint8Array`:
port2.postMessage(uint8Array);
// This does not copy data, but renders `uint8Array` unusable:
port2.postMessage(uint8Array, [ uint8Array.buffer ]);

// The memory for the `sharedUint8Array` is accessible from both the
// original and the copy received by `.on('message')`:
const sharedUint8Array = new Uint8Array(new SharedArrayBuffer(4));
port2.postMessage(sharedUint8Array);

// This transfers a freshly created message port to the receiver.
// This can be used, for example, to create communication channels between
// multiple `Worker` threads that are children of the same parent thread.
const otherChannel = new MessageChannel();
port2.postMessage({ port: otherChannel.port1 }, [ otherChannel.port1 ]);
```

The message object is cloned immediately, and can be modified after posting without having side effects.

For more information on the serialization and deserialization mechanisms behind this API, see the [serialization API of the v8 module](#).

Considerations when transferring TypedArrays and Buffers

All `TypedArray` and `Buffer` instances are views over an underlying `ArrayBuffer`. That is, it is the `ArrayBuffer` that actually stores the raw data while the `TypedArray` and `Buffer` objects provide a way of viewing and manipulating the data. It is possible and common for multiple views to be created over the same `ArrayBuffer` instance. Great care must be taken when using a transfer list to transfer an `ArrayBuffer` as doing so causes all `TypedArray` and `Buffer` instances that share that same `ArrayBuffer` to become unusable.

```
const ab = new ArrayBuffer(10);

const u1 = new Uint8Array(ab);
const u2 = new Uint16Array(ab);

console.log(u2.length); // prints 5

port.postMessage(u1, [u1.buffer]);

console.log(u2.length); // prints 0
```

For `Buffer` instances, specifically, whether the underlying `ArrayBuffer` can be transferred or cloned depends entirely on how instances were created, which often cannot be reliably determined.

An `ArrayBuffer` can be marked with `markAsUntransferable()` to indicate that it should always be cloned and never transferred.

Depending on how a `Buffer` instance was created, it may or may not own its underlying `ArrayBuffer`. An `ArrayBuffer` must not be transferred unless it is known that the `Buffer` instance owns it. In particular, for `Buffer`'s created from the internal `Buffer` pool (using, for instance `Buffer.from()` or `Buffer.allocUnsafe()`), transferring them is not possible and they are always cloned, which sends a copy of the entire `Buffer` pool. This behavior may come with unintended higher memory usage and possible security concerns.

See `Buffer.allocUnsafe()` for more details on `Buffer` pooling.

The `ArrayBuffer`'s for `Buffer` instances created using `Buffer.alloc()` or `Buffer.allocUnsafeSlow()` can always be transferred but doing so renders all other existing views of those `ArrayBuffer`'s unusable.

Considerations when cloning objects with prototypes, classes, and accessors

Because object cloning uses the `HTML structured clone algorithm`, non-enumerable properties, property accessors, and object prototypes are not preserved. In particular, `Buffer` objects will be read as plain `Uint8Array`'s on the receiving side, and instances of JavaScript classes will be cloned as plain JavaScript objects.

```
const b = Symbol('b');

class Foo {
  #a = 1;
  constructor() {
    this[b] = 2;
    this.c = 3;
  }

  get d() { return 4; }
}

const { port1, port2 } = new MessageChannel();

port1.onmessage = ({ data }) => console.log(data);
```

```
port2.postMessage(new Foo());

// Prints: { c: 3 }
```

This limitation extends to many built-in objects, such as the global `URL` object:

```
const { port1, port2 } = new MessageChannel();

port1.onmessage = ({ data }) => console.log(data);

port2.postMessage(new URL('https://example.org'));

// Prints: { }
```

port.ref()

Opposite of `unref()`. Calling `ref()` on a previously `unref()` ed port does *not* let the program exit if it's the only active handle left (the default behavior). If the port is `ref()` ed, calling `ref()` again has no effect.

If listeners are attached or removed using `.on('message')`, the port is `ref()` ed and `unref()` ed automatically depending on whether listeners for the event exist.

port.start()

Starts receiving messages on this `MessagePort`. When using this port as an event emitter, this is called automatically once `'message'` listeners are attached.

This method exists for parity with the Web `MessagePort` API. In Node.js, it is only useful for ignoring messages when no event listener is present. Node.js also diverges in its handling of `.onmessage`. Setting it automatically calls `.start()`, but unsetting it lets messages queue up until a new handler is set or the port is discarded.

port.unref()

Calling `unref()` on a port allows the thread to exit if this is the only active handle in the event system. If the port is already `unref()` ed calling `unref()` again has no effect.

If listeners are attached or removed using `.on('message')`, the port is `ref()` ed and `unref()` ed automatically depending on whether listeners for the event exist.

Class: Worker

- Extends: `<EventEmitter>`

The `Worker` class represents an independent JavaScript execution thread. Most Node.js APIs are available inside of it.

Notable differences inside a Worker environment are:

- The `process.stdin`, `process.stdout` and `process.stderr` may be redirected by the parent thread.
- The `require('worker_threads').isMainThread` property is set to `false`.
- The `require('worker_threads').parentPort` message port is available.
- `process.exit()` does not stop the whole program, just the single thread, and `process.abort()` is not available.

- `process.chdir()` and `process` methods that set group or user ids are not available.
- `process.env` is a copy of the parent thread's environment variables, unless otherwise specified. Changes to one copy are not visible in other threads, and are not visible to native add-ons (unless `worker.SHARE_ENV` is passed as the `env` option to the `Worker` constructor).
- `process.title` cannot be modified.
- Signals are not delivered through `process.on('...')`.
- Execution may stop at any point as a result of `worker.terminate()` being invoked.
- IPC channels from parent processes are not accessible.
- The `trace_events` module is not supported.
- Native add-ons can only be loaded from multiple threads if they fulfill `certain conditions`.

Creating `Worker` instances inside of other `Worker`s is possible.

Like `Web Workers` and the `cluster module`, two-way communication can be achieved through inter-thread message passing. Internally, a `Worker` has a built-in pair of `MessagePort`s that are already associated with each other when the `Worker` is created. While the `MessagePort` object on the parent side is not directly exposed, its functionalities are exposed through `worker.postMessage()` and the `worker.on('message')` event on the `Worker` object for the parent thread.

To create custom messaging channels (which is encouraged over using the default global channel because it facilitates separation of concerns), users can create a `MessageChannel` object on either thread and pass one of the `MessagePort`s on that `MessageChannel` to the other thread through a pre-existing channel, such as the global one.

See `port.postMessage()` for more information on how messages are passed, and what kind of JavaScript values can be successfully transported through the thread barrier.

```
const assert = require('assert');
const {
  Worker, MessageChannel, MessagePort, isMainThread, parentPort
} = require('worker_threads');
if (isMainThread) {
  const worker = new Worker(__filename);
  const subChannel = new MessageChannel();
  worker.postMessage({ hereIsYourPort: subChannel.port1 }, [subChannel.port1]);
  subChannel.port2.on('message', (value) => {
    console.log('received:', value);
  });
} else {
  parentPort.once('message', (value) => {
    assert(value.hereIsYourPort instanceof MessagePort);
    value.hereIsYourPort.postMessage('the worker is sending this');
    value.hereIsYourPort.close();
  });
}
```

new Worker(filename[, options])

- `filename` `<string> | <URL>` The path to the Worker's main script or module. Must be either an absolute path or a relative path (i.e. relative to the current working directory) starting with `./` or `../`, or a WHATWG URL object using `file:` or `data:` protocol. When using a `data: URL`, the data is interpreted based on MIME type using the `ECMAScript module loader`. If `options.eval` is `true`, this is a string containing JavaScript code rather than a path.
- `options` `<Object>`

- `argv <any[]>` List of arguments which would be stringified and appended to `process.argv` in the worker. This is mostly similar to the `workerData` but the values are available on the global `process.argv` as if they were passed as CLI options to the script.
- `env <Object>` If set, specifies the initial value of `process.env` inside the Worker thread. As a special value, `worker.SHARE_ENV` may be used to specify that the parent thread and the child thread should share their environment variables; in that case, changes to one thread's `process.env` object affect the other thread as well. **Default:** `process.env`.
- `eval <boolean>` If `true` and the first argument is a `string`, interpret the first argument to the constructor as a script that is executed once the worker is online.
- `execArgv <string[]>` List of node CLI options passed to the worker. V8 options (such as `--max-old-space-size`) and options that affect the process (such as `--title`) are not supported. If set, this is provided as `process.execArgv` inside the worker. By default, options are inherited from the parent thread.
- `stdin <boolean>` If this is set to `true`, then `worker.stdin` provides a writable stream whose contents appear as `process.stdin` inside the Worker. By default, no data is provided.
- `stdout <boolean>` If this is set to `true`, then `worker.stdout` is not automatically piped through to `process.stdout` in the parent.
- `stderr <boolean>` If this is set to `true`, then `worker.stderr` is not automatically piped through to `process.stderr` in the parent.
- `workerData <any>` Any JavaScript value that is cloned and made available as `require('worker_threads').workerData`. The cloning occurs as described in the [HTML structured clone algorithm](#), and an error is thrown if the object cannot be cloned (e.g. because it contains `function`s).
- `trackUnmanagedFds <boolean>` If this is set to `true`, then the Worker tracks raw file descriptors managed through `fs.open()` and `fs.close()`, and closes them when the Worker exits, similar to other resources like network sockets or file descriptors managed through the `FileHandle` API. This option is automatically inherited by all nested `Worker`s. **Default:** `true`.
- `transferList <Object[]>` If one or more `MessagePort`-like objects are passed in `workerData`, a `transferList` is required for those items or `ERR_MISSING_MESSAGE_PORT_IN_TRANSFER_LIST` is thrown. See `port.postMessage()` for more information.
- `resourceLimits <Object>` An optional set of resource limits for the new JS engine instance. Reaching these limits leads to termination of the `Worker` instance. These limits only affect the JS engine, and no external data, including no `ArrayBuffer`s. Even if these limits are set, the process may still abort if it encounters a global out-of-memory situation.
 - `maxOldGenerationSizeMb <number>` The maximum size of the main heap in MB.
 - `maxYoungGenerationSizeMb <number>` The maximum size of a heap space for recently created objects.
 - `codeRangeSizeMb <number>` The size of a pre-allocated memory range used for generated code.
 - `stackSizeMb <number>` The default maximum stack size for the thread. Small values may lead to unusable Worker instances. **Default:** `4`.

Event: 'error'

- `err <Error>`

The `'error'` event is emitted if the worker thread throws an uncaught exception. In that case, the worker is terminated.

Event: 'exit'

- `exitCode <integer>`

The `'exit'` event is emitted once the worker has stopped. If the worker exited by calling `process.exit()`, the `exitCode` parameter is the passed exit code. If the worker was terminated, the `exitCode` parameter is `1`.

This is the final event emitted by any `Worker` instance.

Event: 'message'

- `value <any>` The transmitted value

The `'message'` event is emitted when the worker thread has invoked `require('worker_threads').parentPort.postMessage()`. See the `port.on('message')` event for more details.

All messages sent from the worker thread are emitted before the `'exit'` event is emitted on the `Worker` object.

Event: 'messageerror'

- `error` `<Error>` An Error object

The `'messageerror'` event is emitted when deserializing a message failed.

Event: 'online'

The `'online'` event is emitted when the worker thread has started executing JavaScript code.

worker.getHeapSnapshot()

- Returns: `<Promise>` A promise for a Readable Stream containing a V8 heap snapshot

Returns a readable stream for a V8 snapshot of the current state of the Worker. See `v8.getHeapSnapshot()` for more details.

If the Worker thread is no longer running, which may occur before the `'exit'` event is emitted, the returned `Promise` is rejected immediately with an `ERR_WORKER_NOT_RUNNING` error.

worker.performance

An object that can be used to query performance information from a worker instance. Similar to `perf_hooks.performance`.

performance.eventLoopUtilization([utilization1[, utilization2]])

- `utilization1` `<Object>` The result of a previous call to `eventLoopUtilization()`.
- `utilization2` `<Object>` The result of a previous call to `eventLoopUtilization()` prior to `utilization1`.
- Returns `<Object>`
 - `idle` `<number>`
 - `active` `<number>`
 - `utilization` `<number>`

The same call as `perf_hooks.eventLoopUtilization()`, except the values of the worker instance are returned.

One difference is that, unlike the main thread, bootstrapping within a worker is done within the event loop. So the event loop utilization is immediately available once the worker's script begins execution.

An `idle` time that does not increase does not indicate that the worker is stuck in bootstrap. The following examples shows how the worker's entire lifetime never accumulates any `idle` time, but is still be able to process messages.

```
const { Worker, isMainThread, parentPort } = require('worker_threads');

if (isMainThread) {
  const worker = new Worker(__filename);
  setInterval(() => {
    worker.postMessage('hi');
    console.log(worker.performance.eventLoopUtilization());
  }, 100).unref();
  return;
}
```

```
parentPort.on('message', () => console.log('msg')).unref();
(function r(n) {
  if (--n < 0) return;
  const t = Date.now();
  while (Date.now() - t < 300);
  setImmediate(r, n);
})(10);
```

The event loop utilization of a worker is available only after the `'online' event` emitted, and if called before this, or after the `'exit' event`, then all properties have the value of `0`.

worker.postMessage(value[, transferList])

- `value` `<any>`
- `transferList` `<Object[]>`

Send a message to the worker that is received via `require('worker_threads').parentPort.on('message')`. See `port.postMessage()` for more details.

worker.ref()

Opposite of `unref()`, calling `ref()` on a previously `unref()` ed worker does *not* let the program exit if it's the only active handle left (the default behavior). If the worker is `ref()` ed, calling `ref()` again has no effect.

worker.resourceLimits

- `<Object>`
 - `maxYoungGenerationSizeMb` `<number>`
 - `maxOldGenerationSizeMb` `<number>`
 - `codeRangeSizeMb` `<number>`
 - `stackSizeMb` `<number>`

Provides the set of JS engine resource constraints for this Worker thread. If the `resourceLimits` option was passed to the `Worker` constructor, this matches its values.

If the worker has stopped, the return value is an empty object.

worker.stderr

- `<stream.Readable>`

This is a readable stream which contains data written to `process.stderr` inside the worker thread. If `stderr: true` was not passed to the `Worker` constructor, then data is piped to the parent thread's `process.stderr` stream.

worker.stdin

- `<null> | <stream.Writable>`

If `stdin: true` was passed to the `Worker` constructor, this is a writable stream. The data written to this stream will be made available in the worker thread as `process.stdin`.

worker.stdout

- `<stream.Readable>`

This is a readable stream which contains data written to `process.stdout` inside the worker thread. If `stdout: true` was not passed to the `Worker` constructor, then data is piped to the parent thread's `process.stdout` stream.

`worker.terminate()`

- Returns: `<Promise>`

Stop all JavaScript execution in the worker thread as soon as possible. Returns a Promise for the exit code that is fulfilled when the `'exit'` event is emitted.

`worker.threadId`

- `<integer>`

An integer identifier for the referenced thread. Inside the worker thread, it is available as `require('worker_threads').threadId`. This value is unique for each `Worker` instance inside a single process.

`worker.unref()`

Calling `unref()` on a worker allows the thread to exit if this is the only active handle in the event system. If the worker is already `unref()` ed calling `unref()` again has no effect.

Notes

Synchronous blocking of stdio

`Worker` s utilize message passing via `<MessagePort>` to implement interactions with `stdio`. This means that `stdio` output originating from a `Worker` can get blocked by synchronous code on the receiving end that is blocking the Node.js event loop.

```
import {
  Worker,
  isMainThread,
} from 'worker_threads';

if (isMainThread) {
  new Worker(new URL(import.meta.url));
  for (let n = 0; n < 1e10; n++) {}
} else {
  // This output will be blocked by the for loop in the main thread.
  console.log('foo');
}'use strict';

const {
  Worker,
  isMainThread,
} = require('worker_threads');

if (isMainThread) {
  new Worker(__filename);
  for (let n = 0; n < 1e10; n++) {}
} else {
  // This output will be blocked by the for loop in the main thread.
```

```
console.log('foo');  
}
```

Launching worker threads from preload scripts

Take care when launching worker threads from preload scripts (scripts loaded and run using the `-r` command line flag). Unless the `execArgv` option is explicitly set, new Worker threads automatically inherit the command line flags from the running process and will preload the same preload scripts as the main thread. If the preload script unconditionally launches a worker thread, every thread spawned will spawn another until the application crashes.

Zlib

Stability: 2 - Stable

Source Code: [lib/zlib.js](#)

The `zlib` module provides compression functionality implemented using Gzip, Deflate/Inflate, and Brotli.

To access it:

```
const zlib = require('zlib');
```

Compression and decompression are built around the Node.js [Streams API](#).

Compressing or decompressing a stream (such as a file) can be accomplished by piping the source stream through a `zlib Transform` stream into a destination stream:

```
const { createGzip } = require('zlib');  
const { pipeline } = require('stream');  
const {  
  createReadStream,  
  createWriteStream  
} = require('fs');  
  
const gzip = createGzip();  
const source = createReadStream('input.txt');  
const destination = createWriteStream('input.txt.gz');  
  
pipeline(source, gzip, destination, (err) => {  
  if (err) {  
    console.error('An error occurred:', err);  
    process.exitCode = 1;  
  }  
});  
  
// Or, Promisified  
  
const { promisify } = require('util');  
const pipe = promisify(pipeline);
```



```

async function do_gzip(input, output) {
  const gzip = createGzip();
  const source = createReadStream(input);
  const destination = createWriteStream(output);
  await pipe(source, gzip, destination);
}

do_gzip('input.txt', 'input.txt.gz')
  .catch((err) => {
    console.error('An error occurred:', err);
    process.exitCode = 1;
  });

```

It is also possible to compress or decompress data in a single step:

```

const { deflate, unzip } = require('zlib');

const input = '.....';
deflate(input, (err, buffer) => {
  if (err) {
    console.error('An error occurred:', err);
    process.exitCode = 1;
  }
  console.log(buffer.toString('base64'));
});

const buffer = Buffer.from('eJzT0yMAAGTvBe8=', 'base64');
unzip(buffer, (err, buffer) => {
  if (err) {
    console.error('An error occurred:', err);
    process.exitCode = 1;
  }
  console.log(buffer.toString());
});

// Or, Promisified

const { promisify } = require('util');
const do_unzip = promisify(unzip);

do_unzip(buffer)
  .then((buf) => console.log(buf.toString()))
  .catch((err) => {
    console.error('An error occurred:', err);
    process.exitCode = 1;
  });

```

Threadpool usage and performance considerations

All `zlib` APIs, except those that are explicitly synchronous, use the Node.js internal threadpool. This can lead to surprising effects and performance limitations in some applications.

Creating and using a large number of `zlib` objects simultaneously can cause significant memory fragmentation.

```
const zlib = require('zlib');

const payload = Buffer.from('This is some data');

// WARNING: DO NOT DO THIS!
for (let i = 0; i < 30000; ++i) {
  zlib.deflate(payload, (err, buffer) => {});
}
```

In the preceding example, 30,000 deflate instances are created concurrently. Because of how some operating systems handle memory allocation and deallocation, this may lead to significant memory fragmentation.

It is strongly recommended that the results of compression operations be cached to avoid duplication of effort.

Compressing HTTP requests and responses

The `zlib` module can be used to implement support for the `gzip`, `deflate` and `br` content-encoding mechanisms defined by [HTTP](#).

The HTTP `Accept-Encoding` header is used within an http request to identify the compression encodings accepted by the client. The `Content-Encoding` header is used to identify the compression encodings actually applied to a message.

The examples given below are drastically simplified to show the basic concept. Using `zlib` encoding can be expensive, and the results ought to be cached. See [Memory usage tuning](#) for more information on the speed/memory/compression tradeoffs involved in `zlib` usage.

```
// Client request example
const zlib = require('zlib');
const http = require('http');
const fs = require('fs');
const { pipeline } = require('stream');

const request = http.get({ host: 'example.com',
                           path: '/',
                           port: 80,
                           headers: { 'Accept-Encoding': 'br,gzip,deflate' } });

request.on('response', (response) => {
  const output = fs.createWriteStream('example.com_index.html');

  const onError = (err) => {
    if (err) {
      console.error('An error occurred:', err);
      process.exitCode = 1;
    }
  };

  switch (response.headers['content-encoding']) {
    case 'br':
```

```

    pipeline(response, zlib.createBrotliDecompress(), output, onError);
    break;
// Or, just use zlib.createUnzip() to handle both of the following cases:
case 'gzip':
    pipeline(response, zlib.createGunzip(), output, onError);
    break;
case 'deflate':
    pipeline(response, zlib.createInflate(), output, onError);
    break;
default:
    pipeline(response, output, onError);
    break;
}
});

```

```

// server example
// Running a gzip operation on every request is quite expensive.
// It would be much more efficient to cache the compressed buffer.
const zlib = require('zlib');
const http = require('http');
const fs = require('fs');
const { pipeline } = require('stream');

http.createServer((request, response) => {
  const raw = fs.createReadStream('index.html');
  // Store both a compressed and an uncompressed version of the resource.
  response.setHeader('Vary', 'Accept-Encoding');
  let acceptEncoding = request.headers['accept-encoding'];
  if (!acceptEncoding) {
    acceptEncoding = '';
  }

  const onError = (err) => {
    if (err) {
      // If an error occurs, there's not much we can do because
      // the server has already sent the 200 response code and
      // some amount of data has already been sent to the client.
      // The best we can do is terminate the response immediately
      // and log the error.
      response.end();
      console.error('An error occurred:', err);
    }
  };

  // Note: This is not a conformant accept-encoding parser.
  // See https://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.3
  if (/^bdeflate\b/.test(acceptEncoding)) {
    response.writeHead(200, { 'Content-Encoding': 'deflate' });
    pipeline(raw, zlib.createDeflate(), response, onError);
  } else if (/^bgzip\b/.test(acceptEncoding)) {

```

```

    response.writeHead(200, { 'Content-Encoding': 'gzip' });
    pipeline(raw, zlib.createGzip(), response, onError);
  } else if (/^bbr\b/.test(acceptEncoding)) {
    response.writeHead(200, { 'Content-Encoding': 'br' });
    pipeline(raw, zlib.createBrotliCompress(), response, onError);
  } else {
    response.writeHead(200, {});
    pipeline(raw, response, onError);
  }
}).listen(1337);

```

By default, the `zlib` methods will throw an error when decompressing truncated data. However, if it is known that the data is incomplete, or the desire is to inspect only the beginning of a compressed file, it is possible to suppress the default error handling by changing the flushing method that is used to decompress the last chunk of input data:

```

// This is a truncated version of the buffer from the above examples
const buffer = Buffer.from('eJzT0yMA', 'base64');

zlib.unzip(
  buffer,
  // For Brotli, the equivalent is zlib.constants.BROTLI_OPERATION_FLUSH.
  { finishFlush: zlib.constants.Z_SYNC_FLUSH },
  (err, buffer) => {
    if (err) {
      console.error('An error occurred:', err);
      process.exitCode = 1;
    }
    console.log(buffer.toString());
  });

```

This will not change the behavior in other error-throwing situations, e.g. when the input data has an invalid format. Using this method, it will not be possible to determine whether the input ended prematurely or lacks the integrity checks, making it necessary to manually check that the decompressed result is valid.

Memory usage tuning

For zlib-based streams

From `zlib/zconf.h`, modified for Node.js usage:

The memory requirements for deflate are (in bytes):

```
(1 << (windowBits + 2)) + (1 << (memLevel + 9))
```

That is: 128K for `windowBits` = 15 + 128K for `memLevel` = 8 (default values) plus a few kilobytes for small objects.

For example, to reduce the default memory requirements from 256K to 128K, the options should be set to:

```
const options = { windowBits: 14, memLevel: 7 };
```

This will, however, generally degrade compression.

The memory requirements for inflate are (in bytes) `1 << windowBits`. That is, 32K for `windowBits = 15` (default value) plus a few kilobytes for small objects.

This is in addition to a single internal output slab buffer of size `chunkSize`, which defaults to 16K.

The speed of `zlib` compression is affected most dramatically by the `level` setting. A higher level will result in better compression, but will take longer to complete. A lower level will result in less compression, but will be much faster.

In general, greater memory usage options will mean that Node.js has to make fewer calls to `zlib` because it will be able to process more data on each `write` operation. So, this is another factor that affects the speed, at the cost of memory usage.

For Brotli-based streams

There are equivalents to the `zlib` options for Brotli-based streams, although these options have different ranges than the `zlib` ones:

- `zlib`'s `level` option matches Brotli's `BROTLI_PARAM_QUALITY` option.
- `zlib`'s `windowBits` option matches Brotli's `BROTLI_PARAM_LGWIN` option.

See [below](#) for more details on Brotli-specific options.

Flushing

Calling `.flush()` on a compression stream will make `zlib` return as much output as currently possible. This may come at the cost of degraded compression quality, but can be useful when data needs to be available as soon as possible.

In the following example, `flush()` is used to write a compressed partial HTTP response to the client:

```
const zlib = require('zlib');
const http = require('http');
const { pipeline } = require('stream');

http.createServer((request, response) => {
  // For the sake of simplicity, the Accept-Encoding checks are omitted.
  response.writeHead(200, { 'content-encoding': 'gzip' });
  const output = zlib.createGzip();
  let i;

  pipeline(output, response, (err) => {
    if (err) {
      // If an error occurs, there's not much we can do because
      // the server has already sent the 200 response code and
      // some amount of data has already been sent to the client.
      // The best we can do is terminate the response immediately
      // and log the error.
      clearInterval(i);
      response.end();
      console.error('An error occurred:', err);
    }
  });

  i = setInterval(() => {
```

```
output.write(`The current time is ${Date()}\n`, () => {
  // The data has been passed to zlib, but the compression algorithm may
  // have decided to buffer the data for more efficient compression.
  // Calling .flush() will make the data available as soon as the client
  // is ready to receive it.
  output.flush();
});
}, 1000);
}).listen(1337);
```

Constants

zlib constants

All of the constants defined in `zlib.h` are also defined on `require('zlib').constants`. In the normal course of operations, it will not be necessary to use these constants. They are documented so that their presence is not surprising. This section is taken almost directly from the [zlib documentation](#).

Previously, the constants were available directly from `require('zlib')`, for instance `zlib.Z_NO_FLUSH`. Accessing the constants directly from the module is currently still possible but is deprecated.

Allowed flush values.

- `zlib.constants.Z_NO_FLUSH`
- `zlib.constants.Z_PARTIAL_FLUSH`
- `zlib.constants.Z_SYNC_FLUSH`
- `zlib.constants.Z_FULL_FLUSH`
- `zlib.constants.Z_FINISH`
- `zlib.constants.Z_BLOCK`
- `zlib.constants.Z_TREES`

Return codes for the compression/decompression functions. Negative values are errors, positive values are used for special but normal events.

- `zlib.constants.Z_OK`
- `zlib.constants.Z_STREAM_END`
- `zlib.constants.Z_NEED_DICT`
- `zlib.constants.Z_ERRNO`
- `zlib.constants.Z_STREAM_ERROR`
- `zlib.constants.Z_DATA_ERROR`
- `zlib.constants.Z_MEM_ERROR`
- `zlib.constants.Z_BUF_ERROR`
- `zlib.constants.Z_VERSION_ERROR`

Compression levels.

- `zlib.constants.Z_NO_COMPRESSION`
- `zlib.constants.Z_BEST_SPEED`
- `zlib.constants.Z_BEST_COMPRESSION`
- `zlib.constants.Z_DEFAULT_COMPRESSION`

Compression strategy.

- `zlib.constants.Z_FILTERED`
- `zlib.constants.Z_HUFFMAN_ONLY`
- `zlib.constants.Z_RLE`
- `zlib.constants.Z_FIXED`
- `zlib.constants.Z_DEFAULT_STRATEGY`

Brotli constants

There are several options and other constants available for Brotli-based streams:

Flush operations

The following values are valid flush operations for Brotli-based streams:

- `zlib.constants.BROTLI_OPERATION_PROCESS` (default for all operations)
- `zlib.constants.BROTLI_OPERATION_FLUSH` (default when calling `.flush()`)
- `zlib.constants.BROTLI_OPERATION_FINISH` (default for the last chunk)
- `zlib.constants.BROTLI_OPERATION_EMIT_METADATA`
 - This particular operation may be hard to use in a Node.js context, as the streaming layer makes it hard to know which data will end up in this frame. Also, there is currently no way to consume this data through the Node.js API.

Compressor options

There are several options that can be set on Brotli encoders, affecting compression efficiency and speed. Both the keys and the values can be accessed as properties of the `zlib.constants` object.

The most important options are:

- `BROTLI_PARAM_MODE`
 - `BROTLI_MODE_GENERIC` (default)
 - `BROTLI_MODE_TEXT`, adjusted for UTF-8 text
 - `BROTLI_MODE_FONT`, adjusted for WOFF 2.0 fonts
- `BROTLI_PARAM_QUALITY`
 - Ranges from `BROTLI_MIN_QUALITY` to `BROTLI_MAX_QUALITY`, with a default of `BROTLI_DEFAULT_QUALITY`.
- `BROTLI_PARAM_SIZE_HINT`
 - Integer value representing the expected input size; defaults to `0` for an unknown input size.

The following flags can be set for advanced control over the compression algorithm and memory usage tuning:

- `BROTLI_PARAM_LGWIN`
 - Ranges from `BROTLI_MIN_WINDOW_BITS` to `BROTLI_MAX_WINDOW_BITS`, with a default of `BROTLI_DEFAULT_WINDOW`, or up to `BROTLI_LARGE_MAX_WINDOW_BITS` if the `BROTLI_PARAM_LARGE_WINDOW` flag is set.
- `BROTLI_PARAM_LGBLOCK`
 - Ranges from `BROTLI_MIN_INPUT_BLOCK_BITS` to `BROTLI_MAX_INPUT_BLOCK_BITS`.
- `BROTLI_PARAM_DISABLE_LITERAL_CONTEXT_MODELING`
 - Boolean flag that decreases compression ratio in favour of decompression speed.
- `BROTLI_PARAM_LARGE_WINDOW`
 - Boolean flag enabling “Large Window Brotli” mode (not compatible with the Brotli format as standardized in [RFC 7932](#)).
- `BROTLI_PARAM_NPOSTFIX`
 - Ranges from `0` to `BROTLI_MAX_NPOSTFIX`.

- `BROTLI_PARAM_NDIRECT`
 - Ranges from 0 to 15 << `NPOSTFIX` in steps of 1 << `NPOSTFIX`.

Decompressor options

These advanced options are available for controlling decompression:

- `BROTLI_DECODER_PARAM_DISABLE_RING_BUFFER_REALLOCATION`
 - Boolean flag that affects internal memory allocation patterns.
- `BROTLI_DECODER_PARAM_LARGE_WINDOW`
 - Boolean flag enabling “Large Window Brotli” mode (not compatible with the Brotli format as standardized in [RFC 7932](#)).

Class: Options

Each zlib-based class takes an `options` object. No options are required.

Some options are only relevant when compressing and are ignored by the decompression classes.

- `flush` `<integer>` **Default:** `zlib.constants.Z_NO_FLUSH`
- `finishFlush` `<integer>` **Default:** `zlib.constants.Z_FINISH`
- `chunkSize` `<integer>` **Default:** `16 * 1024`
- `windowBits` `<integer>`
- `level` `<integer>` (compression only)
- `memLevel` `<integer>` (compression only)
- `strategy` `<integer>` (compression only)
- `dictionary` `<Buffer>` | `<TypedArray>` | `<DataView>` | `<ArrayBuffer>` (deflate/inflate only, empty dictionary by default)
- `info` `<boolean>` (If `true`, returns an object with `buffer` and `engine`.)
- `maxOutputLength` `<integer>` Limits output size when using [convenience methods](#). **Default:** `buffer.kMaxLength`

See the [deflateInit2](#) and [inflateInit2](#) documentation for more information.

Class: BrotliOptions

Each Brotli-based class takes an `options` object. All options are optional.

- `flush` `<integer>` **Default:** `zlib.constants.BROTLI_OPERATION_PROCESS`
- `finishFlush` `<integer>` **Default:** `zlib.constants.BROTLI_OPERATION_FINISH`
- `chunkSize` `<integer>` **Default:** `16 * 1024`
- `params` `<Object>` Key-value object containing indexed [Brotli parameters](#).
- `maxOutputLength` `<integer>` Limits output size when using [convenience methods](#). **Default:** `buffer.kMaxLength`

For example:

```
const stream = zlib.createBrotliCompress({
  chunkSize: 32 * 1024,
  params: {
    [zlib.constants.BROTLI_PARAM_MODE]: zlib.constants.BROTLI_MODE_TEXT,
    [zlib.constants.BROTLI_PARAM_QUALITY]: 4,
    [zlib.constants.BROTLI_PARAM_SIZE_HINT]: fs.statSync(inputFile).size
  }
});
```



```
}  
});
```

Class: `zlib.BrotliCompress`

Compress data using the Brotli algorithm.

Class: `zlib.BrotliDecompress`

Decompress data using the Brotli algorithm.

Class: `zlib.Deflate`

Compress data using deflate.

Class: `zlib.DeflateRaw`

Compress data using deflate, and do not append a `zLib` header.

Class: `zlib.Gunzip`

Decompress a gzip stream.

Class: `zlib.Gzip`

Compress data using gzip.

Class: `zlib.Inflate`

Decompress a deflate stream.

Class: `zlib.InflateRaw`

Decompress a raw deflate stream.

Class: `zlib.Unzip`

Decompress either a Gzip- or Deflate-compressed stream by auto-detecting the header.

Class: `zlib.ZlibBase`

Not exported by the `zlib` module. It is documented here because it is the base class of the compressor/decompressor classes.

This class inherits from `stream.Transform`, allowing `zlib` objects to be used in pipes and similar stream operations.

`zlib.bytesRead`

Stability: 0 - Deprecated: Use `zlib.bytesWritten` instead.

- `<number>`

Deprecated alias for `zlib.bytesWritten`. This original name was chosen because it also made sense to interpret the value as the number of bytes read by the engine, but is inconsistent with other streams in Node.js that expose values under these names.

`zlib.bytesWritten`

- `<number>`

The `zlib.bytesWritten` property specifies the number of bytes written to the engine, before the bytes are processed (compressed or decompressed, as appropriate for the derived class).

`zlib.close([callback])`

- `callback` `<Function>`

Close the underlying handle.

`zlib.flush([kind,]callback)`

- `kind` **Default:** `zlib.constants.Z_FULL_FLUSH` for zlib-based streams, `zlib.constants.BROTLI_OPERATION_FLUSH` for Brotli-based streams.
- `callback` `<Function>`

Flush pending data. Don't call this frivolously, premature flushes negatively impact the effectiveness of the compression algorithm.

Calling this only flushes data from the internal `zlib` state, and does not perform flushing of any kind on the streams level. Rather, it behaves like a normal call to `.write()`, i.e. it will be queued up behind other pending writes and will only produce output when data is being read from the stream.

`zlib.params(level, strategy, callback)`

- `level` `<integer>`
- `strategy` `<integer>`
- `callback` `<Function>`

This function is only available for zlib-based streams, i.e. not Brotli.

Dynamically update the compression level and compression strategy. Only applicable to deflate algorithm.

`zlib.reset()`

Reset the compressor/decompressor to factory defaults. Only applicable to the inflate and deflate algorithms.

`zlib.constants`

Provides an object enumerating Zlib-related constants.

`zlib.createBrotliCompress([options])`

- `options` `<brotli options>`

Creates and returns a new `BrotliCompress` object.

zlib.createBrotliDecompress([options])

- `options` `<brotli options>`

Creates and returns a new `BrotliDecompress` object.

zlib.createDeflate([options])

- `options` `<zlib options>`

Creates and returns a new `Deflate` object.

zlib.createDeflateRaw([options])

- `options` `<zlib options>`

Creates and returns a new `DeflateRaw` object.

An upgrade of zlib from 1.2.8 to 1.2.11 changed behavior when `windowBits` is set to 8 for raw deflate streams. zlib would automatically set `windowBits` to 9 if was initially set to 8. Newer versions of zlib will throw an exception, so Node.js restored the original behavior of upgrading a value of 8 to 9, since passing `windowBits = 9` to zlib actually results in a compressed stream that effectively uses an 8-bit window only.

zlib.createGunzip([options])

- `options` `<zlib options>`

Creates and returns a new `Gunzip` object.

zlib.createGzip([options])

- `options` `<zlib options>`

Creates and returns a new `Gzip` object. See [example](#) .

zlib.createInflate([options])

- `options` `<zlib options>`

Creates and returns a new `Inflate` object.

zlib.createInflateRaw([options])

- `options` `<zlib options>`

Creates and returns a new `InflateRaw` object.

zlib.createUnzip([options])

- `options` `<zlib options>`

Creates and returns a new `Unzip` object.

Convenience methods

All of these take a `Buffer` , `TypedArray` , `DataView` , `ArrayBuffer` or string as the first argument, an optional second argument to supply options to the `zlib` classes and will call the supplied callback with `callback(error, result)` .

Every method has a `*Sync` counterpart, which accept the same arguments, but without a callback.

`zlib.brotliCompress(buffer[, options], callback)`

- `buffer` `<Buffer>` | `<TypedArray>` | `<DataView>` | `<ArrayBuffer>` | `<string>`
- `options` `<brotli options>`
- `callback` `<Function>`

`zlib.brotliCompressSync(buffer[, options])`

- `buffer` `<Buffer>` | `<TypedArray>` | `<DataView>` | `<ArrayBuffer>` | `<string>`
- `options` `<brotli options>`

Compress a chunk of data with `BrotliCompress` .

`zlib.brotliDecompress(buffer[, options], callback)`

- `buffer` `<Buffer>` | `<TypedArray>` | `<DataView>` | `<ArrayBuffer>` | `<string>`
- `options` `<brotli options>`
- `callback` `<Function>`

`zlib.brotliDecompressSync(buffer[, options])`

- `buffer` `<Buffer>` | `<TypedArray>` | `<DataView>` | `<ArrayBuffer>` | `<string>`
- `options` `<brotli options>`

Decompress a chunk of data with `BrotliDecompress` .

`zlib.deflate(buffer[, options], callback)`

- `buffer` `<Buffer>` | `<TypedArray>` | `<DataView>` | `<ArrayBuffer>` | `<string>`
- `options` `<zlib options>`
- `callback` `<Function>`

`zlib.deflateSync(buffer[, options])`

- `buffer` `<Buffer>` | `<TypedArray>` | `<DataView>` | `<ArrayBuffer>` | `<string>`
- `options` `<zlib options>`

Compress a chunk of data with `Deflate` .

`zlib.deflateRaw(buffer[, options], callback)`

- `buffer` `<Buffer>` | `<TypedArray>` | `<DataView>` | `<ArrayBuffer>` | `<string>`
- `options` `<zlib options>`
- `callback` `<Function>`

`zlib.deflateRawSync(buffer[, options])`

- `buffer` `<Buffer>` | `<TypedArray>` | `<DataView>` | `<ArrayBuffer>` | `<string>`
- `options` `<zlib options>`

Compress a chunk of data with `DeflateRaw`.

`zlib.gunzip(buffer[, options], callback)`

- `buffer` `<Buffer>` | `<TypedArray>` | `<DataView>` | `<ArrayBuffer>` | `<string>`
- `options` `<zlib options>`
- `callback` `<Function>`

`zlib.gunzipSync(buffer[, options])`

- `buffer` `<Buffer>` | `<TypedArray>` | `<DataView>` | `<ArrayBuffer>` | `<string>`
- `options` `<zlib options>`

Decompress a chunk of data with `Gunzip`.

`zlib.gzip(buffer[, options], callback)`

- `buffer` `<Buffer>` | `<TypedArray>` | `<DataView>` | `<ArrayBuffer>` | `<string>`
- `options` `<zlib options>`
- `callback` `<Function>`

`zlib.gzipSync(buffer[, options])`

- `buffer` `<Buffer>` | `<TypedArray>` | `<DataView>` | `<ArrayBuffer>` | `<string>`
- `options` `<zlib options>`

Compress a chunk of data with `Gzip`.

`zlib.inflate(buffer[, options], callback)`

- `buffer` `<Buffer>` | `<TypedArray>` | `<DataView>` | `<ArrayBuffer>` | `<string>`
- `options` `<zlib options>`
- `callback` `<Function>`

`zlib.inflateSync(buffer[, options])`

- `buffer` `<Buffer>` | `<TypedArray>` | `<DataView>` | `<ArrayBuffer>` | `<string>`
- `options` `<zlib options>`

Decompress a chunk of data with `Inflate`.

`zlib.inflateRaw(buffer[, options], callback)`

- `buffer` `<Buffer>` | `<TypedArray>` | `<DataView>` | `<ArrayBuffer>` | `<string>`
- `options` `<zlib options>`
- `callback` `<Function>`

`zlib.inflateRawSync(buffer[, options])`

- `buffer` `<Buffer>` | `<TypedArray>` | `<DataView>` | `<ArrayBuffer>` | `<string>`
- `options` `<zlib options>`

Decompress a chunk of data with `InflateRaw` .

zlib.unzip(buffer[, options], callback)

- `buffer` `<Buffer> | <TypedArray> | <DataView> | <ArrayBuffer> | <string>`
- `options` `<zlib options>`
- `callback` `<Function>`

zlib.unzipSync(buffer[, options])

- `buffer` `<Buffer> | <TypedArray> | <DataView> | <ArrayBuffer> | <string>`
- `options` `<zlib options>`

Decompress a chunk of data with `Unzip` .