Stream

> *Stability: 2 - Stable*

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.closed`

- {boolean}

Is `true` after `'close'` has been emitted.

`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, defer calls to `writable.uncork()` 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.writableAborted`

> *Stability: 1 - Experimental*

- {boolean}

Returns whether the stream was destroyed or errored before emitting `'finish'`.

`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.errored`

- {Error}

Returns error if the stream has been destroyed with an error.

`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. Once

`write()` returns false, do not write more chunks 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.

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 or when the end of the stream has been reached. Effectively, the `'readable'` event indicates that the stream has new information. If data is available, `stream.read()` will return that data.

```
const readable = getReadableStreamSomehow();
readable.on('readable', function() {
  // There is some data to read now.
  let data;

  while ((data = this.read()) !== null) {
    console.log(data);
  }
});
```

If the end of the stream has been reached, calling `stream.read()` will return `null` and trigger the `'end'` event. This is also true if there never was any data to be read. 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 some cases, attaching a listener for the `'readable'` event will cause some amount of data to be read into an internal buffer.

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.closed`

- {boolean}

Is `true` after `'close'` has been emitted.

`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 reads data out of the internal buffer and returns it. If no data is available to be read, `null` is returned. By default, the data is 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.readableAborted`

> *Stability: 1 - Experimental*

- {boolean}

Returns whether the stream was destroyed or errored before emitting `'end'`.

`readable.readableDidRead`

> *Stability: 1 - Experimental*

- {boolean}

Returns whether `'data'` 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.errored`

- {Error}

Returns error if the stream has been destroyed with an error.

`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.includes('\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);
        return;
```

```
      }
      // 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 then 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`.
- 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();
```

`readable.map(fn[, options])`

> *Stability: 1 - Experimental*

- `fn` {Function|AsyncFunction} a function to map over every chunk in the stream.
  - `data` {any} a chunk of data from the stream.
  - `options` {Object}
    - `signal` {AbortSignal} aborted if the stream is destroyed allowing to abort the `fn` call early.
- `options` {Object}
  - `concurrency` {number} the maximum concurrent invocation of `fn` to call on the stream at once. **Default:** `1`.
  - `signal` {AbortSignal} allows destroying the stream if the signal is aborted.
- Returns: {Readable} a stream mapped with the function `fn`.

This method allows mapping over the stream. The `fn` function will be called for every chunk in the stream. If the `fn` function returns a promise - that promise will be `await`ed before being passed to the result stream.

```
import { Readable } from 'stream';
import { Resolver } from 'dns/promises';

// With a synchronous mapper.
for await (const chunk of Readable.from([1, 2, 3, 4]).map((x) => x * 2)) {
  console.log(chunk); // 2, 4, 6, 8
}
// With an asynchronous mapper, making at most 2 queries at a time.
const resolver = new Resolver();
const dnsResults = Readable.from([
  'nodejs.org',
  'openjsf.org',
  'www.linuxfoundation.org',
]).map((domain) => resolver.resolve4(domain), { concurrency: 2 });
for await (const result of dnsResults) {
  console.log(result); // Logs the DNS result of resolver.resolve4.
}
```

`readable.filter(fn[, options])`

> *Stability: 1 - Experimental*

- `fn` {Function|AsyncFunction} a function to filter chunks from the stream.
  - `data` {any} a chunk of data from the stream.
  - `options` {Object}
    - `signal` {AbortSignal} aborted if the stream is destroyed allowing to abort the `fn` call early.
- `options` {Object}
  - `concurrency` {number} the maximum concurrent invocation of `fn` to call on the stream at once. **Default:** `1`.
  - `signal` {AbortSignal} allows destroying the stream if the signal is aborted.

- Returns: {Readable} a stream filtered with the predicate `fn`.

This method allows filtering the stream. For each chunk in the stream the `fn` function will be called and if it returns a truthy value, the chunk will be passed to the result stream. If the `fn` function returns a promise - that promise will be `await`ed.

```
import { Readable } from 'stream';
import { Resolver } from 'dns/promises';

// With a synchronous predicate.
for await (const chunk of Readable.from([1, 2, 3, 4]).filter((x) => x > 2)) {
  console.log(chunk); // 3, 4
}
// With an asynchronous predicate, making at most 2 queries at a time.
const resolver = new Resolver();
const dnsResults = Readable.from([
  'nodejs.org',
  'openjsf.org',
  'www.linuxfoundation.org',
]).filter(async (domain) => {
  const { address } = await resolver.resolve4(domain, { ttl: true });
  return address.ttl > 60;
}, { concurrency: 2 });
for await (const result of dnsResults) {
  // Logs domains with more than 60 seconds on the resolved dns record.
  console.log(result);
}
```

`readable.forEach(fn[, options])`

> *Stability: 1 - Experimental*

- `fn` {Function|AsyncFunction} a function to call on each chunk of the stream.
  - `data` {any} a chunk of data from the stream.
  - `options` {Object}
    - `signal` {AbortSignal} aborted if the stream is destroyed allowing to abort the `fn` call early.
- `options` {Object}
  - `concurrency` {number} the maximum concurrent invocation of `fn` to call on the stream at once. **Default:** `1`.
  - `signal` {AbortSignal} allows destroying the stream if the signal is aborted.
- Returns: {Promise} a promise for when the stream has finished.

This method allows iterating a stream. For each chunk in the stream the `fn` function will be called. If the `fn` function returns a promise - that promise will be `await`ed.

This method is different from `for await...of` loops in that it can optionally process chunks concurrently. In addition, a `forEach` iteration can only be stopped by having passed a `signal` option and aborting the related `AbortController` while `for await...of` can be stopped with `break` or `return`. In either case the stream will be destroyed.

This method is different from listening to the `'data'` event in that it uses the `readable` event in the underlying machinary and can limit the number of concurrent `fn` calls.

```
import { Readable } from 'stream';
import { Resolver } from 'dns/promises';

// With a synchronous predicate.
for await (const chunk of Readable.from([1, 2, 3, 4]).filter((x) => x > 2)) {
  console.log(chunk); // 3, 4
}
// With an asynchronous predicate, making at most 2 queries at a time.
const resolver = new Resolver();
const dnsResults = Readable.from([
  'nodejs.org',
  'openjsf.org',
  'www.linuxfoundation.org',
]).map(async (domain) => {
  const { address } = await resolver.resolve4(domain, { ttl: true });
  return address;
}, { concurrency: 2 });
await dnsResults.forEach((result) => {
  // Logs result, similar to `for await (const result of dnsResults)`
  console.log(result);
});
console.log('done'); // Stream has finished
```

`readable.toArray([options])`

> *Stability: 1 - Experimental*

- `options` {Object}
  - `signal` {AbortSignal} allows cancelling the toArray operation if the signal is aborted.
- Returns: {Promise} a promise containing an array with the contents of the stream.

This method allows easily obtaining the contents of a stream.

As this method reads the entire stream into memory, it negates the benefits of streams. It's intended for interoperability and convenience, not as the primary way to consume streams.

```
import { Readable } from 'stream';
import { Resolver } from 'dns/promises';

await Readable.from([1, 2, 3, 4]).toArray(); // [1, 2, 3, 4]

// Make dns queries concurrently using .map and collect
// the results into an array using toArray
const dnsResults = await Readable.from([
  'nodejs.org',
  'openjsf.org',
  'www.linuxfoundation.org',
]).map(async (domain) => {
  const { address } = await resolver.resolve4(domain, { ttl: true });
```

```
  return address;
}, { concurrency: 2 }).toArray();
```

`readable.some(fn[, options])`

> *Stability: 1 - Experimental*

- `fn` {Function|AsyncFunction} a function to call on each chunk of the stream.
  - `data` {any} a chunk of data from the stream.
  - `options` {Object}
    - `signal` {AbortSignal} aborted if the stream is destroyed allowing to abort the `fn` call early.
- `options` {Object}
  - `concurrency` {number} the maximum concurrent invocation of `fn` to call on the stream at once. **Default:** `1`.
  - `signal` {AbortSignal} allows destroying the stream if the signal is aborted.
- Returns: {Promise} a promise evaluating to `true` if `fn` returned a truthy value for at least one of the chunks.

This method is similar to `Array.prototype.some` and calls `fn` on each chunk in the stream until the awaited return value is `true` (or any truthy value). Once an `fn` call on a chunk awaited return value is truthy, the stream is destroyed and the promise is fulfilled with `true`. If none of the `fn` calls on the chunks return a truthy value, the promise is fulfilled with `false`.

```
import { Readable } from 'stream';
import { stat } from 'fs/promises';

// With a synchronous predicate.
await Readable.from([1, 2, 3, 4]).some((x) => x > 2); // true
await Readable.from([1, 2, 3, 4]).some((x) => x < 0); // false

// With an asynchronous predicate, making at most 2 file checks at a time.
const anyBigFile = await Readable.from([
  'file1',
  'file2',
  'file3',
]).some(async (fileName) => {
  const stats = await stat(fileName);
  return stat.size > 1024 * 1024;
}, { concurrency: 2 });
console.log(anyBigFile); // `true` if any file in the list is bigger than 1MB
console.log('done'); // Stream has finished
```

`readable.find(fn[, options])`

> *Stability: 1 - Experimental*

- `fn` {Function|AsyncFunction} a function to call on each chunk of the stream.
  - `data` {any} a chunk of data from the stream.
  - `options` {Object}

        - `signal` {AbortSignal} aborted if the stream is destroyed allowing to abort the `fn` call early.

- `options` {Object}
  - `concurrency` {number} the maximum concurrent invocation of `fn` to call on the stream at once. **Default:** `1`.
  - `signal` {AbortSignal} allows destroying the stream if the signal is aborted.
- Returns: {Promise} a promise evaluating to the first chunk for which `fn` evaluated with a truthy value, or `undefined` if no element was found.

This method is similar to `Array.prototype.find` and calls `fn` on each chunk in the stream to find a chunk with a truthy value for `fn`. Once an `fn` call's awaited return value is truthy, the stream is destroyed and the promise is fulfilled with value for which `fn` returned a truthy value. If all of the `fn` calls on the chunks return a falsy value, the promise is fulfilled with `undefined`.

```
import { Readable } from 'stream';
import { stat } from 'fs/promises';

// With a synchronous predicate.
await Readable.from([1, 2, 3, 4]).find((x) => x > 2); // 3
await Readable.from([1, 2, 3, 4]).find((x) => x > 0); // 1
await Readable.from([1, 2, 3, 4]).find((x) => x > 10); // undefined

// With an asynchronous predicate, making at most 2 file checks at a time.
const foundBigFile = await Readable.from([
  'file1',
  'file2',
  'file3',
]).find(async (fileName) => {
  const stats = await stat(fileName);
  return stat.size > 1024 * 1024;
}, { concurrency: 2 });
console.log(foundBigFile); // File name of large file, if any file in the list is
bigger than 1MB
console.log('done'); // Stream has finished
```

`readable.every(fn[, options])`

> *Stability: 1 - Experimental*

- `fn` {Function|AsyncFunction} a function to call on each chunk of the stream.
  - `data` {any} a chunk of data from the stream.
  - `options` {Object}
    - `signal` {AbortSignal} aborted if the stream is destroyed allowing to abort the `fn` call early.
- `options` {Object}
  - `concurrency` {number} the maximum concurrent invocation of `fn` to call on the stream at once. **Default:** `1`.
  - `signal` {AbortSignal} allows destroying the stream if the signal is aborted.

- Returns: {Promise} a promise evaluating to `true` if `fn` returned a truthy value for all of the chunks.

This method is similar to `Array.prototype.every` and calls `fn` on each chunk in the stream to check if all awaited return values are truthy value for `fn`. Once an `fn` call on a chunk awaited return value is falsy, the stream is destroyed and the promise is fulfilled with `false`. If all of the `fn` calls on the chunks return a truthy value, the promise is fulfilled with `true`.

```
import { Readable } from 'stream';
import { stat } from 'fs/promises';

// With a synchronous predicate.
await Readable.from([1, 2, 3, 4]).every((x) => x > 2); // false
await Readable.from([1, 2, 3, 4]).every((x) => x > 0); // true

// With an asynchronous predicate, making at most 2 file checks at a time.
const allBigFiles = await Readable.from([
  'file1',
  'file2',
  'file3',
]).every(async (fileName) => {
  const stats = await stat(fileName);
  return stat.size > 1024 * 1024;
}, { concurrency: 2 });
// `true` if all files in the list are bigger than 1MiB
console.log(allBigFiles);
console.log('done'); // Stream has finished
```

`readable.flatMap(fn[, options])`

Stability: 1 - Experimental

- `fn` {Function|AsyncGeneratorFunction|AsyncFunction} a function to map over every chunk in the stream.
  - `data` {any} a chunk of data from the stream.
  - `options` {Object}
    - `signal` {AbortSignal} aborted if the stream is destroyed allowing to abort the `fn` call early.
- `options` {Object}
  - `concurrency` {number} the maximum concurrent invocation of `fn` to call on the stream at once. **Default:** `1`.
  - `signal` {AbortSignal} allows destroying the stream if the signal is aborted.
- Returns: {Readable} a stream flat-mapped with the function `fn`.

This method returns a new stream by applying the given callback to each chunk of the stream and then flattening the result.

It is possible to return a stream or another iterable or async iterable from `fn` and the result streams will be merged (flattened) into the returned stream.

```
import { Readable } from 'stream';
import { createReadStream } from 'fs';
```

```
// With a synchronous mapper.
for await (const chunk of Readable.from([1, 2, 3, 4]).flatMap((x) => [x, x])) {
  console.log(chunk); // 1, 1, 2, 2, 3, 3, 4, 4
}
// With an asynchronous mapper, combine the contents of 4 files
const concatResult = Readable.from([
  './1.mjs',
  './2.mjs',
  './3.mjs',
  './4.mjs',
]).flatMap((fileName) => createReadStream(fileName));
for await (const result of concatResult) {
  // This will contain the contents (all chunks) of all 4 files
  console.log(result);
}
```

`readable.drop(limit[, options])`

> *Stability: 1 - Experimental*

- `limit` {number} the number of chunks to drop from the readable.
- `options` {Object}
  - `signal` {AbortSignal} allows destroying the stream if the signal is aborted.
- Returns: {Readable} a stream with `limit` chunks dropped.

This method returns a new stream with the first `limit` chunks dropped.

```
import { Readable } from 'stream';

await Readable.from([1, 2, 3, 4]).drop(2).toArray(); // [3, 4]
```

`readable.take(limit[, options])`

> *Stability: 1 - Experimental*

- `limit` {number} the number of chunks to take from the readable.
- `options` {Object}
  - `signal` {AbortSignal} allows destroying the stream if the signal is aborted.
- Returns: {Readable} a stream with `limit` chunks taken.

This method returns a new stream with the first `limit` chunks.

```
import { Readable } from 'stream';

await Readable.from([1, 2, 3, 4]).take(2).toArray(); // [1, 2]
```

`readable.asIndexedPairs([options])`

> *Stability: 1 - Experimental*

- `options` {Object}

    - `signal` {AbortSignal} allows destroying the stream if the signal is aborted.
- Returns: {Readable} a stream of indexed pairs.

This method returns a new stream with chunks of the underlying stream paired with a counter in the form `[index, chunk]`. The first index value is 0 and it increases by 1 for each chunk produced.

```
import { Readable } from 'stream';

const pairs = await Readable.from(['a', 'b', 'c']).asIndexedPairs().toArray();
console.log(pairs); // [[0, 'a'], [1, 'b'], [2, 'c']]
```

`readable.reduce(fn[, initial[, options]])`

> *Stability: 1 - Experimental*

- `fn` {Function|AsyncFunction} a reducer function to call over every chunk in the stream.
    - `previous` {any} the value obtained from the last call to `fn` or the `initial` value if specified or the first chunk of the stream otherwise.
    - `data` {any} a chunk of data from the stream.
    - `options` {Object}
        - `signal` {AbortSignal} aborted if the stream is destroyed allowing to abort the `fn` call early.
- `initial` {any} the initial value to use in the reduction.
- `options` {Object}
    - `signal` {AbortSignal} allows destroying the stream if the signal is aborted.
- Returns: {Promise} a promise for the final value of the reduction.

This method calls `fn` on each chunk of the stream in order, passing it the result from the calculation on the previous element. It returns a promise for the final value of the reduction.

The reducer function iterates the stream element-by-element which means that there is no `concurrency` parameter or parallism. To perform a `reduce` concurrently, it can be chained to the `readable.map` method.

If no `initial` value is supplied the first chunk of the stream is used as the initial value. If the stream is empty, the promise is rejected with a `TypeError` with the `ERR_INVALID_ARGS` code property.

```
import { Readable } from 'stream';

const ten = await Readable.from([1, 2, 3, 4]).reduce((previous, data) => {
  return previous + data;
});
console.log(ten); // 10
```

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

`duplex.allowHalfOpen`

- {boolean}

If `false` then the stream will automatically end the writable side when the readable side ends. Set initially by the `allowHalfOpen` constructor option, which defaults to `false`.

This can be changed manually to change the half-open behavior of an existing `Duplex` stream instance, but must be changed before the `'end'` event is emitted.

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.');
  }
});

rs.resume(); // Drain the stream.
```

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 signal = ac.signal;

  setTimeout(() => ac.abort(), 1);
  await pipeline(
    fs.createReadStream('archive.tar'),
    zlib.createGzip(),
    fs.createWriteStream('archive.tar.gz'),
    { signal },
  );
}

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, { signal }) {
      source.setEncoding('utf8');  // Work with strings rather than `Buffer`s.
      for await (const chunk of source) {
        yield await processChunk(chunk, { signal });
      }
    },
    fs.createWriteStream('uppercase.txt')
  );
  console.log('Pipeline succeeded.');
}

run().catch(console.error);
```

Remember to handle the `signal` argument passed into the async generator. Especially in the case where the async generator is the source for the pipeline (i.e. first argument) or the pipeline will never complete.

```
const { pipeline } = require('stream/promises');
const fs = require('fs');

async function run() {
```

```
  await pipeline(
    async function* ({ signal }) {
      await someLongRunningfn({ signal });
      yield 'asd';
    },
    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. If the last stream is readable, dangling event listeners will be removed so that the last stream can be consumed later.

`stream.pipeline()` closes all the streams when an error is raised. The `IncomingRequest` usage with `pipeline` could lead to an unexpected behavior once it would destroy the socket without sending the expected response. See the example below:

```
const fs = require('fs');
const http = require('http');
const { pipeline } = require('stream');

const server = http.createServer((req, res) => {
  const fileStream = fs.createReadStream('./fileNotExist.txt');
  pipeline(fileStream, res, (err) => {
    if (err) {
      console.log(err); // No such file
      // this message can't be sent once `pipeline` already destroyed the socket
      return res.end('error!!!');
    }
  });
});
```

`stream.compose(...streams)`

> *Stability: 1 -* `stream.compose` *is experimental.*

- `streams` {Stream[]|Iterable[]|AsyncIterable[]|Function[]}
- Returns: {stream.Duplex}

Combines two or more streams into a `Duplex` stream that writes to the first stream and reads from the last. Each provided stream is piped into the next, using `stream.pipeline`. If any of the streams error then all are destroyed, including the outer `Duplex` stream.

Because `stream.compose` returns a new stream that in turn can (and should) be piped into other streams, it enables composition. In contrast, when passing streams to `stream.pipeline`, typically the first stream is a readable stream and the last a writable stream, forming a closed circuit.

If passed a `Function` it must be a factory method taking a `source` `Iterable`.

```
import { compose, Transform } from 'stream';

const removeSpaces = new Transform({
  transform(chunk, encoding, callback) {
    callback(null, String(chunk).replace(' ', ''));
  }
});

async function* toUpper(source) {
  for await (const chunk of source) {
    yield String(chunk).toUpperCase();
  }
}

let res = '';
for await (const buf of compose(removeSpaces, toUpper).end('hello world')) {
  res += buf;
}

console.log(res); // prints 'HELLOWORLD'
```

`stream.compose` can be used to convert async iterables, generators and functions into streams.

- `AsyncIterable` converts into a readable `Duplex`. Cannot yield `null`.
- `AsyncGeneratorFunction` converts into a readable/writable transform `Duplex`. Must take a source `AsyncIterable` as first parameter. Cannot yield `null`.
- `AsyncFunction` converts into a writable `Duplex`. Must return either `null` or `undefined`.

```
import { compose } from 'stream';
import { finished } from 'stream/promises';

// Convert AsyncIterable into readable Duplex.
const s1 = compose(async function*() {
  yield 'Hello';
  yield 'World';
}());

// Convert AsyncGenerator into transform Duplex.
const s2 = compose(async function*(source) {
  for await (const chunk of source) {
    yield String(chunk).toUpperCase();
  }
});
```

```
let res = '';

// Convert AsyncFunction into writable Duplex.
const s3 = compose(async function(source) {
  for await (const chunk of source) {
    res += chunk;
  }
});

await finished(compose(s1, s2, s3));

console.log(res); // prints 'HELLOWORLD'
```

`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.Readable.fromWeb(readableStream[, options])`

> *Stability: 1 - Experimental*

- `readableStream` {ReadableStream}
- `options` {Object}
  - `encoding` {string}
  - `highWaterMark` {number}
  - `objectMode` {boolean}
  - `signal` {AbortSignal}

- Returns: {stream.Readable}

`stream.Readable.isDisturbed(stream)`

> *Stability: 1 - Experimental*

- `stream` {stream.Readable|ReadableStream}
- Returns: `boolean`

Returns whether the stream has been read from or cancelled.

`stream.isErrored(stream)`

> *Stability: 1 - Experimental*

- `stream` {Readable|Writable|Duplex|WritableStream|ReadableStream}
- Returns: {boolean}

Returns whether the stream has encountered an error.

`stream.isReadable(stream)`

> *Stability: 1 - Experimental*

- `stream` {Readable|Duplex|ReadableStream}
- Returns: {boolean}

Returns whether the stream is readable.

`stream.Readable.toWeb(streamReadable)`

> *Stability: 1 - Experimental*

- `streamReadable` {stream.Readable}
- Returns: {ReadableStream}

`stream.Writable.fromWeb(writableStream[, options])`

> *Stability: 1 - Experimental*

- `writableStream` {WritableStream}
- `options` {Object}
  - `decodeStrings` {boolean}
  - `highWaterMark` {number}
  - `objectMode` {boolean}
  - `signal` {AbortSignal}
- Returns: {stream.Writable}

`stream.Writable.toWeb(streamWritable)`

> *Stability: 1 - Experimental*

- `streamWritable` {stream.Writable}
- Returns: {WritableStream}

`stream.Duplex.from(src)`

- `src` {Stream|Blob|ArrayBuffer|string|Iterable|AsyncIterable|AsyncGeneratorFunction|AsyncFunction|Promise|Object}

A utility method for creating duplex streams.

- `Stream` converts writable stream into writable `Duplex` and readable stream to `Duplex`.
- `Blob` converts into readable `Duplex`.
- `string` converts into readable `Duplex`.
- `ArrayBuffer` converts into readable `Duplex`.
- `AsyncIterable` converts into a readable `Duplex`. Cannot yield `null`.
- `AsyncGeneratorFunction` converts into a readable/writable transform `Duplex`. Must take a source `AsyncIterable` as first parameter. Cannot yield `null`.
- `AsyncFunction` converts into a writable `Duplex`. Must return either `null` or `undefined`
- `Object ({ writable, readable })` converts `readable` and `writable` into `Stream` and then combines them into `Duplex` where the `Duplex` will write to the `writable` and read from the `readable`.
- `Promise` converts into readable `Duplex`. Value `null` is ignored.
- Returns: {stream.Duplex}

`stream.Duplex.fromWeb(pair[, options])`

> *Stability: 1 - Experimental*

- `pair` {Object}
  - `readable` {ReadableStream}
  - `writable` {WritableStream}
- `options` {Object}
  - `allowHalfOpen` {boolean}
  - `decodeStrings` {boolean}
  - `encoding` {string}
  - `highWaterMark` {number}
  - `objectMode` {boolean}
  - `signal` {AbortSignal}
- Returns: {stream.Duplex}

`stream.Duplex.toWeb(streamDuplex)`

> *Stability: 1 - Experimental*

- `streamDuplex` {stream.Duplex}
- Returns: {Object}
  - `readable` {ReadableStream}
  - `writable` {WritableStream}

`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 writeable 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	`Readable`	`_read()`
Writing only	`Writable`	`_write()`, `_writev()`, `_final()`
Reading and writing	`Duplex`	`_read()`, `_write()`, `_writev()`, `_final()`
Operate on written data, then read the result	`Transform`	`_transform()`, `_flush()`, `_final()`

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 writeable 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;
    this.fd = null;
  }
  _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._writev()` 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._writev(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._writev()` 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, `_writev()` will be called instead of `_write()`.

The `writable._writev()` 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. Furthermore, the `callback` should not be mixed with async/await once it is executed when a promise is resolved.

`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._writev()` 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 unpiped.

```
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);
        callback();
      } 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');

const ac = new AbortController();
const signal = ac.signal;

async function * generate() {
  yield 'a';
  await someLongRunningFn({ signal });
  yield 'b';
  yield 'c';
}

const readable = Readable.from(generate());
readable.on('close', () => {
  ac.abort();
});

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');

const ac = new AbortController();
const signal = ac.signal;

const iterator = createIterator({ signal });

// Callback Pattern
```

```
pipeline(iterator, writable, (err, value) => {
  if (err) {
    console.error(err);
  } else {
    console.log(value, 'value returned');
  }
}).on('close', () => {
  ac.abort();
});

// Promise Pattern
pipelinePromise(iterator, writable)
  .then((value) => {
    console.log(value, 'value returned');
  })
  .catch((err) => {
    console.error(err);
    ac.abort();
  });
```

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.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.