Shell scripting with Node.js

Dr. Axel Rauschmayer

Shell scripting with Node.js

Dr. Axel Rauschmayer

2022

Cover: "Hex Hexagonal Abstract" by CreativeMagic

exploringjs.com

Contents

Part I

About this book

Chapter 1

About this book

Contents

This chapter helps you decide whether or not this book is of interest to you.

1.1 Why should I read this book?

This book is about shell scripting with Node.js. You will learn:

- How Node.js works:
  - Its foundations: its architecture, its event loop, etc.
  - Its API: How to use its global variables and modules.
- What *npm packages* (the de-facto standard for JavaScript packages) are.
- How to use *npm* (the package manager bundled with Node.js) to:
  - Install and manage packages.
  - Create and publish packages.
- How to write cross-platform *package scripts* for running development tasks such as generating artifacts and running tests.
- How to use all of the aforementioned knowledge to create and deploy cross-platform shell scripts.

1.2 What knowledge is required to read this book?

You should be familiar with JavaScript – especially:

- ECMAScript modules: importing and exporting values, etc.
- Asynchronous JavaScript: Promises, async functions, etc.

My book on JavaScript, "JavaScript for impatient programmers" is free to read online:

- It has a chapter on modules.
- It covers asynchronous JavaScript in a series of chapters, starting with "Asynchronous programming in JavaScript".

1.3 Buying and previewing this book

1.3.1 How can I buy this book?

You can buy a package with ebooks. They come in these formats (all without DRM):

- PDF
- HTML
- EPUB
- MOBI

1.3.2 How can I preview the book?

- The HTML version is free to read online.
- On the homepage of this book, there are extensive previews for all ebook versions of this book.

1.4 About the author

Dr. Axel Rauschmayer specializes in JavaScript and web development. He has been developing web applications since 1995. In 1999, he was technical manager at a German internet startup that later expanded internationally. In 2006, he held his first talk on Ajax. In 2010, he received a PhD in Informatics from the University of Munich.

Since 2009, he has been blogging about web development at 2ality.com and has written several books on JavaScript. He has held trainings and talks for companies such as eBay, Bank of America, and O'Reilly Media.

He lives in Munich, Germany.

1.5 Acknowledgements

- Cover: Hex Hexagonal Abstract by CreativeMagic

Chapter 2

Instructions

Contents

This chapter contains information that is useful when reading this book.

2.1 How to read this book

There are two ways in which you can read this book:

- Like a guide: Start at the beginning and keep reading.
- Like a reference: Only read chapters that interest you, skip the rest.

This book was written with both ways in mind, so skipping content should not be a problem. If, at any point, there is relevant information elsewhere in the book, I point to it.

2.2 How assertions are used in this book

The following import is always assumed to have been made (similarly to how non-strict `assert` is available in the Node.js REPL):

```
import * as assert from 'node:assert/strict';
```

This module implements *assertions* – which are often used in examples in this book. This is what they look like:

```
// Comparing primitive values:
assert.equal(3 + 4, 7);
assert.equal('abc'.toUpperCase(), 'ABC');

// Comparing objects:
```

```
assert.notEqual({prop: 1}, {prop: 1}); // shallow comparison
assert.deepEqual({prop: 1}, {prop: 1}); // deep comparison
assert.notDeepEqual({prop: 1}, {prop: 2}); // deep comparison
```

Part II

Foundations

Chapter 3

Getting started with Node.js

Contents

This chapter explains the first steps with Node.js.

3.1 Getting help for Node.js

- Online:
  - Overview of online documentation
  - API documentation
  - Command line options
- Command line:
  - Print online help: `node -h`
  - Print version of Node.js: `node -v`
  - Print versions of various Node.js components:
    ```
    * npm version
    * node -p process.versions
    ```

3.2 Installing Node.js and npm

The installer for Node.js also installs the package manager npm. It can be downloaded from the Node.js homepage and is available for many operating systems.

3.3 Running Node.js code

3.3.1 Evaluating code in the Node.js REPL

The Node.js REPL (read-eval-print loop) is a command line where we can interactively evaluate Node.js code.

We can start the Node.js REPL in JavaScript strict mode (which is safer and switched on by default for code in ESM modules):

```
node --use_strict
```

If we run `node` without any arguments, the Node.js REPL does not use strict mode:

```
node
```

This is what using the Node.js REPL looks like (`%` is a Unix shell prompt, `>` is the Node.js REPL prompt):

```
% node
Welcome to Node.js v18.9.0.
Type ".help" for more information.
> path.join('dir', 'sub', 'file.txt')
'dir/sub/file.txt'
>
```

All of Node's built-in modules are available via global variables in the REPL: `assert`, `path`, `fs`, `util`, etc.

3.3.2 Quickly printing the result of a JavaScript expression

We can use the shell command `node` with the option `--print` (abbreviation: `-p`) to print the result of evaluating a JavaScript expression. Similarly to the REPL, all built-in modules are available via global variables. For example, the following command prints the path of the homedirectory and works on both Unixes and Windows:

```
node -p "os.homedir()"
```

For more information on this command line option, see [Content not included].

3.3.3 Running modules with Node.js code

Take, for example, the following module:

```
// my-module.mjs
import * as os from 'node:os';
console.log(os.userInfo());
```

We can run it from a shell via:

```
node my-module.mjs
```

3.3.4 Running Node.js code that's in the clipboard

We can also run Node.js code that we have copied to the clipboard. For example, we could copy the code of `my-module.mjs` from the previous section and run it like this on macOS:

```
pbpaste | node --input-type=module
```

Option `--input-type=module` tells Node.js to interpret the code it receives from standard input as a module. Among other things, that enables us to use `import`.

The macOS shell command `pbpaste` sends the contents of the clipboard to standard output. Other operating systems have similar shell commands:

- Windows Command shell: `powershell get-clipboard`
- Windows PowerShell: `get-clipboard`
- Linux: `xclip`

Chapter 4

An overview of Node.js: architecture, APIs, event loop, concurrency

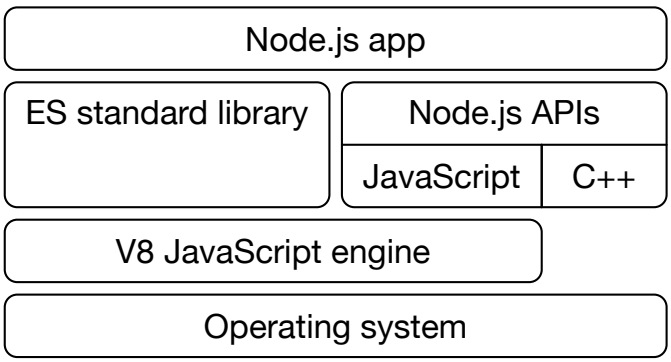
Contents

This chapter gives an overview of how Node.js works:

- What its architecture looks like.
- How its APIs are structured.
  - A few highlights of its global variables and built-in modules.
- How it runs JavaScript in a single thread via an *event loop*.
- Options for concurrent JavaScript on this platform.

4.1 The Node.js platform

The following diagram provides an overview of how Node.js is structured:



The APIs available to a Node.js app consist of:

- The ECMAScript standard library (which is part of the language)
- Node.js APIs (which are not part of the language proper):
  - Some of the APIs are provided via global variables:
    - Especially cross-platform web APIs such as `fetch` and `CompressionStream` fall into this category.
    - But a few Node.js-only APIs are global, too – for example, `process`.
  - The remaining Node.js APIs are provided via built-in modules – for example, `'node:path'` (functions and constants for handling file system paths) and `'node:fs'` (functionality related to the file system).

The Node.js APIs are partially implemented in JavaScript, partially in C++. The latter is needed to interface with the operating system.

Node.js runs JavaScript via an embedded V8 JavaScript engine (the same engine used by Google's Chrome browser).

4.1.1 Global Node.js variables

These are a few highlights of Node's global variables:

- `crypto` gives us access to a web-compatible crypto API.
- `console` has much overlap with the same global variable in browsers (`console.log()` etc.).

- `fetch()` lets us use the Fetch browser API.
- `process` contains an instance of class `Process` and gives us access to command line arguments, standard input, standard out, and more.
- `structuredClone()` is a browser-compatible function for cloning objects.
- `URL` is a browser-compatible class for handling URLs.

More global variables are mentioned throughout this chapter.

4.1.1.1 Using modules instead of global variables

The following built-in modules provide alternatives to global variables:

- `'node:console'` is an alternative to the global variable `console`:

```
console.log('Hello!');

import {log} from 'node:console';
log('Hello!');
```

- `'node:process'` is an alternative to the global variable `process`:

```
console.log(process.argv);

import {argv} from 'node:process';
console.log(process.argv);
```

In principle, using modules is cleaner than using global variables. However, using the global variables `console` and `process` are such established patterns that deviating from them also has downsides.

4.1.2 The built-in Node.js modules

Most of Node's APIs are provided via modules. These are a few frequently used ones (in alphabetical order):

- `'node:assert/strict'`: Assertions are functions that check if a condition is met and report an error if not. They can be used in application code and for unit testing. This is an example of using this API:

```
import * as assert from 'node:assert/strict';
assert.equal(3 + 4, 7);
assert.equal('abc'.toUpperCase(), 'ABC');

assert.deepEqual({prop: true}, {prop: true}); // deep comparison
assert.notEqual({prop: true}, {prop: true}); // shallow comparison
```

- `'node:child_process'` is for running native commands synchronously or in separate processes. This module is described in §12 "Running shell commands in child processes".

- `'node:fs'` provides file system operations such as reading, writing, copying and deleting files and directories. For more information, see §8 "Working with the file system on Node.js".
- `'node:os'` contains operating-system-specific constants and utility functions. Some of them are explained in §7 "Working with file system paths and file URLs on Node.js".
- `'node:path'` is a cross-platform API for working with file system paths. It is described in §7 "Working with file system paths and file URLs on Node.js".
- `'node:stream'` contains a Node.js-specific streams API which are explained in §9 "Native Node.js streams".
  - Node.js also supports the cross-platform web streams API which is the subject of §10 "Using web streams on Node.js".
- `'node:util'` contains various utility functions.
  - Function `util.parseArgs()` is described in [content not included].

Module `'node:module'` contains function `builtinModules()` which returns an Array with the specifiers of all built-in modules:

```
import * as assert from 'node:assert/strict';
import {builtinModules} from 'node:module';
// Remove internal modules (whose names start with underscores)
const modules = builtinModules.filter(m => !m.startsWith('_'));
modules.sort();
assert.deepEqual(
  modules.slice(0, 5),
  [
    'assert',
    'assert/strict',
    'async_hooks',
    'buffer',
    'child_process',
  ]
);
```

4.1.3 The different styles of Node.js functions

In this section, we use the following import:

```
import * as fs from 'node:fs';
```

Node's functions come in three different styles. Let's look at the built-in module `'node:fs'` as an example:

- A synchronous style with normal functions – for example:
  - `fs.readFileSync(path, options?): string|Buffer`
- Two asynchronous styles:
  - An asynchronous style with callback-based functions – for example:
    - `fs.readFile(path, options?, callback): void`

  - An asynchronous style with Promise-based functions – for example:
    * `fsPromises.readFile(path, options?): Promise<string|Buffer>`

The three examples we have just seen, demonstrate the naming convention for functions with similar functionality:

- A callback-based function has a base name: `fs.readFile()`
- Its Promise-based version has the same name, but in a different module: `fsPromises.readFile()`
- The name of its synchronous version is the base name plus the suffix "Sync": `fs.readFileSync()`

Let's take a closer look at how these three styles work.

4.1.3.1 Synchronous functions

Synchronous functions are simplest – they immediately return values and throw errors as exceptions:

```
try {
  const result = fs.readFileSync('/etc/passwd', {encoding: 'utf-8'});
  console.log(result);
} catch (err) {
  console.error(err);
}
```

4.1.3.2 Promise-based functions

Promise-based functions return Promises that are fulfilled with results and rejected with errors:

```
import * as fsPromises from 'node:fs/promises'; // (A)

try {
  const result = await fsPromises.readFile(
    '/etc/passwd', {encoding: 'utf-8'});
  console.log(result);
} catch (err) {
  console.error(err);
}
```

Note the module specifier in line A: The Promise-based API is located in a different module.

Promises are explained in more detail in "JavaScript for impatient programmers".

4.1.3.3 Callback-based functions

Callback-based functions pass results and errors to callbacks which are their last parameters:

```
fs.readFile('/etc/passwd', {encoding: 'utf-8'},
  (err, result) => {
```

```
    if (err) {
      console.error(err);
      return;
    }
    console.log(result);
  }
);
```

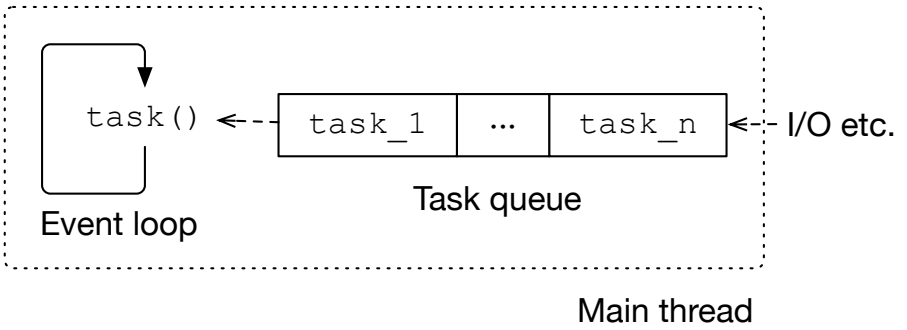
This style is explained in more detail in the Node.js documentation.

4.2 The Node.js event loop

By default, Node.js executes all JavaScript in a single thread, the *main thread*. The main thread continuously runs the *event loop* – a loop that executes chunks of JavaScript. Each chunk is a callback and can be considered a cooperatively scheduled task. The first task contains the code (coming from a module or standard input) that we start Node.js with. Other tasks are usually added later, due to:

- Code manually adding tasks
- I/O (input or output) with the file system, with network sockets, etc.
- Etc.

A first approximation of the event loop looks like this:



That is, the main thread runs code similar to:

```
while (true) { // event loop
  const task = taskQueue.dequeue(); // blocks
  task();
}
```

The event loop takes callbacks out of a *task queue* and executes them in the main thread. Dequeuing *blocks* (pauses the main thread) if the task queue is empty.

We'll explore two topics later:

- How to exit from the event loop.
- How to get around the limitation of JavaScript running in a single thread.

Why is this loop called *event loop*? Many tasks are added in response to events, e.g. ones sent by the operating system when input data is ready to be processed.

How are callbacks added to the task queue? These are common possibilities:

- JavaScript code can add tasks to the queue so that they are executed later.
- When an *event emitter* (a source of events) fires an event, the invocations of the event listeners are added to the task queue.
- Callback-based asynchronous operations in the Node.js API follow this pattern:
  - We ask for something and give Node.js a callback function with which it can report the result to us.
  - Eventually, the operation runs either in the main thread or in an external thread (more on that later).
  - When it is done, an invocation of the callback is added to the task queue.

The following code shows an asynchronous callback-based operation in action. It reads a text file from the file system:

```
import * as fs from 'node:fs';

function handleResult(err, result) {
  if (err) {
    console.error(err);
    return;
  }
  console.log(result); // (A)
}
fs.readFile('reminder.txt', 'utf-8',
  handleResult
);
console.log('AFTER'); // (B)
```

This is the ouput:

```
AFTER
Don't forget!
```

`fs.readFile()` executes the code that reads the file in another thread. In this case, the code succeeds and adds this callback to the task queue:

```
() => handleResult(null, 'Don't forget!')
```

4.2.1 Running to completion makes code simpler

An important rule for how Node.js runs JavaScript code is: Each task finishes ("runs to completion") before other tasks run. We can see that in the previous example: `'AFTER'` in line B is logged before the result is logged in line A because the initial task finishes before the task with the invocation of `handleResult()` runs.

Running to completion means that task lifetimes don't overlap and we don't have to worry about shared data being changed in the background. That simplifies Node.js code. The next example demonstrates that. It implements a simple HTTP server:

```
// server.mjs
import * as http from 'node:http';
```

```
let requestCount = 1;
const server = http.createServer(
  (_req, res) => { // (A)
    res.writeHead(200);
    res.end('This is request number ' + requestCount); // (B)
    requestCount++; // (C)
  }
);
server.listen(8080);
```

We run this code via `node server.mjs`. After that, the code starts and waits for HTTP requests. We can send them by using a web browser to go to `http://localhost:8080`. Each time we reload that HTTP resource, Node.js invokes the callback that starts in line A. It serves a message with the current value of variable `requestCount` (line B) and increments it (line C).

Each invocation of the callback is a new task and variable `requestCount` is shared between tasks. Due to running to completion, it is easy to read and update. There is no need to synchronize with other concurrently running tasks because there aren't any.

4.2.2 Why does Node.js code run in a single thread?

Why does Node.js code run in a single thread (with an event loop) by default? That has two benefits:

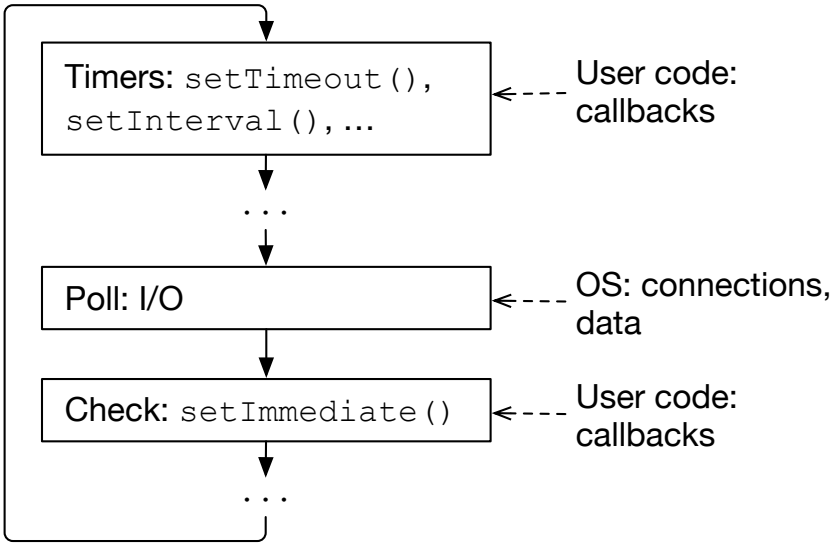
- As we have already seen, sharing data between tasks is simpler if there is only a single thread.
- In traditional multi-threaded code, an operation that takes longer to complete blocks the current thread until the operation is finished. Examples of such operations are reading a file or processing HTTP requests. Performing many of these operations is expensive because we have to create a new thread each time. With an event loop, the per-operation cost is lower, especially if each operation doesn't do much. That's why event-loop-based web servers can handle higher loads than thread-based ones.

Given that some of Node's asynchronous operations run in threads other than the main thread (more on that soon) and report back to JavaScript via the task queue, Node.js is not really single-threaded. Instead, we use a single thread to coordinate operations that run concurrently and asynchronously (in the main thread).

This concludes our first look at the event loop. **Feel free to skip the remainder of this section** if a superficial explanation is enough for you. Read on to learn more details.

4.2.3 The real event loop has multiple phases

The real event loop has multiple task queues from which it reads in multiple phases (you can check out some of the JavaScript code in the GitHub repository `nodejs/node`). The following diagram shows the most important ones of those phases:



What do the event loop phases do that are shown in the diagram?

- Phase "timers" invokes *timed tasks* that were added to its queue by:
  - `setTimeout(task, delay=1)` runs the callback `task` after `delay` milliseconds.
  - `setInterval(task, delay=1)` runs the callback `task` repeatedly, with pauses lasting `delay` milliseconds.
- Phase "poll" retrieves and processes I/O events and runs I/O-related tasks from its queue.
- Phase "check" (the "immediate phase") executes tasks scheduled via:
  - `setImmediate(task)` runs the callback `task` as soon as possible ("immediately" after phase "poll").

Each phase runs until its queue is empty or until a maximum number of tasks was processed. Except for "poll", each phase waits until its next turn before it processes tasks that were added during its run.

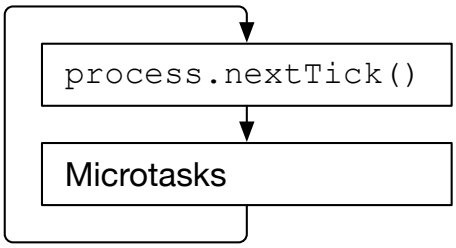
4.2.3.1 Phase "poll"

- If the poll queue is not empty, the poll phase will go through it and run its tasks.
- Once the poll queue is empty:
  - If there are `setImmediate()` tasks, processing advances to the "check" phase.
  - If there are timer tasks that are ready, processing advances to the "timers" phase.
  - Otherwise, this phase blocks the whole main thread and waits until new tasks are added to the poll queue (or until this phase ends, see below). These are processed immediately.

If this phase takes longer than a system-dependent time limit, it ends and the next phase runs.

4.2.4 Next-tick tasks and microtasks

After each invoked task, a "sub-loop" runs that consists of two phases:



The sub-phases handle:

- Next-tick tasks, as enqueued via `process.nextTick()`.
- Microtasks, as enqueued via `queueMicrotask()`, Promise reactions, etc.

Next-tick tasks are Node.js-specific, Microtasks are a cross-platform web standard (see MDN's support table).

This sub-loop runs until both queues are empty. Tasks added during its run, are processed immediately – the sub-loop does not wait until its next turn.

4.2.5 Comparing different ways of directly scheduling tasks

We can use the following functions and methods to add callbacks to one of the task queues:

- Timed tasks (phase "timers")
  - `setTimeout()` (web standard)
  - `setInterval()` (web standard)
- Untimed tasks (phase "check")
  - `setImmediate()` (Node.js-specific)
- Tasks that run immediately after the current task:
  - `process.nextTick()` (Node.js-specific)
  - `queueMicrotask()`: (web standard)

It's important to note that when timing a task via a delay, we are specifying the earliest possible time that the task will run. Node.js cannot always run them at exactly the scheduled time because it can only check between tasks if any timed tasks are due. Therefore, a long-running task can cause timed tasks to be late.

4.2.5.1 Next-tick tasks and microtasks vs. normal tasks

Consider the following code:

```
function enqueueTasks() {
  Promise.resolve().then(() => console.log('Promise reaction 1'));
  queueMicrotask(() => console.log('queueMicrotask 1'));
  process.nextTick(() => console.log('nextTick 1'));
  setImmediate(() => console.log('setImmediate 1')); // (A)
  setTimeout(() => console.log('setTimeout 1'), 0);
```

```
  Promise.resolve().then(() => console.log('Promise reaction 2'));
  queueMicrotask(() => console.log('queueMicrotask 2'));
  process.nextTick(() => console.log('nextTick 2'));
  setImmediate(() => console.log('setImmediate 2')); // (B)
  setTimeout(() => console.log('setTimeout 2'), 0);
}

setImmediate(enqueueTasks);
```

We use `setImmediate()` to avoid a pecularity of ESM modules: They are executed in microtasks, which means that if we enqueue microtasks at the top level of an ESM module, they run before next-tick tasks. As we'll see next, that's different in most other contexts.

This is the output of the previous code:

```
nextTick 1
nextTick 2
Promise reaction 1
queueMicrotask 1
Promise reaction 2
queueMicrotask 2
setTimeout 1
setTimeout 2
setImmediate 1
setImmediate 2
```

Observations:

- All next-tick tasks are executed immediately after `enqueueTasks()`.
- They are followed by all microtasks, including Promise reactions.
- Phase "timers" comes after the immediate phase. That's when the timed tasks are executed.
- We have added immediate tasks during the immediate ("check") phase (line A and line B). They show up last in the output, which means that they were not executed during the current phase, but during the next immediate phase.

4.2.5.2 Enqueuing next-tick tasks and microtasks during their phases

The next code examines what happens if we enqueue a next-tick task during the next-tick phase and a microtask during the microtask phase:

```
setImmediate(() => {
  setImmediate(() => console.log('setImmediate 1'));
  setTimeout(() => console.log('setTimeout 1'), 0);

  process.nextTick(() => {
    console.log('nextTick 1');
    process.nextTick(() => console.log('nextTick 2'));
  });
```

```
  queueMicrotask(() => {
    console.log('queueMicrotask 1');
    queueMicrotask(() => console.log('queueMicrotask 2'));
    process.nextTick(() => console.log('nextTick 3'));
  });
});
```

This is the output:

```
nextTick 1
nextTick 2
queueMicrotask 1
queueMicrotask 2
nextTick 3
setTimeout 1
setImmediate 1
```

Observations:

- Next-tick tasks are executed first.
- "nextTick 2" in enqueued during the next-tick phase and immediately executed. Execution only continues once the next-tick queue is empty.
- The same is true for microtasks.
- We enqueue "nextTick 3" during the microtask phase and execution loops back to the next-tick phase. These subphases are repeated until both their queues are empty. Only then does execution move on to the next global phases: First the "timers" phase ("setTimeout 1"). Then the immediate phase ("setImmediate 1").

4.2.5.3 Starving out event loop phases

The following code explores which kinds of tasks can *starve out* event loop phases (prevent them from running via infinite recursion):

```
import * as fs from 'node:fs/promises';

function timers() { // OK
  setTimeout(() => timers(), 0);
}
function immediate() { // OK
  setImmediate(() => immediate());
}

function nextTick() { // starves I/O
  process.nextTick(() => nextTick());
}

function microtasks() { // starves I/O
  queueMicrotask(() => microtasks());
```

```
}

timers();
console.log('AFTER'); // always logged
console.log(await fs.readFile('./file.txt', 'utf-8'));
```

The "timers" phase and the immediate phase don't execute tasks that are enqueued during their phases. That's why `timers()` and `immediate()` don't starve out `fs.readFile()` which reports back during the "poll" phase (there is also a Promise reaction, but let's ignore that here).

Due to how next-tick tasks and microtasks are scheduled, both `nextTick()` and `microtasks()` prevent the output in the last line.

4.2.6 When does a Node.js app exit?

At the end of each iteration of the event loop, Node.js checks if it's time to exit. It keeps a reference count of pending *timeouts* (for timed tasks):

- Scheduling a timed task via `setImmediate()`, `setInterval()`, or `setTimeout()` increases the reference count.
- Running a timed task decreases the reference count.

If the reference count is zero at the end of an event loop iteration, Node.js exits.

We can see that in the following example:

```
function timeout(ms) {
  return new Promise(
    (resolve, _reject) => {
      setTimeout(resolve, ms); // (A)
    }
  );
}
await timeout(3_000);
```

Node.js waits until the Promise returned by `timeout()` is fulfilled. Why? Because the task we schedule in line A keeps the event loop alive.

In contrast, creating Promises does not increase the reference count:

```
function foreverPending() {
  return new Promise(
    (_resolve, _reject) => {}
  );
}
await foreverPending(); // (A)
```

In this case, execution temporarily leaves this (main) task during `await` in line A. At the end of the event loop, the reference count is zero and Node.js exits. However, the exit is not successful. That is, the exit code is not 0, it is 13 ("Unfinished Top-Level Await").

We can manually control whether a timeout keeps the event loop alive: By default, tasks scheduled via `setImmediate()`, `setInterval()`, and `setTimeout()` keep the event loop

alive as long as they are pending. These functions return instances of class `Timeout` whose method `.unref()` changes that default so that the timeout being active won't prevent Node.js from exiting. Method `.ref()` restores the default.

Tim Perry mentions a use case for `.unref()`: His library used `setInterval()` to repeatedly run a background task. That task prevented applications from exiting. He fixed the issue via `.unref()`.

4.3 libuv: the cross-platform library that handles asynchronous I/O (and more) for Node.js

libuv is a library written in C that supports many platforms (Windows, macOS, Linux, etc.). Node.js uses it to handle I/O and more.

4.3.1 How libuv handles asynchronous I/O

Network I/O is asynchronous and doesn't block the current thread. Such I/O includes:

- TCP
- UDP
- Terminal I/O
- Pipes (Unix domain sockets, Windows named pipes, etc.)

To handle asynchronous I/O, libuv uses native kernel APIs and subscribes to I/O events (epoll on Linux; kqueue on BSD Unix incl. macOS; event ports on SunOS; IOCP on Windows). It then gets notifications when they occur. All of these activities, including the I/O itself, happen on the main thread.

4.3.2 How libuv handles blocking I/O

Some native I/O APIs are blocking (not asynchronous) – for example, file I/O and some DNS services. libuv invokes these APIs from threads in a thread pool (the so-called "worker pool"). That enables the main thread to use these APIs asynchronously.

4.3.3 libuv functionality beyond I/O

libuv helps Node.js with more than just with I/O. Other functionality includes:

- Running tasks in the thread pool
- Signal handling
- High resolution clock
- Threading and synchronization primitives

As an aside, libuv has its own event loop whose source code you can check out in the GitHub repository `libuv/libuv` (function `uv_run()`).

4.4 Escaping the main thread with user code

If we want to keep Node.js responsive to I/O, we should avoid performing long-running computations in main-thread tasks. There are two options for doing so:

- Partitioning: We can split up the computation into smaller pieces and run each piece via `setImmediate()`. That enables the event loop to perform I/O between the pieces.
  - An upside is that we can perform I/O in each piece.
  - A downside is that we still slow down the event loop.
- Offloading: We can perform our computation in a different thread or process.
  - Downsides are that we can't perform I/O from threads other than the main thread and that communicating with outside code becomes more complicated.
  - Upsides are that we don't slow down the event loop, that we can make better use of multiple processor cores, and that errors in other threads don't affect the main thread.

The next subsections cover a few options for offloading.

4.4.1 Worker threads

Worker Threads implement the cross-platform Web Workers API with a few differences – e.g.:

- Worker Threads have to be imported from a module, Web Workers are accessed via a global variable.
- Inside a worker, listening to messages and posting messages is done via methods of the global object in browsers. On Node.js, we import `parentPort` instead.
- We can use most Node.js APIs from workers. In browsers, our choice is more limited (we can't use the DOM, etc.).
- On Node.js, more objects are transferable (all objects whose classes extend the internal class `JSTransferable`) than in browsers.

On one hand, Worker Threads really are threads: They are more lightweight than processes and run in the same process as the main thread.

On the other hand:

- Each worker runs its own event loop.
- Each worker has its own JavaScript engine instance and its own Node.js instance – including separate global variables.
  - (Specifically, each worker is an *V8 isolate* that has its own JavaScript heap but shares its operating system heap with other threads.)
- Sharing data between threads is limited:
  - We can share binary data/numbers via SharedArrayBuffers.
  - `Atomics` offers atomic operations and synchronization primitives that help when using SharedArrayBuffers.

  - The Channel Messaging API lets us send data ("messages") over two-way channels. The data is either *cloned* (copied) or *transferred* (moved). The latter is more efficient and only supported by a few data structures.

For more information, see the Node.js documentation on worker threads.

4.4.2 Clusters

Cluster is a Node.js-specific API. It lets us run *clusters* of Node.js processes that we can use to distribute workloads. The processes are fully isolated but share server ports. They can communicate by passing JSON data over channels.

If we don't need process isolation, we can use Worker Threads which are more lightweight.

4.4.3 Child processes

Child process is another Node.js-specific API. It lets us spawn new processes that run native commands (often via native shells). This API is covered in §12 "Running shell commands in child processes".

4.5 Sources of this chapter

Node.js event loop:

- Node.js documentation: "The Node.js Event Loop, Timers, and `process.nextTick()`"
- "What you should know to really understand the Node.js Event Loop" by Daniel Khan
- "How does Node.js decide whether to exit the event loop or go around again?" by Mark Meyer

Videos on the event loop (which refresh some of the background knowledge needed for this chapter):

- "Node's Event Loop From the Inside Out" (by Sam Roberts) explains why operating systems added support for asynchronous I/O; which operations are asynchronous and which aren't (and have to run in the thread pool); etc.
- "The Node.js Event Loop: Not So Single Threaded" (by Bryan Hughes) contains a brief history of multitasking (cooperative multitasking, preemptive multitasking, symmteric multi-threading, asynchronous multitasking); processes vs. threads; running I/O synchronously vs. in the thread pool; etc.

libuv:

- libuv documentation:
  - "Design overview"
  - "Basics of libuv"
- "A deep dive into libuv" by Saúl Ibarra Corretgé
- "I/O multiplexing (select vs. poll vs. epoll/kqueue) - problems and algorithms" by Nima Aghdaii

- "Developer Initiates I/O Operation. You Won't Believe What Happens Next." by Colin J. Ihrig
  - Traces a JavaScript function call as it goes from JavaScript to Node's core to libuv and back.

JavaScript concurrency:

- Section "Complex calculations without blocking the Event Loop" in "Don't Block the Event Loop (or the Worker Pool)" in the Node.js documentation
- "Understanding Worker Threads in Node.js" by Liz Parody
- "The State Of Web Workers In 2021" by Surma
- Video "Node.js: The Road to Workers" by Anna Henningsen

4.5.1 Acknowledgement

- I'm much obliged to Dominic Elm for reviewing this chapter and providing important feedback.

Chapter 5

Packages: JavaScript's units for software distribution

Contents

This chapter explains what npm packages are and how they interact with ESM modules.

Required knowledge: I'm assuming that you are loosely familiar with the syntax of ECMAScript modules. If you are not, you can read chapter "modules" in "JavaScript for impatient programmers".

5.1 What is a package?

In the JavaScripte ecosystem, a *package* is a way of organizing software projects: It is a directory with a standardized layout. A package can contain all kinds of files - for example:

- A web application written in JavaScript, to be deployed on a server
- JavaScript libraries (for Node.js, for browsers, for all JavaScript platforms, etc.)
- Libraries for programming languages other than JavaScript: TypeScript, Rust, etc.
- Unit tests (e.g. for the libraries in the package)
- *Bin scripts* – Node.js-based shell scripts – e.g., development tools such as compilers, test runners, and documentation generators
- Many other kinds of artifacts

A package can *depend on* other packages (which are called its *dependencies*) which contain:

- Libraries needed by the package's JavaScript code
- Shell scripts used during development
- Etc.

The dependencies of a package are installed inside that package (we'll see how soon).

One common distinction between packages is:

- *Published packages* can be installed by us:
  - Global installation: We can install them globally so that their bin scripts become available at the command line.
  - Local installation: We can install them as dependencies into our own packages. Their bin scripts can be used locally (we'll see how soon).
- *Unpublished packages* never become dependencies of other packages, but do have dependencies themselves. Examples include web applications that are deployed to servers.

The next subsection explains how packages can be published.

5.1.1 Publishing packages: package registries, package managers, package names

The main way of publishing a package is to upload it to a package registry – an online software repository. The de facto standard is the *npm registry* but it is not the only option. For example, companies can host their own internal registries.

A *package manager* is a command line tool that downloads packages from a registry (or other sources) and installs them locally or globally. If a package contains bin scripts, it also makes those available locally or globally.

The most popular package manager is called *npm* and comes bundled with Node.js. Its name originally stood for "Node Package Manager". Later, when npm and the npm registry were used not only for Node.js packages, the definition was changed to "npm is not a package manager" (source).

There are other popular package managers such as yarn and pnpm. All of these package managers use the npm registry by default.

Each package in the npm registry has a name. There are two kinds of names:

- *Global names* are unique across the whole registry. These are two examples:

  ```
  minimatch
  mocha
  ```

- *Scoped names* consist of two parts: A scope and a name. Scopes are globally unique, names are unique per scope. These are two examples:

  ```
  @babel/core
  @rauschma/iterable
  ```

  The scope starts with an `@` symbol and is separated from the name with a slash.

5.2 The file system layout of a package

Once a package `my-package` is fully installed, it almost always looks like this:

```
my-package/
  package.json
  node_modules/
  [More files]
```

What are the purposes of these file system entries?

- `package.json` is a file every package must have:
  - It contains metadata describing the package (its name, its version, its author, etc.).
  - It lists the dependencies of the package: other packages that it needs, such as libraries and tools. Per dependency, we record:
    * A range of version numbers. Not specifying a specific version allows for upgrades and for code sharing between dependencies.
    * By default, dependencies come from the npm registry. But we can also specify other sources: a local directory, a GZIP file, a URL pointing to a GZIP file, a registry other than npm's, a git repository, etc.
- `node_modules/` is a directory into which the dependencies of the package are installed. Each dependency also has a `node_modules` folder with its dependencies, etc. The result is a tree of dependencies.

Some packages also have the file `package-lock.json` that sits next to `package.json`: It records the exact versions of the dependencies that were installed and is kept up to date if we add more dependencies via npm.

5.2.1 `package.json`

This is a starter `package.json` that can be created via npm:

```
{
  "name": "my-package",
  "version": "1.0.0",
  "description": "",
```

```
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

What are the purposes of these properties?

- Some properties are required for public packages (published on the npm registry):
  - `name` specifies the name of this package.
  - `version` is used for version management and follows semantic versioning with three dot-separated numbers:
    - The *major version* is incremented when incompatible API changes are made.
    - The *minor version* is incremented when functionality is added in a backward compatible manner.
    - The *patch version* is incremented when small changes are made that don't really change the functionality.
- Other properties for public packages are optional:
  - `description`, `keywords`, `author` are optional and make it easier to find packages.
  - `license` clarifies how this package can be used. It makes sense to provide this value if the package is public in any way. "Choose an open source license" can help with making this choice.
- `main` is a property for packages with library code. It specifies the module that "is" the package (explained later in this chapter).
- `scripts` is a property for setting up *package scripts* – abbreviations for development-time shell commands. These can be executed via `npm run`. For example, the script `test` can be executed via `npm run test`. For more on this topic, see [content not included].

Other useful properties:

- `dependencies` lists the dependencies of a package. Its format is explained soon.
- `devDependencies` are dependencies that are only needed during development.
- The following setting means that all files with the name extension `.js` are interpreted as ECMAScript modules. Unless we are dealing with legacy code, it makes sense to add it:

  ```
  "type": "module"
  ```

- `bin` lists *bin scripts*, Node.js modules within the package that npm installs as shell scripts. Its format is explained soon.
- `license` specifies a license for the package. Its format is explained soon.

- Normally, the properties `name` and `version` are required and npm warns us if they are missing. However, we can change that via the following setting:

  ```
  "private": true
  ```

  That prevents the package from accidentally being published and allows us to omit name and version.

For more information on `package.json`, see the npm documentation.

5.2.2 Property `"dependencies"` of `package.json`

This is what the dependencies in a `package.json` file look like:

```
"dependencies": {
  "minimatch": "^5.1.0",
  "mocha": "^10.0.0"
}
```

The properties record both the names of packages and constraints for their versions.

Versions themselves follow the semantic versioning standard. They are up to three numbers (the second and third number are optional and zero by default) separated by dots:

1. *Major version*: This number changes when a packages changes in incompatible ways.
2. *Minor version*: This number changes when functionality is added in a backward compatible manner.
3. *Patch version*: This number changes when backward compatible bug fixes are made.

Node's version ranges are explained in the `semver` repository. Examples include:

- A specific version without any extra characters means that the installed version must match the version exactly:

  ```
  "pkg1": "2.0.1",
  ```

- `major.minor.x` or `major.x` means that the components that are numbers must match, the components that are `x` or omitted can have any values:

  ```
  "pkg2": "2.x",
  "pkg3": "3.3.x",
  ```

- `*` matches any version:

  ```
  "pkg4": "*",
  ```

- `>=version` means that the installed version must be `version` or higher:

  ```
  "pkg5": ">=1.0.2",
  ```

- `<=version` means that the installed version must be `version` or lower:

  ```
  "pkg6": "<=2.3.4",
  ```

- `version1-version2` is the same as `>=version1 <=version2`:

```
"pkg7": "1.0.0 - 2.9999.9999",
```

- `^version` (as used in the previous example) is a *caret range* and means that the installed version can be `version` or higher but must not introduce breaking changes. That is, the major version must be the same:

  ```
  "pkg8": "^4.17.21",
  ```

5.2.3 Property `"bin"` of `package.json`

This is how we can tell npm to install modules as shell scripts:

```
"bin": {
  "my-shell-script": "./src/shell/my-shell-script.mjs",
  "another-script": "./src/shell/another-script.mjs"
}
```

If we install a package with this `"bin"` value globally, Node.js ensures that the commands `my-shell-script` and `another-script` become available at the command line.

If we install the package locally, we can use the two commands in package scripts or via the `npx` command.

A string is also allowed as the value of `"bin"`:

```
{
  "name": "my-package",
  "bin": "./src/main.mjs"
}
```

This is an abbreviation for:

```
{
  "name": "my-package",
  "bin": {
    "my-package": "./src/main.mjs"
  }
}
```

5.2.4 Property `"license"` of `package.json`

The value of property `"license"` is always a string with a SPDX license ID. For example, the following value denies others the right to use a package under any terms (which is useful if a package is unpublished):

```
"license": "UNLICENSED"
```

The SPDX website lists all available license IDs. If you find it difficult to pick one, the website "Choose an open source license" can help – for example, this is the advice if you "want it simple and permissive":

> The MIT License is short and to the point. It lets people do almost anything they want with your project, like making and distributing closed source versions.

Babel, .NET, and Rails use the MIT License.

You can use that license like this:

```
"license": "MIT"
```

5.3 Archiving and installing packages

Packages in the npm registry are often archived in two different ways:

- For development, they are stored in a git repository.
- To make them installable via npm, they are uploaded to the npm registry.

Either way, the package is archived without its dependencies – which we have to install before we can use it.

If a package is stored in a git repository:

- We normally want the same dependency tree to be used every time we install the package.
  - That's why `package-lock.json` is usually included.
- We can regenerate artifacts from other artifacts – for example, compile TypeScript files to JavaScript files.

If a package is published to the npm registry:

- It should be flexible with its dependencies so that upgrading dependencies and sharing packages in a dependency tree becomes possible.
  - That's why `package-lock.json` is never uploaded to the npm registry.
- It often contains generated artifacts - for example, JavaScript files compiled from TypeScript files are included so that people who only use JavaScript don't have to install a TypeScript compiler.

Dev dependencies (property `devDependencies` in `package.json`) are only installed during development but not when we install the package from the npm registry.

Note that unpublished packages in git repositories are handled similarly to published packages during development.

5.3.1 Installing a package from git

To install a package `pkg` from git, we clone its repository and:

```
cd pkg/
npm install
```

Then the following steps are performed:

- `node_modules` is created and the dependencies are installed. Installing a dependency also means downloading that dependency and installing its dependencies (etc.).
- Sometimes additional setup steps are performed. Which ones those are can be configured via `package.json`.

If the root package doesn't have a `package-lock.json` file, it is created during installation (as mentioned, dependencies don't have this file).

In a dependency tree, the same dependency may exist multiple times, possibly in different versions. There a ways to minimize duplication, but that is beyond the scope of this chapter.

5.3.1.1 Reinstalling a package

This is a (slightly crude) way of fixing issues in a dependency tree:

```
cd pkg/
rm -rf node_modules/
rm package-lock.json
npm install
```

Note that that may result in different, newer, packages being installed. We can avoid that by not deleting `package-lock.json`.

5.3.2 Creating a new package and installing dependencies

There are many tools and technique for setting up new packages. This is one simple way:

```
mkdir my-package
cd my-package/
npm init --yes
```

Afterward, the directory looks like this:

```
my-package/
  package.json
```

This `package.json` has the starter content that we have already seen.

5.3.2.1 Installing dependencies

Right now, `my-package` doesn't have any dependencies. Let's say we want to use the library `lodash-es`. This is how we install it into our package:

```
npm install lodash-es
```

This command performs the following steps:

- The package is downloaded into `my-package/node_modules/lodash-es`.
- Its dependencies are also installed. Then the dependencies of its dependencies. Etc.
- A new property is added to `package.json`:

  ```
  "dependencies": {
    "lodash-es": "^4.17.21"
  }
  ```

- `package-lock.json` is updated with the exact version that was installed.

5.4 Referring to modules via *specifiers*

Code in other ECMAScript modules is accessed via `import` statements (line A and line B):

```
// Static import
import {namedExport} from 'https://example.com/some-module.js'; // (A)
console.log(namedExport);

// Dynamic import
import('https://example.com/some-module.js') // (B)
.then((moduleNamespace) => {
  console.log(moduleNamespace.namedExport);
});
```

Both static imports and dynamic imports use *module specifiers* to refer to modules:

- The string after `from` in line A.
- The string argument in line B.

There are three kinds of module specifiers:

- *Absolute specifiers* are full URLs – for example:

  ```
  'https://www.unpkg.com/browse/yargs@17.3.1/browser.mjs'
  'file:///opt/nodejs/config.mjs'
  ```

  Absolute specifiers are mostly used to access libraries that are directly hosted on the web.

- *Relative specifiers* are relative URLs (starting with `'/'`, `'./'` or `'../'`) – for example:

  ```
  './sibling-module.js'
  '../module-in-parent-dir.mjs'
  '../../dir/other-module.js'
  ```

  Every module has a URL whose protocol depends on its location (`file:`, `https:`, etc.). If it uses a relative specifier, JavaScript turns that specifier into a full URL by resolving it against the module's URL.

  Relative specifiers are mostly used to access other modules within the same code base.

- *Bare specifiers* are paths (without protocol and domain) that start with neither slashes nor dots. They begin with the names of packages. Those names can optionally be followed by *subpaths*:

  ```
  'some-package'
  'some-package/sync'
  'some-package/util/files/path-tools.js'
  ```

  Bare specifiers can also refer to packages with scoped names:

  ```
  '@some-scope/scoped-name'
  '@some-scope/scoped-name/async'
  '@some-scope/scoped-name/dir/some-module.mjs'
  ```

Each bare specifier refers to exactly one module inside a package; if it has no subpath, it refers to the designated "main" module of its package. A bare specifier is never used directly but always *resolved* – translated to an absolute specifier. How resolution works depends on the platform. We'll learn more soon.

5.4.1 Filename extensions in module specifiers

- Absolute specifiers and relative specifiers always have filename extensions – usually `.js` or `.mjs`.
- There are three styles of bare specifiers:
  - Style 1: no subpath
  - Style 2: a subpath without a filename extension. In this case, the subpath works like a modifier for the package name:
    ```
    'my-parser/sync'
    'my-parser/async'

    'assertions'
    'assertions/strict'
    ```
  - Style 3: a subpath with a filename extension. In this case, the package is seen as a collection of modules and the subpath points to one of them:
    ```
    'large-package/misc/util.js'
    'large-package/main/parsing.js'
    'large-package/main/printing.js'
    ```

Caveat of style 3 bare specifiers: How the filename extension is interpreted depends on the dependency and may differ from the importing package. For example, the importing package may use `.mjs` for ESM modules and `.js` for CommonJS modules, while the ESM modules exported by the dependency may have bare paths with the filename extension `.js`.

5.5 Module specifiers in Node.js

Let's see how module specifiers work in Node.js.

5.5.1 Resolving module specifiers in Node.js

The *Node.js resolution algorithm* works as follows:

- Parameters:
  - URL of importing module
  - Module specifier
- Result: Resolved URL for module specifier

This is the algorithm:

- If a specifier is absolute, resolution is already finished. Three protocols are most common:
  - `file:` for local files
  - `https:` for remote files

  - `node:` for built-in modules (discussed later)

- If a specifier is relative, it is resolved against the URL of the importing module.
- If a specifier is bare:
  - If it starts with `'#'`, it is resolved by looking it up among the *package imports* (which are explained later) and resolving the result.
  - Otherwise, it is a bare specifier that has one of these formats (the subpath is optional):

    * `«package»/sub/path`
    * `@«scope»/«scoped-package»/sub/path`

    The resolution algorithm traverses the current directory and its ancestors until it finds a directory `node_modules` that has a subdirectory matching the beginning of the bare specifier, i.e. either:

    * `node_modules/«package»/`
    * `node_modules/@«scope»/«scoped-package»/`

    That directory is the directory of the package. By default, the (potentially empty) subpath after the package ID is interpreted as relative to the package directory. The default can be overridden via *package exports* which are explained next.

The result of the resolution algorithm must point to a file. That explains why absolute specifiers and relative specifiers always have filename extensions. Bare specifiers mostly don't because they are abbreviations that are looked up in package exports.

Module files usually have these filename extensions:

- If a file has the name extension `.mjs`, it is always an ES module.
- A file that has the name extension `.js` is an ES module if the closest `package.json` has this entry:
  - `"type": "module"`

If Node.js executes code provided via stdin, `--eval` or `--print`, we use the following command-line option so that it is interpreted as an ES module:

```
--input-type=module
```

5.5.2 Package exports: controlling what other packages see

In this subsection, we are working with a package that has the following file layout:

```
my-lib/
  dist/
    src/
      main.js
      util/
        errors.js
      internal/
```

```
        internal-module.js
    test/
```

Package exports are specified via property `"exports"` in `package.json` and support two important features:

- Hiding the internals of a package:
  - Without property `"exports"`, every module in package `my-lib` can be accessed via a relative path after the package name – e.g.:
    `'my-lib/dist/src/internal/internal-module.js'`
  - Once the property exists, only specifiers listed in it can be used. Everything else is hidden from the outside.
- Nicer module specifiers: Package export let us define bare specifier subpaths for modules that are shorter and/or have better names.

Recall the three styles of bare specifiers:

- Style 1: bare specifiers without subpaths
- Style 2: bare specifiers with extension-less subpaths
- Style 3: bare specifiers with subpaths with extensions

Package exports help us with all three styles

5.5.2.1 Style 1: configuring which file represents (the bare specifier for) the package

`package.json`:

```
{
  "main": "./dist/src/main.js",
  "exports": {
    ".": "./dist/src/main.js"
  }
}
```

We only provide `"main"` for backward-compatibility (with older bundlers and Node.js 12 and older). Otherwise, the entry for `"."` is enough.

With these package exports, we can now import from `my-lib` as follows.

```
import {someFunction} from 'my-lib';
```

This imports `someFunction()` from this file:

```
my-lib/dist/src/main.js
```

5.5.2.2 Style 2: mapping extension-less subpaths to module files

`package.json`:

```
{
  "exports": {
    "./util/errors": "./dist/src/util/errors.js"
  }
}
```

We are mapping the specifier subpath `'util/errors'` to a module file. That enables the following import:

```
import {UserError} from 'my-lib/util/errors';
```

5.5.2.3 Style 2: better subpaths without extensions for a subtree

The previous subsection explained how to create a single mapping for an extension-less subpath. There is also a way to create multiple such mappings via a single entry:

`package.json`:

```
{
  "exports": {
    "./lib/*": "./dist/src/*.js"
  }
}
```

Any file that is a descendant of `./dist/src/` can now be imported without a filename extension:

```
import {someFunction} from 'my-lib/lib/main';
import {UserError}    from 'my-lib/lib/util/errors';
```

Note the asterisks in this `"exports"` entry:

```
"./lib/*": "./dist/src/*.js"
```

These are more instructions for how to map subpaths to actual paths than wildcards that match fragments of file paths.

5.5.2.4 Style 3: mapping subpaths with extensions to module files

`package.json`:

```
{
  "exports": {
    "./util/errors.js": "./dist/src/util/errors.js"
  }
}
```

We are mapping the specifier subpath `'util/errors.js'` to a module file. That enables the following import:

```
import {UserError} from 'my-lib/util/errors.js';
```

5.5.2.5 Style 3: better subpaths with extensions for a subtree

`package.json`:

```
{
  "exports": {
    "./*": "./dist/src/*"
  }
}
```

Here, we shorten the module specifiers of the whole subtree under `my-package/dist/src`:

```
import {InternalError} from 'my-package/util/errors.js';
```

Without the exports, the import statement would be:

```
import {InternalError} from 'my-package/dist/src/util/errors.js';
```

Note the asterisks in this `"exports"` entry:

```
"./*": "./dist/src/*"
```

These are not filesystem globs but instructions for how to map external module specifiers to internal ones.

5.5.2.6 Exposing a subtree while hiding parts of it

With the following trick, we expose everything in directory `my-package/dist/src/` with the exception of `my-package/dist/src/internal/`

```
"exports": {
  "./*": "./dist/src/*",
  "./internal/*": null
}
```

Note that this trick also works when exporting subtrees *without* filename extensions.

5.5.2.7 Conditional package exports

We can also make exports *conditional*: Then a given path maps to different values depending on the context in which a package is used.

Node.js vs. browsers. For example, we could provide different implementations for Node.js and for browsers:

```
"exports": {
  ".": {
    "node": "./main-node.js",
    "browser": "./main-browser.js",
    "default": "./main-browser.js"
  }
}
```

The `"default"` condition matches when no other key matches and must come last. Having one is recommended whenever we are distinguishing between platforms because it takes care of new and/or unknown platforms.

Development vs. production. Another use case for conditional package exports is switching between "development" and "production" environments:

```
"exports": {
  ".": {
    "development": "./main-development.js",
    "production": "./main-production.js",
```

```
    }
  }
```

In Node.js we can specify an environment like this:

```
node --conditions development app.mjs
```

5.5.3 Package imports

Package imports let a package define abbreviations for module specifiers that it can use itself, internally (where package exports define abbreviations for other packages). This is an example:

`package.json`:

```
{
  "imports": {
    "#some-pkg": {
      "node": "some-pkg-node-native",
      "default": "./polyfills/some-pkg-polyfill.js"
    }
  },
  "dependencies": {
    "some-pkg-node-native": "^1.2.3"
  }
}
```

The package import `#` is *conditional* (with the same features as conditional package exports):

- If the current package is used on Node.js, the module specifier `'#some-pkg'` refers to package `some-pkg-node-native`.
- Elsewhere, `'#some-pkg'` refers to the file `./polyfills/some-pkg-polyfill.js` inside the current package.

(Only package imports can refer to external packages, package exports can't do that.)

What are the use cases for package imports?

- Referring to different platform-specific implementations modules via the same module specifier (as demonstrated above).
- Aliases to modules inside the current package – to avoid relative specifiers (which can get complicated with deeply nested directories).

Be careful when using package imports with a bundler: This feature is relatively new and your bundler may not support it.

5.5.4 `node:` protocol imports

Node.js has many built-in modules such as `'path'` and `'fs'`. All of them are available as both ES modules and CommonJS modules. One issue with them is that they can be

overridden by modules installed in `node_modules` which is both a security risk (if it happens accidentally) and a problem if Node.js wants to introduce new built-in modules in the future and their names are already taken by npm packages.

We can use the `node:` protocol to make it clear that we want to import a built-in module. For example, the following two import statements are mostly equivalent (if no npm module is installed that has the name `'fs'`):

```
import * as fs from 'node:fs/promises';
import * as fs from 'fs/promises';
```

An additional benefit of using the `node:` protocol is that we immediately see that an imported module is built-in. Given how many built-in modules there are, that helps when reading code.

Due to `node:` specifiers having a protocol, they are considered absolute. That's why they are not looked up in `node_modules`.

Chapter 6

An overview of npm (a package manager for JavaScript)

Contents

6.1 The npm package manager

The *npm registry* is the de-facto standard for hosting JavaScript packages. Those packages have a particular format and are called *npm packages*.

Therefore, in the JavaScript ecosystem, a *package manager* is a command line tool for installing npm packages – from the npm registry or other sources.

The most popular package manager is called *npm* and comes bundled with Node.js. Its name originally stood for "Node Package Manager". Later, when npm and the npm registry were used not just for Node.js packages, the definition was changed to "npm is not a package manager" (source).

There are other popular package managers such as yarn and pnpm. All of these package managers use the npm registry by default.

We use npm via the shell command `npm` which provides several subcommands such as `npm install`.

6.2 Getting help for npm

6.2.1 Getting help on the command line

We can use the `npm` command to explain itself: On one hand, there is the option `-h` which can be used after `npm` and after npm commands. It provides brief explanations:

```
npm -h        # brief explanation of `npm`
npm <cmd> -h  # brief explanation of `npm <cmd>`
```

On the other hand, there is the command `npm help` which provides longer explanations:

```
npm help         # brief explanation of `npm` (same as `npm -h`)
npm help npm     # longer explanation of `npm`
npm help <cmd>   # longer explanation of `npm <cmd>`
npm help <topic> # longer explanation of <topic>
```

Help topics include:

- `folders`
- `npmrc`
- `package.json`

6.2.2 Getting help online

The official npm documentation is also available online.

6.3 Common npm commands

These are a few common commands:

- `npm init` "initializes" the current directory to be a package. That is, it creates the file `package.json` in it. This command is explained in [content not included].
- `npm install` installs npm packages globally or locally. It is explained in [content not included].
- `npm publish` publishes packages to registries: It either creates a new package or updates an existing package. It is explained in [content not included].
- `npm run` (which is short for `npm run-script`) executes package scripts. Package scripts are explained in [content not included].
- `npm uninstall` removes a package that was installed globally or locally.
- `npm version` prints the object `process.versions` which records the versions of various components of Node.js and npm:

```
{
  'my-package': '1.0.0', // current package
  npm: '8.15.0',
  node: '18.7.0',
  v8: '10.2.154.13-node.9',
  uv: '1.43.0', // libuv
  ...
  tz: '2022a', // version of tz database
```

```
    unicode: '14.0', // version of Unicode standard
    ...
}
```

- `npx` lets us run bin scripts in packages without installing them. It is described in [content not included].

The npm documentation has a list of all npm commands.

6.4 Abbreviations for npm commands

Many npm commands have abbreviations – for example:

Short	Long
`npm i`	`npm install`
`npm rm`	`npm uninstall`
`npm run`	`npm run-script`

For each npm command it describes, the npm documentation also lists all of its aliases (including abbreviations).

Part III

Core Node.js functionality

Chapter 7

Working with file system paths and file URLs on Node.js

Contents

In this chapter, we learn how to work with file system paths and file URLs on Node.js.

7.1 Path-related functionality on Node.js

In this chapter, we explore path-related functionality on Node.js:

- Most path-related functionality is in module `'node:path'`.
- The global variable `process` has methods for changing the *current working directory* (what that is, is explained soon).
- Module `'node:os'` has functions that return the paths of important directories.

7.1.1 The three ways of accessing the `'node:path'` API

Module `'node:path'` is often imported as follows:

```
import * as path from 'node:path';
```

In this chapter, this import statement is occasionally omitted. We also omit the following import:

```
import * as assert from 'node:assert/strict';
```

We can access Node's path API in three ways:

- We can access platform-specific versions of the API:
  - `path.posix` supports Unixes including macOS.
  - `path.win32` supports Windows.
- `path` itself always supports the current platform. For example, this is a REPL interaction on macOS:

```
> path.parse === path.posix.parse
true
```

Let's see how function `path.parse()`, which parses file system paths, differs for the two platforms:

```
> path.win32.parse(String.raw`C:\Users\jane\file.txt`)
{
  dir: 'C:\\Users\\jane',
  root: 'C:\\',
  base: 'file.txt',
  name: 'file',
  ext: '.txt',
}
> path.posix.parse(String.raw`C:\Users\jane\file.txt`)
{
  dir: '',
  root: '',
  base: 'C:\\Users\\jane\\file.txt',
  name: 'C:\\Users\\jane\\file',
  ext: '.txt',
}
```

We parse a Windows path – first correctly via the `path.win32` API, then via the `path.posix` API. We can see that in the latter case, the path isn't correctly split into its parts – for example, the basename of the file should be `file.txt` (more on what the other properties mean later).

7.2 Foundational path concepts and their API support

7.2.1 Path segments, path separators, path delimiters

Terminology:

- A non-empty path consists of one or more *path segments* – most often names of directories or files.
- A *path separator* is used to separate two adjacent path segments in a path. `path.sep` contains the path separator of the current platform:

```
assert.equal(
  path.posix.sep, '/' // Path separator on Unix
);
assert.equal(
  path.win32.sep, '\\' // Path separator on Windows
);
```

- A *path delimiter* separates elements in lists of paths. `path.delimiter` contains the path delimiter of the current platform:

```
assert.equal(
  path.posix.delimiter, ':' // Path delimiter on Unix
);
assert.equal(
  path.win32.delimiter, ';' // Path delimiter on Windows
);
```

We can see path separators and path delimitors if we examine the PATH shell variable – which contains the paths where the operating system looks for executables when a command is entered in a shell.

This is an example of a macOS PATH (shell variable `$PATH`):

```
> process.env.PATH.split(/(?<=:)/)
[
  '/opt/homebrew/bin:',
  '/opt/homebrew/sbin:',
  '/usr/local/bin:',
  '/usr/bin:',
  '/bin:',
  '/usr/sbin:',
  '/sbin',
]
```

The split separator has a length of zero because the lookbehind assertion `(?<=:)` matches if a given location is preceded by a colon but it does not capture anything. Therefore, the path delimiter `':'` is included in the preceding path.

This is an example of a Windows PATH (shell variable `%Path%`):

```
> process.env.Path.split(/(?<=;)/)
[
  'C:\\Windows\\system32;',
  'C:\\Windows;',
  'C:\\Windows\\System32\\Wbem;',
  'C:\\Windows\\System32\\WindowsPowerShell\\v1.0\\;',
  'C:\\Windows\\System32\\OpenSSH\\;',
  'C:\\ProgramData\\chocolatey\\bin;',
  'C:\\Program Files\\nodejs\\',
]
```

7.2.2 The current working directory

Many shells have the concept of the *current working directory* (CWD) – "the directory I'm currently in":

- If we use a command with a partially qualified path, that path is resolved against the CWD.
- If we omit a path when a command expects a path, the CWD is used.
- On both Unixes and Windows, the command to change the CWD is `cd`.

`process` is a global Node.js variable. It provides us with methods for getting and setting the CWD:

- `process.cwd()` returns the CWD.
- `process.chdir(dirPath)` changes the CWD to `dirPath`.
  - There must be a directory at `dirPath`.
  - That change does not affect the shell, only the currently running Node.js process.

Node.js uses the CWD to fill in missing pieces whenever a path isn't *fully qualified* (complete). That enables us to use partially qualified paths with various functions – e.g. `fs.readFileSync()`.

7.2.2.1 The current working directory on Unix

The following code demonstrates `process.chdir()` and `process.cwd()` on Unix:

```
process.chdir('/home/jane');
assert.equal(
  process.cwd(), '/home/jane'
);
```

7.2.2.2 The current working directory on Windows

So far, we have used the current working directory on Unix. Windows works differently:

- Each drive has a *current directory*.
- There is a *current drive*.

We can use `path.chdir()` to set both at the same time:

```
process.chdir('C:\\Windows');
process.chdir('Z:\\tmp');
```

When we revisit a drive, Node.js remembers the previous current directory of that drive:

```
assert.equal(
  process.cwd(), 'Z:\\tmp'
);
process.chdir('C:');
assert.equal(
  process.cwd(), 'C:\\Windows'
);
```

7.2.3 Fully vs. partially qualified paths, resolving paths

- A *fully qualified path* does not rely on any other information and can be used as is.
- A *partially qualified path* is missing information: We need to turn it into a fully qualified path before we can use it. That is done by *resolving* it against a fully qualified path.

7.2.3.1 Fully and partially qualified paths on Unix

Unix only knows two kinds of paths:

- *Absolute paths* are fully qualified and start with a slash:

  ```
  /home/john/proj
  ```

- *Relative paths* are partially qualified and start with a filename or a dot:

  ```
  .   (current directory)
  ..  (parent directory)
  ```

```
dir
./dir
../dir
../../dir/subdir
```

Let's use `path.resolve()` (which is explained in more detail later) to resolve relative paths against absolute paths. The results are absolute paths:

```
> const abs = '/home/john/proj';

> path.resolve(abs, '.')
'/home/john/proj'
> path.resolve(abs, '..')
'/home/john'
> path.resolve(abs, 'dir')
'/home/john/proj/dir'
> path.resolve(abs, './dir')
'/home/john/proj/dir'
> path.resolve(abs, '../dir')
'/home/john/dir'
> path.resolve(abs, '../../dir/subdir')
'/home/dir/subdir'
```

7.2.3.2 Fully and partially qualified paths on Windows

Windows distinguishes four kinds of paths (for more information, see Microsoft's documentation):

- There are absolute paths and relative paths.
- Each of those two kinds of paths can have a drive letter ("volume designator") or not.

Absolute paths with drive letters are fully qualified. All other paths are partially qualified.

Resolving an absolute path without a drive letter against a fully qualified path `full`, picks up the drive letter of `full`:

```
> const full = 'C:\\Users\\jane\\proj';

> path.resolve(full, '\\Windows')
'C:\\Windows'
```

Resolving a relative path without a drive letter against a fully qualified path, can be viewed as updating the latter:

```
> const full = 'C:\\Users\\jane\\proj';

> path.resolve(full, '.')
'C:\\Users\\jane\\proj'
> path.resolve(full, '..')
'C:\\Users\\jane'
```

```
> path.resolve(full, 'dir')
'C:\\Users\\jane\\proj\\dir'
> path.resolve(full, '.\\dir')
'C:\\Users\\jane\\proj\\dir'
> path.resolve(full, '..\\dir')
'C:\\Users\\jane\\dir'
> path.resolve(full, '..\\..\\dir')
'C:\\Users\\dir'
```

Resolving a relative path `rel` with a drive letter against a fully qualified path `full` depends on the drive letter of `rel`:

- Same drive letter as `full`? Resolve `rel` against `full`.
- Different drive letter than `full`? Resolve `rel` against the current directory of `rel`'s drive.

That looks as follows:

```
// Configure current directories for C: and Z:
process.chdir('C:\\Windows\\System');
process.chdir('Z:\\tmp');

const full = 'C:\\Users\\jane\\proj';

// Same drive letter
assert.equal(
  path.resolve(full, 'C:dir'),
  'C:\\Users\\jane\\proj\\dir'
);
assert.equal(
  path.resolve(full, 'C:'),
  'C:\\Users\\jane\\proj'
);

// Different drive letter
assert.equal(
  path.resolve(full, 'Z:dir'),
  'Z:\\tmp\\dir'
);
assert.equal(
  path.resolve(full, 'Z:'),
  'Z:\\tmp'
);
```

7.3 Getting the paths of standard directories via module `'node:os'`

The module `'node:os'` provides us with the paths of two important directories:

- `os.homedir()` returns the path to the home directory of the current user – for example:

```
> os.homedir() // macOS
'/Users/rauschma'
> os.homedir() // Windows
'C:\\Users\\axel'
```

- `os.tmpdir()` returns the path of the operating system's directory for temporary files – for example:

```
> os.tmpdir() // macOS
'/var/folders/ph/sz0384m11vxf5byk12fzjms40000gn/T'
> os.tmpdir() // Windows
'C:\\Users\\axel\\AppData\\Local\\Temp'
```

7.4 Concatenating paths

There are two functions for concatenating paths:

- `path.resolve()` always returns fully qualified paths
- `path.join()` preserves relative paths

7.4.1 `path.resolve()`: concatenating paths to create fully qualified paths

```
path.resolve(...paths: Array<string>): string
```

Concatenates the `paths` and return a fully qualified path. It uses the following algorithm:

- Start with the current working directory.
- Resolve `path[0]` against the previous result.
- Resolve `path[1]` against the previous result.
- Do the same for all remaining paths.
- Return the final result.

Without arguments, `path.resolve()` returns the path of the current working directory:

```
> process.cwd()
'/usr/local'
> path.resolve()
'/usr/local'
```

One or more relative paths are used for resolution, starting with the current working directory:

```
> path.resolve('.')
'/usr/local'
> path.resolve('..')
'/usr'
> path.resolve('bin')
'/usr/local/bin'
```

```
> path.resolve('./bin', 'sub')
'/usr/local/bin/sub'
> path.resolve('../lib', 'log')
'/usr/lib/log'
```

Any fully qualified path replaces the previous result:

```
> path.resolve('bin', '/home')
'/home'
```

That enables us to resolve partially qualified paths against fully qualified paths:

```
> path.resolve('/home/john', 'proj', 'src')
'/home/john/proj/src'
```

7.4.2 `path.join()`: concatenating paths while preserving relative paths

```
path.join(...paths: Array<string>): string
```

Starts with `paths[0]` and interprets the remaining paths as instructions for ascending or descending. In contrast to `path.resolve()`, this function preserves partially qualified paths: If `paths[0]` is partially qualified, the result is partially qualified. If it is fully qualified, the result is fully qualified.

Examples of descending:

```
> path.posix.join('/usr/local', 'sub', 'subsub')
'/usr/local/sub/subsub'
> path.posix.join('relative/dir', 'sub', 'subsub')
'relative/dir/sub/subsub'
```

Double dots ascend:

```
> path.posix.join('/usr/local', '..')
'/usr'
> path.posix.join('relative/dir', '..')
'relative'
```

Single dots do nothing:

```
> path.posix.join('/usr/local', '.')
'/usr/local'
> path.posix.join('relative/dir', '.')
'relative/dir'
```

If arguments after the first one are fully qualified paths, they are interpreted as relative paths:

```
> path.posix.join('dir', '/tmp')
'dir/tmp'
> path.win32.join('dir', 'C:\\Users')
'dir\\C:\\Users'
```

Using more than two arguments:

```
> path.posix.join('/usr/local', '../lib', '.', 'log')
'/usr/lib/log'
```

7.5 Ensuring paths are normalized, fully qualified, or relative

7.5.1 `path.normalize()`: ensuring paths are normalized

```
path.normalize(path: string): string
```

On Unix, `path.normalize()`:

- Removes path segments that are single dots (`.`).
- Resolves path segments that are double dots (`..`).
- Turns multiple path separators into a single path separator.

For example:

```
// Fully qualified path
assert.equal(
  path.posix.normalize('/home/./john/lib/../photos///pet'),
  '/home/john/photos/pet'
);

// Partially qualified path
assert.equal(
  path.posix.normalize('./john/lib/../photos///pet'),
  'john/photos/pet'
);
```

On Windows, `path.normalize()`:

- Removes path segments that are single dots (`.`).
- Resolves path segments that are double dots (`..`).
- Converts each path separator slash (`/`) – which is legal – into a the preferred path separator (`\`).
- Converts sequences of more than one path separator to single backslashes.

For example:

```
// Fully qualified path
assert.equal(
  path.win32.normalize('C:\\Users/jane\\doc\\..\\proj\\\\src'),
  'C:\\Users\\jane\\proj\\src'
);

// Partially qualified path
assert.equal(
  path.win32.normalize('.\\jane\\doc\\..\\proj\\\\src'),
  'jane\\proj\\src'
);
```

Note that `path.join()` with a single argument also normalizes and works the same as `path.normalize()`:

```
> path.posix.normalize('/home/./john/lib/../photos///pet')
'/home/john/photos/pet'
> path.posix.join('/home/./john/lib/../photos///pet')
'/home/john/photos/pet'

> path.posix.normalize('./john/lib/../photos///pet')
'john/photos/pet'
> path.posix.join('./john/lib/../photos///pet')
'john/photos/pet'
```

7.5.2 `path.resolve()` (one argument): ensuring paths are normalized and fully qualified

We have already encountered `path.resolve()`. Called with a single argument, it both normalizes paths and ensures that they are fully qualified.

Using `path.resolve()` on Unix:

```
> process.cwd()
'/usr/local'

> path.resolve('/home/./john/lib/../photos///pet')
'/home/john/photos/pet'
> path.resolve('./john/lib/../photos///pet')
'/usr/local/john/photos/pet'
```

Using `path.resolve()` on Windows:

```
> process.cwd()
'C:\\Windows\\System'

> path.resolve('C:\\Users/jane\\doc\\..\\proj\\\\src')
'C:\\Users\\jane\\proj\\src'
> path.resolve('.\\jane\\doc\\..\\proj\\\\src')
'C:\\Windows\\System\\jane\\proj\\src'
```

7.5.3 `path.relative()`: creating relative paths

```
path.relative(sourcePath: string, destinationPath: string): string
```

Returns a relative path that gets us from `sourcePath` to `destinationPath`:

```
> path.posix.relative('/home/john/', '/home/john/proj/my-lib/README.md')
'proj/my-lib/README.md'
> path.posix.relative('/tmp/proj/my-lib/', '/tmp/doc/zsh.txt')
'../../doc/zsh.txt'
```

On Windows, we get a fully qualified path if `sourcePath` and `destinationPath` are on different drives:

```
> path.win32.relative('Z:\\tmp\\', 'C:\\Users\\Jane\\')
'C:\\Users\\Jane'
```

This function also works with relative paths:

```
> path.posix.relative('proj/my-lib/', 'doc/zsh.txt')
'../../doc/zsh.txt'
```

7.6 Parsing paths: extracting various parts of a path (filename extension etc.)

7.6.1 `path.parse()`: creating an object with path parts

```
type PathObject = {
  dir: string,
    root: string,
  base: string,
    name: string,
    ext: string,
};
path.parse(path: string): PathObject
```

Extracts various parts of `path` and returns them in an object with the following properties:

- `.base`: last segment of a path
  - `.ext`: the filename extension of the base
  - `.name`: the base without the extension. This part is also called the *stem* of a path.
- `.root`: the beginning of a path (before the first segment)
- `.dir`: the directory in which the base is located – the path without the base

Later, we'll see function `path.format()` which is the inverse of `path.parse()`: It converts an object with path parts into a path.

7.6.1.1 Example: `path.parse()` on Unix

This is what using `path.parse()` on Unix looks like:

```
> path.posix.parse('/home/jane/file.txt')
{
  dir: '/home/jane',
  root: '/',
  base: 'file.txt',
  name: 'file',
  ext: '.txt',
}
```

The following diagram visualizes the extent of the parts:

```
  /      home/jane / file   .txt
| root |           | name | ext  |
```

```
| dir              | base        |
```

For example, we can see that `.dir` is the path without the base. And that `.base` is `.name` plus `.ext`.

7.6.1.2 Example: `path.parse()` on Windows

This is how `path.parse()` works on Windows:

```
> path.win32.parse(String.raw`C:\Users\john\file.txt`)
{
  dir: 'C:\\Users\\john',
  root: 'C:\\',
  base: 'file.txt',
  name: 'file',
  ext: '.txt',
}
```

This is a diagram for the result:

```
  C:\    Users\john \ file   .txt
| root |            | name | ext  |
| dir               | base        |
```

7.6.2 `path.basename()`: extracting the base of a path

```
path.basename(path, ext?)
```

Returns the base of `path`:

```
> path.basename('/home/jane/file.txt')
'file.txt'
```

Optionally, this function can also remove a suffix:

```
> path.basename('/home/jane/file.txt', '.txt')
'file'
> path.basename('/home/jane/file.txt', 'txt')
'file.'
> path.basename('/home/jane/file.txt', 'xt')
'file.t'
```

Removing the extension is case sensitive – even on Windows!

```
> path.win32.basename(String.raw`C:\Users\john\file.txt`, '.txt')
'file'
> path.win32.basename(String.raw`C:\Users\john\file.txt`, '.TXT')
'file.txt'
```

7.6.3 `path.dirname()`: extracting the parent directory of a path

```
path.dirname(path)
```

Returns the parent directory of the file or directory at `path`:

```
> path.win32.dirname(String.raw`C:\Users\john\file.txt`)
'C:\\Users\\john'
> path.win32.dirname('C:\\Users\\john\\dir\\')
'C:\\Users\\john'

> path.posix.dirname('/home/jane/file.txt')
'/home/jane'
> path.posix.dirname('/home/jane/dir/')
'/home/jane'
```

7.6.4 `path.extname()`: extracting the extension of a path

```
path.extname(path)
```

Returns the extension of `path`:

```
> path.extname('/home/jane/file.txt')
'.txt'
> path.extname('/home/jane/file.')
'.'
> path.extname('/home/jane/file')
''
> path.extname('/home/jane/')
''
> path.extname('/home/jane')
''
```

7.7 Categorizing paths

7.7.1 `path.isAbsolute()`: Is a given path absolute?

```
path.isAbsolute(path: string): boolean
```

Returns `true` if `path` is absolute and `false` otherwise.

The results on Unix are straightforward:

```
> path.posix.isAbsolute('/home/john')
true
> path.posix.isAbsolute('john')
false
```

On Windows, "absolute" does not necessarily mean "fully qualified" (only the first path is fully qualified):

```
> path.win32.isAbsolute('C:\\Users\\jane')
true
> path.win32.isAbsolute('\\Users\\jane')
true
> path.win32.isAbsolute('C:jane')
false
```

```
> path.win32.isAbsolute('jane')
false
```

7.8 `path.format()`: creating paths out of parts

```
type PathObject = {
  dir: string,
    root: string,
  base: string,
    name: string,
    ext: string,
};
path.format(pathObject: PathObject): string
```

Creates a path out of a path object:

```
> path.format({dir: '/home/jane', base: 'file.txt'})
'/home/jane/file.txt'
```

7.8.1 Example: changing the filename extension

We can use `path.format()` to change the extension of a path:

```
function changeFilenameExtension(pathStr, newExtension) {
  if (!newExtension.startsWith('.')) {
    throw new Error(
      'Extension must start with a dot: '
      + JSON.stringify(newExtension)
    );
  }
  const parts = path.parse(pathStr);
  return path.format({
    ...parts,
    base: undefined, // prevent .base from overriding .name and .ext
    ext: newExtension,
  });
}

assert.equal(
  changeFilenameExtension('/tmp/file.md', '.html'),
  '/tmp/file.html'
);
assert.equal(
  changeFilenameExtension('/tmp/file', '.html'),
  '/tmp/file.html'
);
assert.equal(
  changeFilenameExtension('/tmp/file/', '.html'),
```

```
    '/tmp/file.html'
);
```

If we know the original filename extension, we can also use a regular expression to change the filename extension:

```
> '/tmp/file.md'.replace(/\.md$/i, '.html')
'/tmp/file.html'
> '/tmp/file.MD'.replace(/\.md$/i, '.html')
'/tmp/file.html'
```

7.9 Using the same paths on different platforms

Sometimes we'd like to use the same paths on different platforms. Then there are two issues that we are facing:

- The path separator may be different.
- The file structure may be different: home directories and directories for temporary files may be in different locations, etc.

As an example, consider a Node.js app that operates on a directory with data. Let's assume that the app can be configured with two kinds of paths:

- Fully qualified paths anywhere on the system
- Paths inside the data directory

Due to the aforementioned issues:

- We can't reuse fully qualified paths between platforms.
  - Sometimes we need absolute paths. These have to be configured per "instance" of the data directory and stored externally (or inside it and ignored by version control). These paths stay put and are not moved with the data directory.
- We can reuse paths that point into the data directory. Such paths may be stored in configuration files (inside the data directory or not) and in constants in the app's code. To do that:
  - We have to store them as relative paths.
  - We have to ensure that the path separator is correct on each platform.

  The next subsection explains how both can be achieved.

7.9.1 Relative platform-independent paths

Relative platform-independent paths can be stored as Arrays of path segments and turned into fully qualified platform-specific paths as follows:

```
const universalRelativePath = ['static', 'img', 'logo.jpg'];

const dataDirUnix = '/home/john/data-dir';
assert.equal(
```

```
  path.posix.resolve(dataDirUnix, ...universalRelativePath),
  '/home/john/data-dir/static/img/logo.jpg'
);

const dataDirWindows = 'C:\\Users\\jane\\data-dir';
assert.equal(
  path.win32.resolve(dataDirWindows, ...universalRelativePath),
  'C:\\Users\\jane\\data-dir\\static\\img\\logo.jpg'
);
```

To create relative platform-specific paths, we can use:

```
const dataDir = '/home/john/data-dir';
const pathInDataDir = '/home/john/data-dir/static/img/logo.jpg';
assert.equal(
  path.relative(dataDir, pathInDataDir),
  'static/img/logo.jpg'
);
```

The following function converts relative platform-specific paths into platform-independent paths:

```
import * as path from 'node:path';

function splitRelativePathIntoSegments(relPath) {
  if (path.isAbsolute(relPath)) {
    throw new Error('Path isn't relative: ' + relPath);
  }
  relPath = path.normalize(relPath);
  const result = [];
  while (true) {
    const base = path.basename(relPath);
    if (base.length === 0) break;
    result.unshift(base);
    const dir = path.dirname(relPath);
    if (dir === '.') break;
    relPath = dir;
  }
  return result;
}
```

Using `splitRelativePathIntoSegments()` on Unix:

```
> splitRelativePathIntoSegments('static/img/logo.jpg')
[ 'static', 'img', 'logo.jpg' ]
> splitRelativePathIntoSegments('file.txt')
[ 'file.txt' ]
```

Using `splitRelativePathIntoSegments()` on Windows:

```
> splitRelativePathIntoSegments('static/img/logo.jpg')
[ 'static', 'img', 'logo.jpg' ]
```

```
> splitRelativePathIntoSegments('C:static/img/logo.jpg')
[ 'static', 'img', 'logo.jpg' ]

> splitRelativePathIntoSegments('file.txt')
[ 'file.txt' ]
> splitRelativePathIntoSegments('C:file.txt')
[ 'file.txt' ]
```

7.10 Using a library to match paths via *globs*

The npm module `'minimatch'` lets us match paths against patterns that are called *glob expressions*, *glob patterns*, or *globs*:

```
import minimatch from 'minimatch';
assert.equal(
  minimatch('/dir/sub/file.txt', '/dir/sub/*.txt'), true
);
assert.equal(
  minimatch('/dir/sub/file.txt', '/**/file.txt'), true
);
```

Use cases for globs:

- Specifying which files in a directory should be processed by a script.
- Specifying which files to ignore.

More glob libraries:

- multimatch extends minimatch with support for multiple patterns.
- micromatch is an alternative to minimatch and multimatch that has a similar API.
- globby is a library based on fast-glob that adds convenience features.

7.10.1 The minimatch API

The whole API of minimatch is documented in the project's readme file. In this subsection, we look at the most important functionality.

Minimatch compiles globs to JavaScript `RegExp` objects and uses those to match.

7.10.1.1 `minimatch()`: compiling and matching once

```
minimatch(path: string, glob: string, options?: MinimatchOptions): boolean
```

Returns `true` if `glob` matches `path` and `false` otherwise.

Two interesting options:

- `.dot: boolean` (default: `false`)
  If `true`, wildcard symbols such as `*` and `**` match "invisible" path segments (whose names begin with dots):

  ```
  > minimatch('/usr/local/.tmp/data.json', '/usr/**/data.json')
  false
  ```

```
> minimatch('/usr/local/.tmp/data.json', '/usr/**/data.json', {dot: true})
true

> minimatch('/tmp/.log/events.txt', '/tmp/*/events.txt')
false
> minimatch('/tmp/.log/events.txt', '/tmp/*/events.txt', {dot: true})
true
```

- `.matchBase: boolean` (default: `false`)
  If `true`, a pattern without slashes is matched against the basename of a path:

```
> minimatch('/dir/file.txt', 'file.txt')
false
> minimatch('/dir/file.txt', 'file.txt', {matchBase: true})
true
```

7.10.1.2 `new minimatch.Minimatch()`: compiling once, matching multiple times

Class `minimatch.Minimatch` enables us to only compile the glob to a regular expression once and match multiple times:

```
new Minimatch(pattern: string, options?: MinimatchOptions)
```

This is how this class is used:

```
import minimatch from 'minimatch';
const {Minimatch} = minimatch;
const glob = new Minimatch('/dir/sub/*.txt');
assert.equal(
  glob.match('/dir/sub/file.txt'), true
);
assert.equal(
  glob.match('/dir/sub/notes.txt'), true
);
```

7.10.2 Syntax of glob expressions

This subsection covers the essentials of the syntax. But there are more features. These are documented here:

- Minimatch's unit tests have many examples of globs.
- The Bash Reference manual has a section on filename expansion.

7.10.2.1 Matching Windows paths

Even on Windows, glob segments are separated by slashes – but they match both backslashes and slashes (which are legal path separators on Windows):

```
> minimatch('dir\\sub/file.txt', 'dir/sub/file.txt')
true
```

7.10.2.2 Minimatch does not normalize paths

Minimatch does not normalize paths for us:

```
> minimatch('./file.txt', './file.txt')
true
> minimatch('./file.txt', 'file.txt')
false
> minimatch('file.txt', './file.txt')
false
```

Therefore, we have to normalize paths if we don't create them ourselves:

```
> path.normalize('./file.txt')
'file.txt'
```

7.10.2.3 Patterns without wildcard symbols: path separators must line up

Patterns without *wildcard symbols* (that match more flexibly) must match exactly. Especially the path separators must line up:

```
> minimatch('/dir/file.txt', '/dir/file.txt')
true
> minimatch('dir/file.txt', 'dir/file.txt')
true
> minimatch('/dir/file.txt', 'dir/file.txt')
false

> minimatch('/dir/file.txt', 'file.txt')
false
```

That is, we must decide on either absolute or relative paths.

With option `.matchBase`, we can match patterns without slashes against the basenames of paths:

```
> minimatch('/dir/file.txt', 'file.txt', {matchBase: true})
true
```

7.10.2.4 The asterisk (*) matches any (part of a) single segment

The *wildcard symbol* asterisk (`*`) matches any path segment or any part of a segment:

```
> minimatch('/dir/file.txt', '/*/file.txt')
true
> minimatch('/tmp/file.txt', '/*/file.txt')
true

> minimatch('/dir/file.txt', '/dir/*.txt')
true
> minimatch('/dir/data.txt', '/dir/*.txt')
true
```

The asterisk does not match "invisible files" whose names start with dots. If we want to match those, we have to prefix the asterisk with a dot:

```
> minimatch('file.txt', '*')
true
> minimatch('.gitignore', '*')
false
> minimatch('.gitignore', '.*')
true
> minimatch('/tmp/.log/events.txt', '/tmp/*/events.txt')
false
```

Option `.dot` lets us switch off this behavior:

```
> minimatch('.gitignore', '*', {dot: true})
true
> minimatch('/tmp/.log/events.txt', '/tmp/*/events.txt', {dot: true})
true
```

7.10.2.5 The double asterisk (**) matches zero or more segments

´**/ matches zero or more segments:

```
> minimatch('/file.txt', '/**/file.txt')
true
> minimatch('/dir/file.txt', '/**/file.txt')
true
> minimatch('/dir/sub/file.txt', '/**/file.txt')
true
```

If we want to match relative paths, the pattern still must not start with a path separator:

```
> minimatch('file.txt', '/**/file.txt')
false
```

The double asterisk does not match "invisible" path segments whose names start with dots:

```
> minimatch('/usr/local/.tmp/data.json', '/usr/**/data.json')
false
```

We can switch off that behavior via option `.dot`:

```
> minimatch('/usr/local/.tmp/data.json', '/usr/**/data.json', {dot: true})
true
```

7.10.2.6 Negating globs

If we start a glob with an exclamation mark, it matches if the pattern after the exclamation mark does not match:

```
> minimatch('file.txt', '!**/*.txt')
false
```

```
> minimatch('file.js', '!**/*.txt')
true
```

7.10.2.7 Alternative patterns

Comma-separate patterns inside braces match if one of the patterns matches:

```
> minimatch('file.txt', 'file.{txt,js}')
true
> minimatch('file.js', 'file.{txt,js}')
true
```

7.10.2.8 Ranges of integers

A pair of integers separated by double dots defines a range of integers and matches if any of its elements matches:

```
> minimatch('file1.txt', 'file{1..3}.txt')
true
> minimatch('file2.txt', 'file{1..3}.txt')
true
> minimatch('file3.txt', 'file{1..3}.txt')
true
> minimatch('file4.txt', 'file{1..3}.txt')
false
```

Padding with zeros is supported, too:

```
> minimatch('file1.txt', 'file{01..12}.txt')
false
> minimatch('file01.txt', 'file{01..12}.txt')
true
> minimatch('file02.txt', 'file{01..12}.txt')
true
> minimatch('file12.txt', 'file{01..15}.txt')
true
```

7.11 Using `file:` URLs to refer to files

There are two common ways to refer to files in Node.js:

- Paths in strings
- Instances of `URL` with the protocol `file:`

For example:

```
assert.equal(
  fs.readFileSync(
    '/tmp/data.txt', {encoding: 'utf-8'}),
  'Content'
);
```

```
assert.equal(
  fs.readFileSync(
    new URL('file:///tmp/data.txt'), {encoding: 'utf-8'}),
  'Content'
);
```

7.11.1 Class `URL`

In this section, we take a closer look at class `URL`. More information on this class:

- Node.js documentation: section "The WHATWG URL API"
- Section "API" of the WHATWG URL standard

In this chapter, we access class `URL` via a global variable because that's how it's used on other web platforms. But it can also be imported:

```
import {URL} from 'node:url';
```

7.11.1.1 URIs vs. relative references

URLs are a subset of URIs. RFC 3986, the standard for URIs, distinguishes two kinds of *URI-references*:

- A *URI* starts with a scheme followed by a colon separator.
- All other URI references are *relative references*.

7.11.1.2 Constructor of `URL`

Class `URL` can be instantiated in two ways:

- `new URL(uri: string)`

  `uri` must be a URI. It specifies the URI of the new instance.

- `new URL(uriRef: string, baseUri: string)`

  `baseUri` must be a URI. If `uriRef` is a relative reference, it is resolved against `baseUri` and the result becomes the URI of the new instance.

  If `uriRef` is a URI, it completely replaces `baseUri` as the data on which the instance is based.

Here we can see the class in action:

```
// If there is only one argument, it must be a proper URI
assert.equal(
  new URL('https://example.com/public/page.html').toString(),
  'https://example.com/public/page.html'
);
assert.throws(
  () => new URL('../book/toc.html'),
  /^TypeError \[ERR_INVALID_URL\]: Invalid URL$/
);
```

```
// Resolve a relative reference against a base URI
assert.equal(
  new URL(
    '../book/toc.html',
    'https://example.com/public/page.html'
  ).toString(),
  'https://example.com/book/toc.html'
);
```

7.11.1.3 Resolving relative references against instances of `URL`

Let's revisit this variant of the `URL` constructor:

```
new URL(uriRef: string, baseUri: string)
```

The argument `baseUri` is coerced to string. Therefore, any object can be used – as long as it becomes a valid URL when coereced to string:

```
const obj = { toString() {return 'https://example.com'} };
assert.equal(
  new URL('index.html', obj).href,
  'https://example.com/index.html'
);
```

That enables us to resolve relative references against `URL` instances:

```
const url = new URL('https://example.com/dir/file1.html');
assert.equal(
  new URL('../file2.html', url).href,
  'https://example.com/file2.html'
);
```

Used this way, the constructor is loosely similar to `path.resolve()`.

7.11.1.4 Properties of `URL` instances

Instances of `URL` have the following properties:

```
type URL = {
  protocol: string,
  username: string,
  password: string,
  hostname: string,
  port: string,
  host: string,
  readonly origin: string,

  pathname: string,

  search: string,
  readonly searchParams: URLSearchParams,
  hash: string,
```

```
  href: string,
  toString(): string,
  toJSON(): string,
}
```

7.11.1.5 Converting URLs to strings

There are three common ways in which we can convert URLs to strings:

```
const url = new URL('https://example.com/about.html');

assert.equal(
  url.toString(),
  'https://example.com/about.html'
);
assert.equal(
  url.href,
  'https://example.com/about.html'
);
assert.equal(
  url.toJSON(),
  'https://example.com/about.html'
);
```

Method `.toJSON()` enables us to use URLs in JSON data:

```
const jsonStr = JSON.stringify({
  pageUrl: new URL('https://exploringjs.com')
});
assert.equal(
  jsonStr, '{"pageUrl":"https://exploringjs.com"}'
);
```

7.11.1.6 Getting `URL` properties

The properties of `URL` instances are not own data properties, they are implemented via getters and setters. In the next example, we use the utility function `pickProps()` (whose code is shown at the end), to copy the values returned by those getters into a plain object:

```
const props = pickProps(
  new URL('https://jane:pw@example.com:80/news.html?date=today#misc'),
  'protocol', 'username', 'password', 'hostname', 'port', 'host',
  'origin', 'pathname', 'search', 'hash', 'href'
);
assert.deepEqual(
  props,
  {
    protocol: 'https:',
    username: 'jane',
    password: 'pw',
```

```
    hostname: 'example.com',
    port: '80',
    host: 'example.com:80',
    origin: 'https://example.com:80',
    pathname: '/news.html',
    search: '?date=today',
    hash: '#misc',
    href: 'https://jane:pw@example.com:80/news.html?date=today#misc'
  }
);
function pickProps(input, ...keys) {
  const output = {};
  for (const key of keys) {
    output[key] = input[key];
  }
  return output;
}
```

Alas, the pathname is a single atomic unit. That is, we can't use class `URL` to access its parts (base, extension, etc.).

7.11.1.7 Setting parts of a URL

We can also change parts of a URL by setting properties such as `.hostname`:

```
const url = new URL('https://example.com');
url.hostname = '2ality.com';
assert.equal(
  url.href, 'https://2ality.com/'
);
```

We can use the setters to create URLs from parts (idea by Haroen Viaene):

```
// Object.assign() invokes setters when transferring property values
const urlFromParts = (parts) => Object.assign(
  new URL('https://example.com'), // minimal dummy URL
  parts // assigned to the dummy
);

const url = urlFromParts({
  protocol: 'https:',
  hostname: '2ality.com',
  pathname: '/p/about.html',
});
assert.equal(
  url.href, 'https://2ality.com/p/about.html'
);
```

7.11.1.8 Managing search parameters via `.searchParams`

We can use property `.searchParams` to manage the search parameters of URLs. Its value is an instance of `URLSearchParams`.

We can use it to read search parameters:

```
const url = new URL('https://example.com/?topic=js');
assert.equal(
  url.searchParams.get('topic'), 'js'
);
assert.equal(
  url.searchParams.has('topic'), true
);
```

We can also change search parameters via it:

```
url.searchParams.append('page', '5');
assert.equal(
  url.href, 'https://example.com/?topic=js&page=5'
);

url.searchParams.set('topic', 'css');
assert.equal(
  url.href, 'https://example.com/?topic=css&page=5'
);
```

7.11.2 Converting between URLs and file paths

It's tempting to convert between file paths and URLs manually. For example, we can try to convert an `URL` instance `myUrl` to a file path via `myUrl.pathname`. However that doesn't always work – it's better to use this function:

```
url.fileURLToPath(url: URL | string): string
```

The following code compares the results of that function with the values of `.pathname`:

```
import * as url from 'node:url';

//::::: Unix :::::

const url1 = new URL('file:///tmp/with%20space.txt');
assert.equal(
  url1.pathname, '/tmp/with%20space.txt');
assert.equal(
  url.fileURLToPath(url1), '/tmp/with space.txt');

const url2 = new URL('file:///home/thor/Mj%C3%B6lnir.txt');
assert.equal(
  url2.pathname, '/home/thor/Mj%C3%B6lnir.txt');
assert.equal(
  url.fileURLToPath(url2), '/home/thor/Mjölnir.txt');
```

```
//::::: Windows :::::

const url3 = new URL('file:///C:/dir/');
assert.equal(
  url3.pathname, '/C:/dir/');
assert.equal(
  url.fileURLToPath(url3), 'C:\\dir\\');
```

This function is the inverse of `url.fileURLToPath()`:

```
url.pathToFileURL(path: string): URL
```

It converts `path` to a file URL:

```
> url.pathToFileURL('/home/john/Work Files').href
'file:///home/john/Work%20Files'
```

7.11.3 Use case for URLs: accessing files relative to the current module

One important use case for URLs is accessing a file that is a sibling of the current module:

```
function readData() {
  const url = new URL('data.txt', import.meta.url);
  return fs.readFileSync(url, {encoding: 'UTF-8'});
}
```

This function uses `import.meta.url` which contains the URL of the current module (which is usually a `file:` URL on Node.js).

Using `fetch()` would have made the previous code even more cross-platform. However, as of Node.js 18.9.0, `fetch()` doesn't work for `file:` URLs yet:

```
> await fetch('file:///tmp/file.txt')
TypeError: fetch failed
  cause: Error: not implemented... yet...
```

7.11.4 Use case for URLs: detecting if the current module is "main" (the app entry point)

An ESM module can be used in two ways:

1. It can be used as a library from which other modules can import values.
2. It can be used as script that we run via Node.js – e.g., from a command line. In that case, it is called the *main module*.

If we want a module to be used in both ways, we need a way to check if the current module is the main module because only then do we execute the script functionality. In this chapter, we learn how to perform that check.

7.11.4.1 Determining if a CommonJS module is main

With CommonJS, we can use the following pattern to detect if the current module was the entry point (source: Node.js documentation):

```
if (require.main === module) {
  // Main CommonJS module
}
```

7.11.4.2 Determining if an ESM module is main

As of now, ESM modules have no simple built-in way to check if a module is main. Instead, we have to use the following workaround (based on a tweet by Rich Harris):

```
import * as url from 'node:url';

if (import.meta.url.startsWith('file:')) { // (A)
  const modulePath = url.fileURLToPath(import.meta.url);
  if (process.argv[1] === modulePath) { // (B)
    // Main ESM module
  }
}
```

Explanations:

- `import.meta.url` contains the URL of the currently executed ESM module.
- If we are sure our code always runs locally (which may become less common in the future), we can omit the check in line A. If we do and the code does not run locally, at least we get an exception (and not a silent failure) – thanks to `url.fileURLToPath()` (see next item).
- We use `url.fileURLToPath()` to convert the URL to a local path. This function throws an exception if the protocol isn't `file:`.
- `process.argv[1]` contains the path of the initial module. The comparison in line B works because this value is always an absolute path – Node.js sets it up as follows (source code):

  ```
  process.argv[1] = path.resolve(process.argv[1]);
  ```

7.11.5 Paths vs. `file:` URLs

When shell scripts receive references to files or export references to files (e.g. by logging them on screen), they are virtually always paths. However, there are two cases where we need URLs (as discussed in previous subsections):

- To access files relative to the current module
- To detect if the current module is running as a script

Chapter 8

Working with the file system on Node.js

Contents

This chapter contains:

- An overview of the different parts of Node's file system APIs.
- *Recipes* (code snippets) for performing various tasks via those APIs.

Given that the focus of this book is on shell scripting, we only work with textual data.

8.1 Concepts, patterns and conventions of Node's file system APIs

8.1.1 Ways of accessing files

1. We can read or write the whole content of a file via a string.
2. We can open a stream for reading or a stream for writing and process a file in smaller pieces, one at a time. Streams only allow sequential access.
3. We can use file descriptors or FileHandles and get both sequential and random access, via an API that is loosely similar to streams.
   - *File descriptors* are integer numbers that represent files. They are managed via these functions (only the synchronous names are shown, there are also callback-based versions – `fs.open()` etc.):
     - `fs.openSync(path, flags?, mode?)` opens a new file descriptor for a file at a given path and returns it.
     - `fs.closeSync(fd)` closes a file descriptor.
     - `fs.fchmodSync(fd, mode)`
     - `fs.fchownSync(fd, uid, gid)`
     - `fs.fdatasyncSync(fd)`
     - `fs.fstatSync(fd, options?)`
     - `fs.fsyncSync(fd)`
     - `fs.ftruncateSync(fd, len?)`
     - `fs.futimesSync(fd, atime, mtime)`
   - Only the synchronous API and the callback-based API use file descriptors. The Promise-based API has a better abstraction, class `FileHandle`, which is based on file descriptors. Instances are created via `fsPromises.open()`. Various operations are provided via methods (not via functions):
     - `fileHandle.close()`
     - `fileHandle.chmod(mode)`

  - `fileHandle.chown(uid, gid)`
  - Etc.

Note that we don't use (3) in this chapter – (1) and (2) are enough for our purposes.

8.1.2 Function name prefixes

8.1.2.1 Prefix "l": symbolic links

Functions whose names start with an "l" usually operate on symbolic links:

- `fs.lchmodSync()`, `fs.lchmod()`, `fsPromises.lchmod()`
- `fs.lchownSync()`, `fs.lchown()`, `fsPromises.lchown()`
- `fs.lutimesSync()`, `fs.lutimes()`, `fsPromises.lutimes()`
- Etc.

8.1.2.2 Prefix "f": file descriptors

Functions whose names start with an "f" usually manage file descriptors:

- `fs.fchmodSync()`, `fs.fchmod()`
- `fs.fchownSync()`, `fs.fchown()`
- `fs.fstatSync()`, `fs.fstat()`
- Etc.

8.1.3 Important classes

Several classes play important roles in Node's file system APIs.

8.1.3.1 URLs: an alternative to file system paths in strings

Whenever a Node.js function accepts a file system path in a string (line A), it usually also accepts an instance of `URL` (line B):

```
assert.equal(
  fs.readFileSync(
    '/tmp/text-file.txt', {encoding: 'utf-8'}), // (A)
  'Text content'
);
assert.equal(
  fs.readFileSync(
    new URL('file:///tmp/text-file.txt'), {encoding: 'utf-8'}), // (B)
  'Text content'
);
```

Manually converting between paths and `file:` URLs seems easy but has surprisingly many pitfalls: percent encoding or decoding, Windows drive letters, etc. Instead, it's better to use the following two functions:

- `url.pathToFileURL()`
- `url.fileURLToPath()`

We don't use file URLs in this chapter. Use cases for them are described in §7.11.1 "Class URL".

8.1.3.2 Buffers

Class `Buffer` represents fixed-length byte sequences on Node.js. It is a subclass of `Uint8Array` (a TypedArray). Buffers are mostly used when working with binary files and therefore of less interest in this book.

Whenever Node.js accepts a Buffer, it also accepts a Uint8Array. Thus, given that Uint8Arrays are cross-platform and Buffers aren't, the former is preferable.

Buffers can do one thing that Uint8Arrays can't: encoding and decoding text in various encodings. If we need to encode or decode UTF-8 in Uint8Arrays, we can use class `TextEncoder` or class `TextDecoder`. These classes are available on most JavaScript platforms:

```
> new TextEncoder().encode('café')
Uint8Array.of(99, 97, 102, 195, 169)
> new TextDecoder().decode(Uint8Array.of(99, 97, 102, 195, 169))
'café'
```

8.1.3.3 Node.js streams

Some functions accept or return native Node.js streams:

- `stream.Readable` is Node's class for readable streams. Module `node:fs` uses `fs.ReadStream` which is a subclass.
- `stream.Writable` is Node's class for writable streams. Module `node:fs` uses `fs.WriteStream` which is a subclass.

Instead of native streams, we can now use cross-platform *web streams* on Node.js. How is explained in §10 "Using web streams on Node.js".

8.2 Reading and writing files

8.2.1 Reading a file synchronously into a single string (optional: splitting into lines)

`fs.readFileSync(filePath, options?)` reads the file at `filePath` into a single string:

```
assert.equal(
  fs.readFileSync('text-file.txt', {encoding: 'utf-8'}),
  'there\r\nare\nmultiple\nlines'
);
```

Pros and cons of this approach (vs. using a stream):

- Pro: Easy to use and synchronous. Good enough for many use cases.
- Con: Not a good choice for large files.
  - Before we can process the data, we have to read it in its entirety.

Next, we'll look into spliting the string we have read into lines.

8.2.1.1 Splitting lines without including line terminators

The following code splits a string into lines while removing line terminators. It works with Unix and Windows line terminators:

```
const RE_SPLIT_EOL = /\r?\n/;
function splitLines(str) {
  return str.split(RE_SPLIT_EOL);
}
assert.deepEqual(
  splitLines('there\r\nare\nmultiple\nlines'),
  ['there', 'are', 'multiple', 'lines']
);
```

"EOL" stands for "end of line". We accept both Unix line terminators (`'\n'`) and Windows line terminators (`'\r\n'`, like the first one in the previous example). For more information, see §8.3 "Handling line terminators across platforms".

8.2.1.2 Splitting lines while including line terminators

The following code splits a string into lines while including line terminators. It works with Unix and Windows line terminators ("EOL" stands for "end of line"):

```
const RE_SPLIT_AFTER_EOL = /(?<=\r?\n)/; // (A)
function splitLinesWithEols(str) {
  return str.split(RE_SPLIT_AFTER_EOL);
}

assert.deepEqual(
  splitLinesWithEols('there\r\nare\nmultiple\nlines'),
  ['there\r\n', 'are\n', 'multiple\n', 'lines']
);
assert.deepEqual(
  splitLinesWithEols('first\n\nthird'),
  ['first\n', '\n', 'third']
);
assert.deepEqual(
  splitLinesWithEols('EOL at the end\n'),
  ['EOL at the end\n']
);
assert.deepEqual(
  splitLinesWithEols(''),
  ['']
);
```

Line A contains a regular expression with a lookbehind assertion. It matches at locations that are preceded by a match for the pattern `\r?\n` but it doesn't capture anything. Therefore, it doesn't remove anything between the string fragments that the input string is split into.

On engines that don't support lookbehind assertions (see this table), we can use the fol-

lowing solution:

```
function splitLinesWithEols(str) {
  if (str.length === 0) return [''];
  const lines = [];
  let prevEnd = 0;
  while (prevEnd < str.length) {
    // Searching for '\n' means we'll also find '\r\n'
    const newlineIndex = str.indexOf('\n', prevEnd);
    // If there is a newline, it's included in the line
    const end = newlineIndex < 0 ? str.length : newlineIndex+1;
    lines.push(str.slice(prevEnd, end));
    prevEnd = end;
  }
  return lines;
}
```

This solution is simple, but more verbose.

In both versions of `splitLinesWithEols()`, we again accept both Unix line terminators (`'\n'`) and Windows line terminators (`'\r\n'`). For more information, see §8.3 "Handling line terminators across platforms".

8.2.2 Reading a file via a stream, line by line

We can also read text files via streams:

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

const nodeReadable = fs.createReadStream(
  'text-file.txt', {encoding: 'utf-8'});
const webReadableStream = Readable.toWeb(nodeReadable);
const lineStream = webReadableStream.pipeThrough(
  new ChunksToLinesStream());
for await (const line of lineStream) {
  console.log(line);
}

// Output:
// 'there\r\n'
// 'are\n'
// 'multiple\n'
// 'lines'
```

We used the following external functionality:

- `fs.createReadStream(filePath, options?)` creates a Node.js stream (an instance of `stream.Readable`).
- `stream.Readable.toWeb(streamReadable)` converts a readable Node.js stream to a web stream (an instance of `ReadableStream`).

- The TransformStream class `ChunksToLinesStream` is explained in §10.7.1 "Example: transforming a stream of arbitrary chunks to a stream of lines". *Chunks* are the pieces of data produced by streams. If we have a stream whose chunks are strings with arbitrary lengths and pipe it through a ChunksToLinesStream, then we get a stream whose chunks are lines.

Web streams are asynchronously iterable, which is why we can use a `for-await-of` loop to iterate over lines.

If we are not interested in text lines, then we don't need `ChunksToLinesStream`, can iterate over `webReadableStream` and get chunks with arbitrary lengths.

More information:

- Web streams are covered in §10 "Using web streams on Node.js".
- Line terminators are covered in §8.3 "Handling line terminators across platforms".

Pros and cons of this approach (vs. reading a single string):

- Pro: Works well with large files.
  - We can process the data incrementally, in smaller pieces and don't have to wait for everything to be read.
- Con: More complicated to use and not synchronous.

8.2.3 Writing a single string to a file synchronously

`fs.writeFileSync(filePath, str, options?)` writes `str` to a file at `filePath`. If a file already exists at that path, it is overwritten.

The following code shows how to use this function:

```
fs.writeFileSync(
  'new-file.txt',
  'First line\nSecond line\n',
  {encoding: 'utf-8'}
);
```

For information on line terminators, see §8.3 "Handling line terminators across platforms".

Pros and cons (vs. using a stream):

- Pro: Easy to use and synchronous. Works for many use cases.
- Con: Not suited for large files.

8.2.4 Appending a single string to a file (synchronously)

The following code appends a line of text to an existing file:

```
fs.appendFileSync(
  'existing-file.txt',
  'Appended line\n',
  {encoding: 'utf-8'}
);
```

We can also use `fs.writeFileSync()` to perform this task:

```
fs.writeFileSync(
  'existing-file.txt',
  'Appended line\n',
  {encoding: 'utf-8', flag: 'a'}
);
```

This code is almost the same as the one we used to overwrite existing content (see the previous section for more information). The only difference is that we added the option `.flag`: The value `'a'` means that we append data. Other possible values (e.g. to throw an error if a file doesn't exist yet) are explained in the Node.js documentation.

Watch out: In some functions, this option is named `.flag`, in others `.flags`.

8.2.5 Writing multiple strings to a file via stream

The following code uses a stream to write multiple strings to a file:

```
import {Writable} from 'node:stream';

const nodeWritable = fs.createWriteStream(
  'new-file.txt', {encoding: 'utf-8'});
const webWritableStream = Writable.toWeb(nodeWritable);

const writer = webWritableStream.getWriter();
try {
  await writer.write('First line\n');
  await writer.write('Second line\n');
  await writer.close();
} finally {
  writer.releaseLock()
}
```

We used the following functions:

- `fs.createWriteStream(path, options?)` creates a Node.js stream (an instance of `stream.Writable`).
- `stream.Writable.toWeb(streamWritable)` converts a writable Node.js stream to a web stream (an instance of `WritableStream`).

More information:

- WritableStreams and Writers are covered in §10 "Using web streams on Node.js".
- Line terminators are covered in §8.3 "Handling line terminators across platforms".

Pros and cons (vs. writing a single string):

- Pro: Works well with large files because we can write the data incrementally, in smaller pieces.
- Con: More complicated to use and not synchronous.

8.2.6 Appending multiple strings to a file via a stream (asynchronously)

The following code uses a stream to append text to an existing file:

```
import {Writable} from 'node:stream';

const nodeWritable = fs.createWriteStream(
  'existing-file.txt', {encoding: 'utf-8', flags: 'a'});
const webWritableStream = Writable.toWeb(nodeWritable);

const writer = webWritableStream.getWriter();
try {
  await writer.write('First appended line\n');
  await writer.write('Second appended line\n');
  await writer.close();
} finally {
  writer.releaseLock()
}
```

This code is almost the same as the one we used to overwrite existing content (see the previous section for more information). The only difference is that we added the option `.flags`: The value `'a'` means that we append data. Other possible values (e.g. to throw an error if a file doesn't exist yet) are explained in the Node.js documentation.

Watch out: In some functions, this option is named `.flag`, in others `.flags`.

8.3 Handling line terminators across platforms

Alas, not all platform have the same *line terminator* characters that mark the *end of line* (EOL):

- On Windows, EOL is `'\r\n'`.
- On Unix (incl. macOS), EOL is `'\n'`.

To handle EOL in a manner that works on all platforms, we can use several strategies.

8.3.1 Reading line terminators

When reading text, it's best to recognize both EOLs.

What might that look like when splitting a text into lines? We can include the EOLs (in either format) at the ends. That enables us to change as little as possible if we modify those lines and write them to a file.

When processing lines with EOLs, it's sometimes useful to remove them – e.g. via the following function:

```
const RE_EOL_REMOVE = /\r?\n$/;
function removeEol(line) {
  const match = RE_EOL_REMOVE.exec(line);
  if (!match) return line;
```

```
  return line.slice(0, match.index);
}

assert.equal(
  removeEol('Windows EOL\r\n'),
  'Windows EOL'
);
assert.equal(
  removeEol('Unix EOL\n'),
  'Unix EOL'
);
assert.equal(
  removeEol('No EOL'),
  'No EOL'
);
```

8.3.2 Writing line terminators

When it comes to writing line terminators, we have two options:

- Constant `EOL` in module `'node:os'` contains the EOL of the current platform.
- We can detect the EOL format of an input file and use that when we change that file.

8.4 Traversing and creating directories

8.4.1 Traversing a directory

The following function traverses a directory and lists all of its descendants (its children, the children of its children, etc.):

```
import * as path from 'node:path';

function* traverseDirectory(dirPath) {
  const dirEntries = fs.readdirSync(dirPath, {withFileTypes: true});
  // Sort the entries to keep things more deterministic
  dirEntries.sort(
    (a, b) => a.name.localeCompare(b.name, 'en')
  );
  for (const dirEntry of dirEntries) {
    const fileName = dirEntry.name;
    const pathName = path.join(dirPath, fileName);
    yield pathName;
    if (dirEntry.isDirectory()) {
      yield* traverseDirectory(pathName);
    }
  }
}
```

We used this functionality:

- `fs.readdirSync(thePath, options?)` returns the children of the directory at `thePath`.
  - If option `.withFileTypes` is `true`, the function returns *directory entries*, instances of `fs.Dirent`. These have properties such as:
    * `dirent.name`
    * `dirent.isDirectory()`
    * `dirent.isFile()`
    * `dirent.isSymbolicLink()`
  - If option `.withFileTypes` is `false` or missing, the function returns strings with file names.

The following code shows `traverseDirectory()` in action:

```
for (const filePath of traverseDirectory('dir')) {
  console.log(filePath);
}

// Output:
// 'dir/dir-file.txt'
// 'dir/subdir'
// 'dir/subdir/subdir-file1.txt'
// 'dir/subdir/subdir-file2.csv'
```

8.4.2 Creating a directory (`mkdir`, `mkdir -p`)

We can use the following function to create directories:

```
fs.mkdirSync(thePath, options?): undefined | string
```

`options.recursive` determines how the function creates the directory at `thePath`:

- If `.recursive` is missing or `false`, `mkdirSync()` returns `undefined` and an exception is thrown if:
  - A directory (or file) already exists at `thePath`.
  - The parent directory of `thePath` does not exist.
- If `.recursive` is `true`:
  - It's OK if there is already a directory at `thePath`.
  - The ancestor directories of `thePath` are created as needed.
  - `mkdirSync()` returns the path of the first newly created directory.

This is `mkdirSync()` in action:

```
assert.deepEqual(
  Array.from(traverseDirectory('.')),
  [
    'dir',
  ]
);
fs.mkdirSync('dir/sub/subsub', {recursive: true});
assert.deepEqual(
```

```
  Array.from(traverseDirectory('.')),
  [
    'dir',
    'dir/sub',
    'dir/sub/subsub',
  ]
);
```

Function `traverseDirectory(dirPath)` lists all descendants of the directory at `dirPath`.

8.4.3 Ensuring that a parent directory exists

If we want to set up a nested file structure on demand, we can't always be sure that the ancestor directories exist when we create a new file. Then the following function helps:

```
import * as path from 'node:path';

function ensureParentDirectory(filePath) {
  const parentDir = path.dirname(filePath);
  if (!fs.existsSync(parentDir)) {
    fs.mkdirSync(parentDir, {recursive: true});
  }
}
```

Here we can see `ensureParentDirectory()` in action (line A):

```
assert.deepEqual(
  Array.from(traverseDirectory('.')),
  [
    'dir',
  ]
);
const filePath = 'dir/sub/subsub/new-file.txt';
ensureParentDirectory(filePath); // (A)
fs.writeFileSync(filePath, 'content', {encoding: 'utf-8'});
assert.deepEqual(
  Array.from(traverseDirectory('.')),
  [
    'dir',
    'dir/sub',
    'dir/sub/subsub',
    'dir/sub/subsub/new-file.txt',
  ]
);
```

8.4.4 Creating a temporary directory

`fs.mkdtempSync(pathPrefix, options?)` creates a temporary directory: It appends 6 random characters to `pathPrefix`, creates a directory at the new path and returns that path.

`pathPrefix` shouldn't end with a capital "X" because some platforms replace trailing Xs with random characters.

If we want to create our temporary directory inside an operating-system-specific global temporary directory, we can use function `os.tmpdir()`:

```
import * as os from 'node:os';
import * as path from 'node:path';

const pathPrefix = path.resolve(os.tmpdir(), 'my-app');
  // e.g. '/var/folders/ph/sz0384m11vxf/T/my-app'

const tmpPath = fs.mkdtempSync(pathPrefix);
  // e.g. '/var/folders/ph/sz0384m11vxf/T/my-app1QXOXP'
```

It's important to note that temporary directories are not automatically removed when a Node.js script terminates. We either have to delete it ourselves or rely on the operating system to periodically clean up its global temporary directory (which it may or may not do).

8.5 Copying, renaming, moving files or directories

8.5.1 Copying files or directories

`fs.cpSync(srcPath, destPath, options?)`: copies a file or directory from `srcPath` to `destPath`. Interesting options:

- `.recursive` (default: `false`): Directories (including empty ones) are only copied if this option is `true`.
- `.force` (default: `true`): If `true`, existing files are overwritten. If `false`, existing files are preserved.
  - In the latter case, setting `.errorOnExist` to `true` leads to errors being thrown if file paths clash.
- `.filter` is a function that lets us control which files are copied.
- `.preserveTimestamps` (default: `false`): If `true`, the copies in `destPath` get the same timestamps as the originals in `srcPath`.

This is the function in action:

```
assert.deepEqual(
  Array.from(traverseDirectory('.')),
  [
    'dir-orig',
    'dir-orig/some-file.txt',
  ]
);
fs.cpSync('dir-orig', 'dir-copy', {recursive: true});
assert.deepEqual(
  Array.from(traverseDirectory('.')),
  [
    'dir-copy',
```

```
      'dir-copy/some-file.txt',
      'dir-orig',
      'dir-orig/some-file.txt',
    ]
  );
```

Function `traverseDirectory(dirPath)` lists all descendants of the directory at `dirPath`.

8.5.2 Renaming or moving files or directories

`fs.renameSync(oldPath, newPath)` renames or moves a file or a directory from `oldPath` to `newPath`.

Let's use this function to rename a directory:

```
assert.deepEqual(
  Array.from(traverseDirectory('.')),
  [
    'old-dir-name',
    'old-dir-name/some-file.txt',
  ]
);
fs.renameSync('old-dir-name', 'new-dir-name');
assert.deepEqual(
  Array.from(traverseDirectory('.')),
  [
    'new-dir-name',
    'new-dir-name/some-file.txt',
  ]
);
```

Here we use the function to move a file:

```
assert.deepEqual(
  Array.from(traverseDirectory('.')),
  [
    'dir',
    'dir/subdir',
    'dir/subdir/some-file.txt',
  ]
);
fs.renameSync('dir/subdir/some-file.txt', 'some-file.txt');
assert.deepEqual(
  Array.from(traverseDirectory('.')),
  [
    'dir',
    'dir/subdir',
    'some-file.txt',
  ]
);
```

Function `traverseDirectory(dirPath)` lists all descendants of the directory at `dirPath`.

8.6 Removing files or directories

8.6.1 Removing files and arbitrary directories (shell: `rm`, `rm -r`)

`fs.rmSync(thePath, options?)` removes a file or directory at `thePath`. Interesting options:

- `.recursive` (default: `false`): Directories (including empty ones) are only removed if this option is `true`.
- `.force` (default: `false`): If `false`, an exception will be thrown if there is no file or directory at `thePath`.

Let's use `fs.rmSync()` to remove a file:

```
assert.deepEqual(
  Array.from(traverseDirectory('.')),
  [
    'dir',
    'dir/some-file.txt',
  ]
);
fs.rmSync('dir/some-file.txt');
assert.deepEqual(
  Array.from(traverseDirectory('.')),
  [
    'dir',
  ]
);
```

Here we use `fs.rmSync()` to recursively remove a non-empty directory.

```
assert.deepEqual(
  Array.from(traverseDirectory('.')),
  [
    'dir',
    'dir/subdir',
    'dir/subdir/some-file.txt',
  ]
);
fs.rmSync('dir/subdir', {recursive: true});
assert.deepEqual(
  Array.from(traverseDirectory('.')),
  [
    'dir',
  ]
);
```

Function `traverseDirectory(dirPath)` lists all descendants of the directory at `dirPath`.

8.6.2 Removing an empty directory (shell: `rmdir`)

`fs.rmdirSync(thePath, options?)` removes an empty directory (an exception is thrown if a directory isn't empty).

The following code shows how this function works:

```
assert.deepEqual(
  Array.from(traverseDirectory('.')),
  [
    'dir',
    'dir/subdir',
  ]
);
fs.rmdirSync('dir/subdir');
assert.deepEqual(
  Array.from(traverseDirectory('.')),
  [
    'dir',
  ]
);
```

Function `traverseDirectory(dirPath)` lists all descendants of the directory at `dirPath`.

8.6.3 Clearing directories

A script that saves its output to a directory `dir`, often needs to *clear* `dir` before it starts: Remove every file in `dir` so that it is empty. The following function does that.

```
import * as path from 'node:path';

function clearDirectory(dirPath) {
  for (const fileName of fs.readdirSync(dirPath)) {
    const pathName = path.join(dirPath, fileName);
    fs.rmSync(pathName, {recursive: true});
  }
}
```

We used two file system functions:

- `fs.readdirSync(dirPath)` returns the names of all children of the directory at `dirPath`. It is explained in §8.4.1 "Traversing a directory".
- `fs.rmSync(pathName, options?)` removes files and directories (including non-empty ones). It is explained in §8.6.1 "Removing files and arbitrary directories (shell: `rm`, `rm -r`)".

This is an example of using `clearDirectory()`:

```
assert.deepEqual(
  Array.from(traverseDirectory('.')),
  [
    'dir',
```

```
      'dir/dir-file.txt',
      'dir/subdir',
      'dir/subdir/subdir-file.txt'
    ]
  );
  clearDirectory('dir');
  assert.deepEqual(
    Array.from(traverseDirectory('.')),
    [
      'dir',
    ]
  );
```

8.6.4 Trashing files or directories

The library `trash` moves files and folders to the trash. It works on macOS, Windows, and Linux (where support is limited and help is wanted). This is an example from its readme file:

```
import trash from 'trash';

await trash(['*.png', '!rainbow.png']);
```

`trash()` accepts either an Array of strings or a string as its first parameter. Any string can be a glob pattern (with asterisks and other meta-characters).

8.7 Reading and changing file system entries

8.7.1 Checking if a file or directory exists

`fs.existsSync(thePath)` returns `true` if a file or directory exists at `thePath`:

```
assert.deepEqual(
  Array.from(traverseDirectory('.')),
  [
    'dir',
    'dir/some-file.txt',
  ]
);
assert.equal(
  fs.existsSync('dir'), true
);
assert.equal(
  fs.existsSync('dir/some-file.txt'), true
);
assert.equal(
  fs.existsSync('dir/non-existent-file.txt'), false
);
```

Function `traverseDirectory(dirPath)` lists all descendants of the directory at `dirPath`.

8.7.2 Checking the stats of a file: Is it a directory? When was it created? Etc.

`fs.statSync(thePath, options?)` returns an instance of `fs.Stats` with information on the file or directory at `thePath`.

Interesting `options`:

- `.throwIfNoEntry` (default: `true`): What happens if there is no entity at `path`?
  - If this option is `true`, an exception is thrown.
  - If it is `false`, `undefined` is returned.
- `.bigint` (default: `false`): If `true`, this function uses bigints for numeric values (such as timestamps, see below).

Properties of instances of `fs.Stats`:

- What kind of file system entry is it?
  - `stats.isFile()`
  - `stats.isDirectory()`
  - `stats.isSymbolicLink()`
- `stats.size` is the size in bytes
- Timestamps:
  - There are three kinds of timestamps:
    - `stats.atime`: time of last access
    - `stats.mtime`: time of last modification
    - `stats.birthtime`: time of creation
  - Each of these timestamps can be specified with three different units – for example, `atime`:
    - `stats.atime`: instance of `Date`
    - `stats.atimeMS`: milliseconds since the POSIX Epoch
    - `stats.atimeNs`: nanoseconds since the POSIX Epoch (requires option `.bigint`)

In the following example, we use `fs.statSync()` to implement a function `isDirectory()`:

```
function isDirectory(thePath) {
  const stats = fs.statSync(thePath, {throwIfNoEntry: false});
  return stats !== undefined && stats.isDirectory();
}

assert.deepEqual(
  Array.from(traverseDirectory('.')),
  [
    'dir',
    'dir/some-file.txt',
  ]
);

assert.equal(
  isDirectory('dir'), true
```

```
);
assert.equal(
  isDirectory('dir/some-file.txt'), false
);
assert.equal(
  isDirectory('non-existent-dir'), false
);
```

Function `traverseDirectory(dirPath)` lists all descendants of the directory at `dirPath`.

8.7.3 Changing file attributes: permissions, owner, group, timestamps

Let's briefly look at functions for changing file attributes:

- `fs.chmodSync(path, mode)` changes the permission of a file.
- `fs.chownSync(path, uid, gid)` changes the owner and group of a file.
- `fs.utimesSync(path, atime, mtime)` changes the timestamps of a file:
  - `atime`: time of last access
  - `mtime`: time of last modification

8.8 Working with links

Functions for working with hard links:

- `fs.linkSync(existingPath, newPath)` create a hard link.
- `fs.unlinkSync(path)` removes a hard link and possibly the file it points to (if it is the last hard link to that file).

Functions for working with symbolic links:

- `fs.symlinkSync(target, path, type?)` creates a symbolic link from `path` to `target`.
- `fs.readlinkSync(path, options?)` returns the target of the symbolic link at `path`.

The following functions operate on symbolic links without dereferencing them (note the name prefix "l"):

- `fs.lchmodSync(path, mode)` changes the permissions of the symbolic link at `path`.
- `fs.lchownSync(path, uid, gid)` changes user and group of the symbolic link at `path`.
- `fs.lutimesSync(path, atime, mtime)` changes the timestamps of the symbolic link at `path`.
- `fs.lstatSync(path, options?)` returns the stats (timestamps etc.) of the symbolic link at `path`.

Other useful functions:

- `fs.realpathSync(path, options?)` computes the canonical pathname by resolving dots (`.`), double dots (`..`), and symbolic links.

Options of functions that affect how symbolic links are handled:

- `fs.cpSync(src, dest, options?)`:

  - `.dereference` (default: `false`): If `true`, copy the files that symbolic links points to, not the symbolic links themselves.
  - `.verbatimSymlinks` (default: `false`): If `false`, the target of a copied symbolic link will be updated so that it still points to the same location. If `true`, the target won't be changed.

8.9 Further reading

- "JavaScript for impatient programmers" has several chapters on writing asynchronous code:
  - "Foundations of asynchronous programming in JavaScript"
  - "Promises for asynchronous programming"
  - "Async functions"
  - "Asynchronous iteration"

Chapter 9

Native Node.js streams

Contents

This chapter is an introduction to Node's native streams. They support asynchronous iteration which makes them easier to work with and which is what we will mostly use in this chapter.

Note that cross-platform *web streams* are covered in §10 "Using web streams on Node.js". We will mostly use those in this book. Therefore, you can skip the current chapter if you want to.

9.1 Recap: asynchronous iteration and asynchronous generators

Asynchronous iteration is a protocol for retrieving the contents of a data container asynchronously (meaning the current "task" may be paused before retrieving an item).

Asynchronous generators help with async iteration. For example, this is an asynchronous generator function:

```
/**
 * @returns an asynchronous iterable
 */
async function* asyncGenerator(asyncIterable) {
  for await (const item of asyncIterable) { // input
    if (···) {
      yield '> ' + item; // output
    }
  }
}
```

- The `for-await-of` loop iterates over the input `asyncIterable`. This loop is also available in normal asynchronous functions.
- The `yield` feeds values into the asynchronous iterable that is returned by this generator.

In the remainder of the chapter, pay close attention to whether a function is an async function or an async generator function:

```
/** @returns a Promise */
async function asyncFunction() { /*···*/ }

/** @returns an async iterable */
async function* asyncGeneratorFunction() { /*···*/ }
```

9.2 Streams

A stream is a pattern whose core idea is to "divide and conquer" a large amount of data: We can handle it if we split it into smaller pieces and handle one portion at a time.

Node.js supports several kinds of streams – for example:

- *Readable streams* are streams from which we can read data. In other words, they are sources of data. An example is a *readable file stream*, which lets us read the contents of a file.
- *Writable streams* are streams to which we can write data. In other words, they are sinks for data. An example is a *writable file stream*, which lets us write data to a file.
- A *transform stream* is both readable and writable. As a writable stream, it receives pieces of data, *transforms* (changes or discards) them and then outputs them as a readable stream.

9.2.1 Pipelining

To process streamed data in multiple steps, we can *pipeline* (connect) streams:

1. Input is received via a readable stream.
2. Each processing step is performed via a transform stream.
3. For the last processing step, we have two options:
   - We can write the data in the most recent readable stream into a writable stream. That is, the writable stream is the last element of our pipeline.
   - We can process the data in the most recent readable stream in some other manner.

Part (2) is optional.

9.2.2 Text encodings

When creating text streams, it is best to always specify an encoding:

- The Node.js docs have a list of supported encodings and their default spellings – for example:
  - `'utf8'`
  - `'utf16le'`
  - `'base64'`
- A few different spellings are also allowed. You can use `Buffer.isEncoding()` to check which ones are:

```
> buffer.Buffer.isEncoding('utf8')
true
> buffer.Buffer.isEncoding('utf-8')
true
> buffer.Buffer.isEncoding('UTF-8')
true
> buffer.Buffer.isEncoding('UTF:8')
false
```

The default value for encodings is `null`, which is equivalent to `'utf8'`.

9.2.3 Helper function: `readableToString()`

We will occasionally use the following helper function. You don't need to understand how it works, only (roughly) what it does.

```
import * as stream from 'stream';

/**
 * Reads all the text in a readable stream and returns it as a string,
 * via a Promise.
 * @param {stream.Readable} readable
 */
function readableToString(readable) {
  return new Promise((resolve, reject) => {
```

```
    let data = '';
    readable.on('data', function (chunk) {
      data += chunk;
    });
    readable.on('end', function () {
      resolve(data);
    });
    readable.on('error', function (err) {
      reject(err);
    });
  });
}
```

This function is implemented via the event-based API. We'll later see a simpler way of doing this – via async iteration.

9.2.4 A few preliminary remarks

- We'll only use text streams in this chapter.
- In the examples, we'll occasionally encounter `await` being used at the top level. In that case, we imagine that we are inside a module or inside the body of an async function.
- Whenever there are newlines, we support both:
  - Unix: `'\n'` (LF)
  - Windows: `'\r\n'` (CR LF)

  The newline characters of the current platform can be accessed via the constant `EOL` in module `os`.

9.3 Readable streams

9.3.1 Creating readable streams

9.3.1.1 Creating readable streams from files

We can use `fs.createReadStream()` to create readable streams:

```
import * as fs from 'fs';

const readableStream = fs.createReadStream(
  'tmp/test.txt', {encoding: 'utf8'});

assert.equal(
  await readableToString(readableStream),
  'This is a test!\n');
```

9.3.1.2 `Readable.from()`: Creating readable streams from iterables

The static method `Readable.from(iterable, options?)` creates a readable stream which holds the data contained in `iterable`. `iterable` can be a synchronous iterable

or an asynchronous iterable. The parameter `options` is optional and can, among other things, be used to specify a text encoding.

```
import * as stream from 'stream';

function* gen() {
  yield 'One line\n';
  yield 'Another line\n';
}
const readableStream = stream.Readable.from(gen(), {encoding: 'utf8'});
assert.equal(
  await readableToString(readableStream),
  'One line\nAnother line\n');
```

9.3.1.2.1 Creating readable streams from strings

`Readable.from()` accepts any iterable and can therefore also be used to convert strings to streams:

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

const str = 'Some text!';
const readable = Readable.from(str, {encoding: 'utf8'});
assert.equal(
  await readableToString(readable),
  'Some text!');
```

At the moment, `Readable.from()` treats a string like any other iterable and therefore iterates over its code points. That isn't ideal, performance-wise, but should be OK for most use cases. I expect `Readable.from()` to be often used with strings, so maybe there will be optimizations in the future.

9.3.2 Reading chunks from readable streams via `for-await-of`

Every readable stream is asynchronously iterable, which means that we can use a `for-await-of` loop to read its contents:

```
import * as fs from 'fs';

async function logChunks(readable) {
  for await (const chunk of readable) {
    console.log(chunk);
  }
}

const readable = fs.createReadStream(
  'tmp/test.txt', {encoding: 'utf8'});
logChunks(readable);
```

```
// Output:
// 'This is a test!\n'
```

9.3.2.1 Collecting the contents of a readable stream in a string

The following function is a simpler reimplementation of the function that we have seen at the beginning of this chapter.

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

async function readableToString2(readable) {
  let result = '';
  for await (const chunk of readable) {
    result += chunk;
  }
  return result;
}

const readable = Readable.from('Good morning!', {encoding: 'utf8'});
assert.equal(await readableToString2(readable), 'Good morning!');
```

Note that, in this case, we had to use an async function because we wanted to return a Promise.

9.3.3 Reading lines from readable streams via module `'node:readlines'`

The built-in module `'node:readline'` lets us read lines from readable streams:

```
import * as fs from 'node:fs';
import * as readline from 'node:readline/promises';

const filePath = process.argv[2]; // first command line argument

const rl = readline.createInterface({
  input: fs.createReadStream(filePath, {encoding: 'utf-8'}),
});
for await (const line of rl) {
  console.log('>', line);
}
rl.close();
```

9.4 Transforming readable streams via async generators

Async iteration provides an elegant alternative to transform streams for processing streamed data in multiple steps:

- The input is a readable stream.
- The first transformation is performed by an async generator that iterates over the readable streams and yields as it sees fit.
- Optionally, we can transform further, by using more async generators.

- At the end, we have several options for handling the async iterable returned by the last generator:
  - We can convert it to a readable stream via `Readable.from()` (which can later be piped into a writable stream).
  - We can use an async function to process it.
  - Etc.

To summarize, these are the pieces of such processing pipelines:

readable
□ first async generator [□ … □ last async generator]
□ readable or async function

9.4.1 Going from chunks to numbered lines in async iterables

In the next example, we'll see an example of a processing pipeline as it was just explained.

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

/**
 * @param chunkIterable An asynchronous or synchronous iterable
 * over “chunks” (arbitrary strings)
 * @returns An asynchronous iterable over “lines”
 * (strings with at most one newline that always appears at the end)
 */
async function* chunksToLines(chunkIterable) {
  let previous = '';
  for await (const chunk of chunkIterable) {
    let startSearch = previous.length;
    previous += chunk;
    while (true) {
      // Works for EOL === '\n' and EOL === '\r\n'
      const eolIndex = previous.indexOf('\n', startSearch);
      if (eolIndex < 0) break;
      // Line includes the EOL
      const line = previous.slice(0, eolIndex+1);
      yield line;
      previous = previous.slice(eolIndex+1);
      startSearch = 0;
    }
  }
  if (previous.length > 0) {
    yield previous;
  }
}

async function* numberLines(lineIterable) {
  let lineNumber = 1;
  for await (const line of lineIterable) {
    yield lineNumber + ' ' + line;
```

```
    lineNumber++;
  }
}

async function logLines(lineIterable) {
  for await (const line of lineIterable) {
    console.log(line);
  }
}

const chunks = Readable.from(
  'Text with\nmultiple\nlines.\n',
  {encoding: 'utf8'});
await logLines(numberLines(chunksToLines(chunks))); // (A)

// Output:
// '1 Text with\n'
// '2 multiple\n'
// '3 lines.\n'
```

The processing pipeline is set up in line A. The steps are:

- `chunksToLines()`: Go from an async iterable with chunks to an async iterable with lines.
- `numberLines()`: Go from an async iterable with lines to an async iterable with numbered lines.
- `logLines()`: Log the items in an async iterable.

Observation:

- Both input and output of `chunksToLines()` and `numberLines()` are async iterables. That's why they are async generators (as indicated by `async` and `*`).
- Only the input of `logLines()` is an async iterable. That's why it is an async function (as indicated by `async`).

9.5 Writable streams

9.5.1 Creating writable streams for files

We can use `fs.createWriteStream()` to create writable streams:

```
const writableStream = fs.createWriteStream(
  'tmp/log.txt', {encoding: 'utf8'});
```

9.5.2 Writing to writable streams

In this section, we look at approaches to writing to a writable stream:

1. Writing directly to the writable stream via its method `.write()`.
2. Using function `pipeline()` from module `stream` to pipe a readable stream to the writable stream.

To demonstrate these approaches, we use them to implement the same function `writeIterableToFile()`.

Method `.pipe()` of readable streams also supports piping but it has a downside and it's better to avoid it.

9.5.2.1 `writable.write(chunk)`

When it comes to writing data to streams, there are two callback-based mechanisms that help us:

- Event `'drain'` signals that backpressure is over.
- Function `finished()` invokes a callback when a stream:
  - Is no longer readable or writable
  - Has experienced an error or a premature close event

In the following example, we promisify these mechanisms so that we can use them via an async function:

```
import * as util from 'util';
import * as stream from 'stream';
import * as fs from 'fs';
import {once} from 'events';

const finished = util.promisify(stream.finished); // (A)

async function writeIterableToFile(iterable, filePath) {
  const writable = fs.createWriteStream(filePath, {encoding: 'utf8'});
  for await (const chunk of iterable) {
    if (!writable.write(chunk)) { // (B)
      // Handle backpressure
      await once(writable, 'drain');
    }
  }
  writable.end(); // (C)
  // Wait until done. Throws if there are errors.
  await finished(writable);
}

await writeIterableToFile(
  ['One', ' line of text.\n'], 'tmp/log.txt');
assert.equal(
  fs.readFileSync('tmp/log.txt', {encoding: 'utf8'}),
  'One line of text.\n');
```

The default version of `stream.finished()` is callback-based but can be turned into a Promise-based version via `util.promisify()` (line A).

We used the following two patterns:

- Writing to a writable stream while handling backpressure (line B):

```
    if (!writable.write(chunk)) {
      await once(writable, 'drain');
    }
```

- Closing a writable stream and waiting until writing is done (line C):

```
writable.end();
await finished(writable);
```

9.5.2.2 Piping readable streams to writable streams via `stream.pipeline()`

In line A, we use a promisified version of `stream.pipeline()` to pipe a readable stream `readable` to a writable stream `writable`:

```
import * as stream from 'stream';
import * as fs from 'fs';
const pipeline = util.promisify(stream.pipeline);

async function writeIterableToFile(iterable, filePath) {
  const readable = stream.Readable.from(
    iterable, {encoding: 'utf8'});
  const writable = fs.createWriteStream(filePath);
  await pipeline(readable, writable); // (A)
}
await writeIterableToFile(
  ['One', ' line of text.\n'], 'tmp/log.txt');
// ···
```

9.5.2.3 Not recommended: `readable.pipe(destination)`

Method `readable.pipe()` also supports piping, but has a caveat: If the readable emits an error, then the writable is not closed automatically. `pipeline()` does not have that caveat.

9.6 Quick reference: stream-related functionality

Module `os`:

- `const EOL: string` (since 0.7.8)

  Contains the end-of-line character sequence used by the current platform.

Module `buffer`:

- `Buffer.isEncoding(encoding: string): boolean` (since 0.9.1)

  Returns `true` if `encoding` correctly names one of the supported Node.js encodings for text. Supported encodings include:

  - `'utf8'`
  - `'utf16le'`
  - `'ascii'`

  - `'latin1`
  - `'base64'`
  - `'hex'` (each byte as two hexadecimal characters)

Module `stream`:

- `Readable.prototype[Symbol.asyncIterator](): AsyncIterableIterator<any>` (since 10.0.0)

  Readable streams are asynchronously iterable. For example, you can use `for-await-of` loops in asyc functions or async generators to iterate over them.

- `finished(stream: ReadableStream | WritableStream | ReadWriteStream, callback: (err?: ErrnoException | null) => void): () => Promise<void>` (since 10.0.0)

  The returned Promise is settled when reading/writing is done or there was an error.

  This promisified version is created as follows:

  ```
  const finished = util.promisify(stream.finished);
  ```

- `pipeline(...streams: Array<ReadableStream|ReadWriteStream|WritableStream>): Promise<void>` (since 10.0.0)

  Pipes between streams. The returned Promise is settled when the pipeline is complete or when there was an error.

  This promisified version is created as follows:

  ```
  const pipeline = util.promisify(stream.pipeline);
  ```

- `Readable.from(iterable: Iterable<any> | AsyncIterable<any>, options?: ReadableOptions): Readable` (since 12.3.0)

  Converts an iterable into a readable stream.

  ```
  interface ReadableOptions {
    highWaterMark?: number;
    encoding?: string;
    objectMode?: boolean;
    read?(this: Readable, size: number): void;
    destroy?(this: Readable, error: Error | null,
      callback: (error: Error | null) => void): void;
    autoDestroy?: boolean;
  }
  ```

  These options are the same as the options for the `Readable` constructor and documented there.

Module `fs`:

- `createReadStream(path: string | Buffer | URL, options?: string | {encoding?: string; start?: number}): ReadStream` (since 2.3.0)

  Creates a readable stream. More options are available.

- `createWriteStream(path: PathLike, options?: string | {encoding?: string; flags?: string; mode?: number; start?: number}): WriteStream` (since 2.3.0)

  With option `.flags` you can specify if you want to write or append and what happens if a file does or does not exist. More options are available.

The static type information in this section is based on Definitely Typed.

9.7 Further reading and sources of this chapter

- Section "Streams Compatibility with Async Generators and Async Iterators" in the Node.js docs
- Chapter "Async functions" in "JavaScript for impatient programmers"
- Chapter "Asynchronous iteration" in "JavaScript for impatient programmers"

Chapter 10

Using web streams on Node.js

Contents

Web streams are a standard for *streams* that is now supported on all major web platforms: web browsers, Node.js, and Deno. (Streams are an abstraction for reading and writing data sequentially in small pieces from all kinds of sources – files, data hosted on servers, etc.)

For example, the global function `fetch()` (which downloads online resources) asynchronously returns a Response which has a property `.body` with a web stream.

This chapter covers web streams on Node.js, but most of what we learn applies to all web platforms that support them.

10.1 What are web streams?

Let's start with an overview of a few fundamentals of web streams. Afterwards, we'll quickly move on to examples.

Streams are a data structure for accessing data such as:

- Files
- Data hosted on web servers
- Etc.

Two of their benefits are:

- We can work with large amounts of data because streams allow us to split them up into smaller pieces (so-called *chunks*) which we can process one at a time.
- We can work with the same data structure, streams, while processing different data. That makes it easier to reuse code.

Web streams ("web" is often omitted) are a relatively new standard that originated in web browsers but is now also supported by Node.js and Deno (as shown in this MDN compatibility table).

In web streams, chunks are usually either:

- Text streams: Strings
- Binary streams: Uint8Arrays (a kind of TypedArray)

10.1.1 Kinds of streams

There are three main kinds of web streams:

- A ReadableStream is used to read data from a *source*. Code that does that is called a *consumer*.

- A WritableStream is used to write data to a *sink*. Code that does that is called a *producer*.
- A TransformStream consists of two streams:
  - It receives input from its *writable side*, a WritableStream.
  - It sends output to its *readable side*, a ReadableStream.

  The idea is to transform data by "piping it through" a TransformStream. That is, we write data to the writable side and read transformed data from the readable side. The following TransformStreams are built into most JavaScript platforms (more on them later):
  - Because JavaScript strings are UTF-16 encoded, UTF-8 encoded data is treated as binary in JavaScript. A `TextDecoderStream` converts such data to strings.
  - A `TextEncoderStream` converts JavaScript strings to UTF-8 data.
  - A `'CompressionStream` compresses binary data to GZIP and other compression formats.
  - A `DecompressionStream` decompresses binary data from GZIP and other compression formats.

ReadableStreams, WritableStreams and TransformStreams can be used to transport text or binary data. We'll mostly do the former in this chapter. *Byte streams* for binary data are briefly mentioned at the end.

10.1.2 Pipe chains

Piping is an operation that lets us *pipe* a ReadableStream to a WritableStream: As long as the ReadableStream produces data, this operation reads that data and writes it to the WritableStream. If we connect just two streams, we get a convenient way of transferring data from one location to another (e.g. to copy a file). However, we can also connect more than two streams and get *pipe chains* that can process data in a variety of ways. This is an example of a pipe chain:

- It starts with a ReadableStream.
- Next are one or more TransformStreams.
- The chain ends with a WritableStream.

A ReadableStream is connected to a TransformStream by piping the former to the writable side of the latter. Similarly, a TransformStream is connected to another TransformStream by piping the readable side of the former to the writable side of the latter. And a TransformStream is connected to a WritableStream by piping the readable side of the former to the latter.

10.1.3 Backpressure

One problem in pipe chains is that a member may receive more data than it can handle at the moment. *Backpressure* is a technique for solving this problem: It enables a receiver of data to tell its sender that it should temporarily stop sending data so that the receiver doesn't get overwhelmed.

Another way to look at backpressure is as a signal that travels backwards through a pipe chain, from a member that is getting overwhelmed to the beginning of the chain. As an example, consider the following pipe chain:

```
ReadableStream -pipeTo-> TransformStream -pipeTo-> WriteableStream
```

This is how backpressure travels through this chain:

- Initially, the WriteableStream signals that it can't process more data at the moment.
- The pipe stops reading from the TransformStream.
- Input accumulates inside the TransformStream (which is buffered).
- The TransformStream signals that it's full.
- The pipe stops reading from the ReadableStream.

We have reached the beginning of the pipe chain. Therefore, no data accumulates inside the ReadableStream (which is also buffered) and the WriteableStream has time to recover. Once it does, it signals that it is ready to receive data again. That signal also travels back through the chain until it reaches the ReadableStream and data processing resumes.

In this first look at backpressure, several details were omitted to make things easier to understand. These will be covered later.

10.1.4 Support for web streams in Node.js

In Node.js, web streams are available from two sources:

- From module `'node:stream/web'`
- Via global variables (like in web browsers)

At the moment, only one API has direct support for web streams in Node.js – the Fetch API:

```
const response = await fetch('https://example.com');
const readableStream = response.body;
```

For other things, we need to use one of the following static methods in module `'node:stream'` to either convert a Node.js stream to a web stream or vice versa:

- Node.js Readables can be converted to and from WritableStreams:
  - `Readable.toWeb(nodeReadable)`
  - `Readable.fromWeb(webReadableStream, options?)`
- Node.js Writables can be converted to and from ReadableStreams:
  - `Writable.toWeb(nodeWritable)`
  - `Writable.fromWeb(webWritableStream, options?)`
- Node.js Duplexes can be converted to and from TransformStreams:
  - `Duplex.toWeb(nodeDuplex)`
  - `Duplex.fromWeb(webTransformStream, options?)`

One other API partially supports web streams: FileHandles have the method `.readableWebStream()`.

10.2 Reading from ReadableStreams

ReadableStreams let us read chunks of data from various sources. They have the following type (feel free to skim this type and the explanations of its properties; they will be explained again when we encounter them in examples):

```
interface ReadableStream<TChunk> {
  getReader(): ReadableStreamDefaultReader<TChunk>;
  readonly locked: boolean;
  [Symbol.asyncIterator](): AsyncIterator<TChunk>;

  cancel(reason?: any): Promise<void>;

  pipeTo(
    destination: WritableStream<TChunk>,
    options?: StreamPipeOptions
  ): Promise<void>;
  pipeThrough<TChunk2>(
    transform: ReadableWritablePair<TChunk2, TChunk>,
    options?: StreamPipeOptions
  ): ReadableStream<TChunk2>;

  // Not used in this chapter:
  tee(): [ReadableStream<TChunk>, ReadableStream<TChunk>];
}

interface StreamPipeOptions {
  signal?: AbortSignal;
  preventClose?: boolean;
  preventAbort?: boolean;
  preventCancel?: boolean;
}
```

Explanations of these properties:

- `.getReader()` returns a Reader – an object through which we can read from a ReadableStream. ReadableStreams returning Readers is similar to iterables returning iterators.
- `.locked`: There can only be one active Reader per ReadableStream at a time. While one Reader is in use, the ReadableStream is locked and `.getReader()` cannot be invoked.
- `[Symbol.asyncIterator](https://exploringjs.com/impatient-js/ch_async-iteration.html)`: This method makes ReadableStreams asynchronously iterable. It is currently only implemented on some platforms.
- `.cancel(reason)` cancels the stream because the consumer isn't interested in it anymore. `reason` is passed on to the `.cancel()` method of the ReadableStream's *underlying source* (more on that later). The returned Promise fulfills when this operation is done.
- `.pipeTo()` feeds the contents of its ReadableStream to a WritableStream. The returned Promise fulfills when this operation is done. `.pipeTo()` ensures that back-

pressure, closing, errors, etc. are all correctly propagated through a pipe chain. We can specify options via its second parameter:
  - `.signal` lets us pass an AbortSignal to this method, which enables us to abort piping via an AbortController.
  - `.preventClose`: If `true`, it prevents the WritableStream from being closed when the ReadableStream is closed. That is useful when we want to pipe more than one ReadableStream to the same WritableStream.
  - The remaining options are beyond the scope of this chapter. They are documented in the web streams specification.
- `.pipeThrough()` connects its ReadableStream to a ReadableWritablePair (roughly: a TransformStream, more on that later). It returns the resulting ReadableStream (i.e., the readable side of the ReadableWritablePair).

The following subsections cover three ways of consuming ReadableStreams:

- Reading via Readers
- Reading via asynchronous iteration
- Piping ReadableStreams to WritableStreams

10.2.1 Consuming ReadableStreams via Readers

We can use *Readers* to read data from ReadableStreams. They have the following type (feel free to skim this type and the explanations of its properties; they will be explained again when we encounter them in examples):

```
interface ReadableStreamGenericReader {
  readonly closed: Promise<undefined>;
  cancel(reason?: any): Promise<void>;
}
interface ReadableStreamDefaultReader<TChunk>
  extends ReadableStreamGenericReader
{
  releaseLock(): void;
  read(): Promise<ReadableStreamReadResult<TChunk>>;
}

interface ReadableStreamReadResult<TChunk> {
  done: boolean;
  value: TChunk | undefined;
}
```

Explanations of these properties:

- `.closed`: This Promise is fulfilled after the stream is closed. It is rejected if the stream errors or if a Reader's lock is released before the stream is closed.
- `.cancel()`: In an active Reader, this method cancels the associated ReadableStream.
- `.releaseLock()` deactivates the Reader and unlocks its stream.
- `.read()` returns a Promise for a ReadableStreamReadResult (a wrapped chunk) which has two properties:

- `.done` is a boolean that is `false` as long as chunks can be read and `true` after the last chunk.
- `.value` is the chunk (or `undefined` after the last chunk).

ReadableStreamReadResult may look familiar if you know how iteration works: ReadableStreams are similar to iterables, Readers are similar to iterators, and ReadableStreamReadResults are similar to the objects returned by the iterator method `.next()`.

The following code demonstrates the protocol for using Readers:

```
const reader = readableStream.getReader(); // (A)
assert.equal(readableStream.locked, true); // (B)
try {
  while (true) {
    const {done, value: chunk} = await reader.read(); // (C)
    if (done) break;
    // Use `chunk`
  }
} finally {
  reader.releaseLock(); // (D)
}
```

Getting a Reader. We can't read directly from `readableStream`, we first need to acquire a *Reader* (line A). Each ReadableStream can have at most one Reader. After a Reader was acquired, `readableStream` is locked (line B). Before we can call `.getReader()` again, we must call `.releaseLock()` (line D).

Reading chunks. `.read()` returns a Promise for an object with the properties `.done` and `.value` (line C). After the last chunk was read, `.done` is `true`. This approach is similar to how asynchronous iteration works in JavaScript.

10.2.1.1 Example: reading a file via a ReadableStream

In the following example, we read chunks (strings) from a text file `data.txt`:

```
import * as fs from 'node:fs';
import {Readable} from 'node:stream';

const nodeReadable = fs.createReadStream(
  'data.txt', {encoding: 'utf-8'});
const webReadableStream = Readable.toWeb(nodeReadable); // (A)

const reader = webReadableStream.getReader();
try {
  while (true) {
    const {done, value} = await reader.read();
    if (done) break;
    console.log(value);
  }
} finally {
  reader.releaseLock();
```

```
}
// Output:
// 'Content of text file\n'
```

We are converting a Node.js Readable to a web ReadableStream (line A). Then we use the previously explained protocol to read the chunks.

10.2.1.2 Example: assembling a string with the contents of a ReadableStream

In the next example, we concatenate all chunks of a ReadableStream into a string and return it:

```
/**
 * Returns a string with the contents of `readableStream`.
 */
async function readableStreamToString(readableStream) {
  const reader = readableStream.getReader();
  try {
    let result = '';
    while (true) {
      const {done, value} = await reader.read();
      if (done) {
        return result; // (A)
      }
      result += value;
    }
  } finally {
    reader.releaseLock(); // (B)
  }
}
```

Conveniently, the `finally` clause is always executed – now matter how we leave the `try` clause. That is, the lock is correctly released (line B) if we return a result (line A).

10.2.2 Consuming ReadableStreams via asynchronous iteration

ReadableStreams can also be consumed via asynchronous iteration:

```
const iterator = readableStream[Symbol.asyncIterator]();
let exhaustive = false;
try {
  while (true) {
    let chunk;
    ({done: exhaustive, value: chunk} = await iterator.next());
    if (exhaustive) break;
    console.log(chunk);
  }
} finally {
  // If the loop was terminated before we could iterate exhaustively
  // (via an exception or `return`), we must call `iterator.return()`.
```

```
  // Check if that was the case.
  if (!exhaustive) {
    iterator.return();
  }
}
```

Thankfully, the `for-await-of` loop handles all the details of asynchronous iteration for us:

```
for await (const chunk of readableStream) {
  console.log(chunk);
}
```

10.2.2.1 Example: using asynchronous iteration to read a stream

Let's redo our previous attempt to read text from a file. This time, we use asynchronous iteration instead of a Reader:

```
import * as fs from 'node:fs';
import {Readable} from 'node:stream';

const nodeReadable = fs.createReadStream(
  'text-file.txt', {encoding: 'utf-8'});
const webReadableStream = Readable.toWeb(nodeReadable);
for await (const chunk of webReadableStream) {
  console.log(chunk);
}
// Output:
// 'Content of text file'
```

10.2.2.2 Example: assembling a string with the contents of a ReadableStream

We have previously used a Reader to assemble a string with the contents of a ReadableStream. With asynchronous iteration, the code becomes simpler:

```
/**
 * Returns a string with the contents of `readableStream`.
 */
async function readableStreamToString2(readableStream) {
  let result = '';
  for await (const chunk of readableStream) {
    result += chunk;
  }
  return result;
}
```

10.2.2.3 Caveat: Browsers don't support asynchronous iteration over ReadableStreams

At the moment, Node.js and Deno support asynchronous iteration over ReadableStreams but web browsers don't: There is a GitHub issue that links to bug reports.

Given that it's not yet completely clear how async iteration will be supported on browsers, wrapping is a safer choice than polyfilling. The following code is based on a suggestion in the Chromium bug report:

```
async function* getAsyncIterableFor(readableStream) {
  const reader = readableStream.getReader();
  try {
    while (true) {
      const {done, value} = await reader.read();
      if (done) return;
      yield value;
    }
  } finally {
    reader.releaseLock();
  }
}
```

10.2.3 Piping ReadableStreams to WritableStreams

ReadableStreams have two methods for piping:

- `readableStream.pipeTo(writeableStream)` synchronously returns a Promise `p`. It asynchronously reads all chunks of `readableStream` and writes them to `writableStream`. When it is done, it fulfills `p`.

  We'll see examples of `.pipeTo()` when we explore WritableStreams, as it provides a convenient way to transfer data into them.

- `readableStream.pipeThrough(transformStream)` pipes `readableStream` into `transformStream.writable` and returns `transformStream.readable` (every TransformStream has these properties that refer to its writable side and its readable side). Another way to view this operation is that we create a new ReadableStream by connecting a `transformStream` to a `readableStream`.

  We'll see examples of `.pipeThrough()` when we explore TransformStreams, as this method is the main way in which they are used.

10.3 Turning data sources into ReadableStreams via wrapping

If we want to read an external source via a ReadableStream, we can wrap it in an adapter object and pass that object to the `ReadableStream` constructor. The adapter object is called the *underlying source* of the ReadableStream (queuing strategies are explained later, when we take a closer look at backpressure):

```
new ReadableStream(underlyingSource?, queuingStrategy?)
```

This is the type of underlying sources (feel free to skim this type and the explanations of its properties; they will be explained again when we encounter them in examples):

```
interface UnderlyingSource<TChunk> {
  start?(
    controller: ReadableStreamController<TChunk>
  ): void | Promise<void>;
  pull?(
    controller: ReadableStreamController<TChunk>
  ): void | Promise<void>;
  cancel?(reason?: any): void | Promise<void>;

  // Only used in byte streams and ignored in this section:
  type: 'bytes' | undefined;
  autoAllocateChunkSize: bigint;
}
```

This is when the ReadableStream calls these methods:

- `.start(controller)` is called immediately after we invoke the constructor of `ReadableStream`.
- `.pull(controller)` is called whenever there is room in the internal queue of the ReadableStream. It is called repeatedly until the queue is full again. This method will only be called after `.start()` is finished. If `.pull()` doesn't enqueue anything, it won't be called again.
- `.cancel(reason)` is called if the consumer of a ReadableStream cancels it via `readableStream.cancel()` or `reader.cancel()`. `reason` is the value that was passed to these methods.

Each of these methods can return a Promise and no further steps will be taken until the Promise is settled. That is useful if we want to do something asynchronous.

The parameter `controller` of `.start()` and `.pull()` lets them access the stream. It has the following type:

```
type ReadableStreamController<TChunk> =
  | ReadableStreamDefaultController<TChunk>
  | ReadableByteStreamController<TChunk> // ignored here
;

interface ReadableStreamDefaultController<TChunk> {
  enqueue(chunk?: TChunk): void;
  readonly desiredSize: number | null;
  close(): void;
  error(err?: any): void;
}
```

For now, chunks are strings. We'll later get to byte streams, where Uint8Arrays are common. This is what the methods do:

- `.enqueue(chunk)` adds `chunk` to the ReadableStream's internal queue.
- `.desiredSize` indicates how much room there is in the queue into which `.enqueue()` writes. It is zero if the queue is full and negative if it has exceeded its

maximum size. Therefore, if the desired size is zero or negative, we have to stop enqueuing.
  - If a stream is closed, its desired size is zero.
  - If a stream is in error mode, its desired size is `null`.
- `.close()` closes the ReadableStream. Consumers will still be able to empty the queue, but after that, the stream ends. It's important that an underlying source calls this method – otherwise, reading its stream will never finish.
- `.error(err)` puts the stream in an error mode: All future interactions with it will fail with the error value `err`.

10.3.1 A first example of implementing an underlying source

In our first example of implementing an underlying source, we only provide method `.start()`. We'll see use cases for `.pull()` in the next subsection.

```
const readableStream = new ReadableStream({
  start(controller) {
    controller.enqueue('First line\n'); // (A)
    controller.enqueue('Second line\n'); // (B)
    controller.close(); // (C)
  },
});
for await (const chunk of readableStream) {
  console.log(chunk);
}

// Output:
// 'First line\n'
// 'Second line\n'
```

We use the controller to create a stream with two chunks (line A and line B). It's important that we close the stream (line C). Otherwise, the `for-await-of` loop would never finish!

Note that this way of enqueuing isn't completely safe: There is a risk of exceding the capacity of the internal queue. We'll see soon how we can avoid that risk.

10.3.2 Using a ReadableStream to wrap a push source or a pull source

A common scenario is turning a push source or a pull source into a ReadableStream. The source being push or pull determines how we will hook into the ReadableStream with our UnderlyingSource:

- Push source: Such a source notifies us when there is new data. We use `.start()` to set up listeners and supporting data structures. If we receive too much data and the desired size isn't positive anymore, we must tell our source to pause. If `.pull()` is called later, we can unpause it. Pausing an external source in reaction to the desired size becoming non-positive is called *applying backpressure*.
- Pull source: We ask such a source for new data – often asynchronously. Therefore, we usually don't do much in `.start()` and retrieve data whenever `.pull()` is called.

We'll see examples for both kinds of sources next.

10.3.2.1 Example: creating a ReadableStream from a push source with backpressure support

In the following example, we wrap a ReadableStream around a socket – which pushes its data to us (it calls us). This example is taken from the web stream specification:

```
function makeReadableBackpressureSocketStream(host, port) {
  const socket = createBackpressureSocket(host, port);

  return new ReadableStream({
    start(controller) {
      socket.ondata = event => {
        controller.enqueue(event.data);

        if (controller.desiredSize <= 0) {
          // The internal queue is full, so propagate
          // the backpressure signal to the underlying source.
          socket.readStop();
        }
      };

      socket.onend = () => controller.close();
      socket.onerror = () => controller.error(
        new Error('The socket errored!'));
    },

    pull() {
      // This is called if the internal queue has been emptied, but the
      // stream's consumer still wants more data. In that case, restart
      // the flow of data if we have previously paused it.
      socket.readStart();
    },

    cancel() {
      socket.close();
    },
  });
}
```

10.3.2.2 Example: creating a ReadableStream from a pull source

The tool function `iterableToReadableStream()` takes an iterable over chunks and turns it into a ReadableStream:

```
/**
 * @param iterable an iterable (asynchronous or synchronous)
 */
function iterableToReadableStream(iterable) {
```

```
  return new ReadableStream({
    start() {
      if (typeof iterable[Symbol.asyncIterator] === 'function') {
        this.iterator = iterable[Symbol.asyncIterator]();
      } else if (typeof iterable[Symbol.iterator] === 'function') {
        this.iterator = iterable[Symbol.iterator]();
      } else {
        throw new Error('Not an iterable: ' + iterable);
      }
    },

    async pull(controller) {
      if (this.iterator === null) return;
      // Sync iterators return non-Promise values,
      // but `await` doesn't mind and simply passes them on
      const {value, done} = await this.iterator.next();
      if (done) {
        this.iterator = null;
        controller.close();
        return;
      }
      controller.enqueue(value);
    },

    cancel() {
      this.iterator = null;
      controller.close();
    },
  });
}
```

Let's use an async generator function to create an asynchronous iterable and turn that iterable into a ReadableStream:

```
async function* genAsyncIterable() {
  yield 'how';
  yield 'are';
  yield 'you';
}
const readableStream = iterableToReadableStream(genAsyncIterable());
for await (const chunk of readableStream) {
  console.log(chunk);
}

// Output:
// 'how'
// 'are'
// 'you'
```

`iterableToReadableStream()` also works with synchronous iterables:

```
const syncIterable = ['hello', 'everyone'];
const readableStream = iterableToReadableStream(syncIterable);
for await (const chunk of readableStream) {
  console.log(chunk);
}

// Output:
// 'hello'
// 'everyone'
```

There may eventually by a static helper method `ReadableStream.from()` that provides this functionality (see its pull request for more information).

10.4 Writing to WritableStreams

WritableStreams let us write chunks of data to various sinks. They have the following type (feel free to skim this type and the explanations of its properties; they will be explained again when we encounter them in examples):

```
interface WritableStream<TChunk> {
  getWriter(): WritableStreamDefaultWriter<TChunk>;
  readonly locked: boolean;

  close(): Promise<void>;
  abort(reason?: any): Promise<void>;
}
```

Explanations of these properties:

- `.getWriter()` returns a Writer – an object through which we can write to a WritableStream.
- `.locked`: There can only be one active Writer per WritableStream at a time. While one Writer is in use, the WritableStream is locked and `.getWriter()` cannot be invoked.
- `.close()` closes the stream:
  - The *underlying sink* (more on that later) will still receive all queued chunks before it's closed.
  - From now on, all attempts to write will fail silently (without errors).
  - The method returns a Promise that will be fulfilled if the sink succeeds in writing all queued chunks and closing. It will be rejected if any errors occur during these steps.
- `.abort()` aborts the stream:
  - It puts the stream in error mode.
  - The returned Promise fulfills if the sink shuts down successfully and rejects if errors occur.

The following subsections cover two approaches to sending data to WritableStreams:

- Writing to WritableStreams via Writers

- Piping to WritableStreams

10.4.1 Writing to WritableStreams via Writers

We can use *Writers* to write to WritableStreams. They have the following type (feel free to skim this type and the explanations of its properties; they will be explained again when we encounter them in examples):

```
interface WritableStreamDefaultWriter<TChunk> {
  readonly desiredSize: number | null;
  readonly ready: Promise<undefined>;
  write(chunk?: TChunk): Promise<void>;
  releaseLock(): void;

  close(): Promise<void>;
  readonly closed: Promise<undefined>;
  abort(reason?: any): Promise<void>;
}
```

Explanations of these properties:

- `.desiredSize` indicates how much room there is in this WriteStream's queue. It is zero if the queue is full and negative if it has exceeded its maximum size. Therefore, if the desired size is zero or negative, we have to stop writing.
  - If a stream is closed, its desired size is zero.
  - If a stream is in error mode, its desired size is `null`.
- `.ready` returns a Promise that is fulfilled when the desired size changes from non-positive to positive. That means that no backpressure is active and it's OK to write data. If the desired size later changes back to non-positive, a new pending Promise is created and returned.
- `.write()` writes a chunk to the stream. It returns a Promise that is fulfilled after writing succeeds and rejected if there is an error.
- `.releaseLock()` releases the Writer's lock on its stream.
- `.close()` has the same effect as closing the Writer's stream.
- `.closed` returns a Promise that is fulfilled when the stream is closed.
- `.abort()` has the same effect as aborting the Writer's stream.

The following code shows the protocol for using Writers:

```
const writer = writableStream.getWriter(); // (A)
assert.equal(writableStream.locked, true); // (B)
try {
  // Writing the chunks (explained later)
} finally {
  writer.releaseLock(); // (C)
}
```

We can't write directly to a `writableStream`, we first need to acquire a *Writer* (line A). Each WritableStream can have at most one Writer. After a Writer was acquired, `writableStream` is locked (line B). Before we can call `.getWriter()` again, we must call `.releaseLock()` (line C).

There are three approaches to writing chunks.

10.4.1.1 Writing approach 1: awaiting `.write()` (handling backpressure inefficiently)

The first writing approach is to await each result of `.write()`:

```
await writer.write('Chunk 1');
await writer.write('Chunk 2');
await writer.close();
```

The Promise returned by `.write()` fulfills when the chunk that we passed to it, was successfully written. What exactly "successfully written" means, depends on how a WritableStream is implemented – e.g., with a file stream, the chunk may have been sent to the operating system but still reside in a cache and therefore not have actually been written to disk.

The Promise returned by `.close()` is fulfilled when the stream becomes closed.

A downside of this writing approach is that waiting until writing succeeds means that the queue isn't used. As a consequence, data throughput may be lower.

10.4.1.2 Writing approach 2: ignoring `.write()` rejections (ignoring backpressure)

In the second writing approach, we ignore the Promises returned by `.write()` and only await the Promise returned by `.close()`:

```
writer.write('Chunk 1').catch(() => {}); // (A)
writer.write('Chunk 2').catch(() => {}); // (B)
await writer.close(); // reports errors
```

The synchronous invocations of `.write()` add chunks to the internal queue of the WritableStream. By not awaiting the returned Promises, we don't wait until each chunk is written. However, awaiting `.close()` ensures that the queue is empty and all writing succeeded before we continue.

Invoking `.catch()` in line A and line B is necessary to avoid warnings about unhandled Promise rejections when something goes wrong during writing. Such warnings are often logged to the console. We can afford to ignore the errors reported by `.write()` because `.close()` will also report them to us.

The previous code can be improved by using a helper function that ignores Promise rejections:

```
ignoreRejections(
  writer.write('Chunk 1'),
  writer.write('Chunk 2'),
);
await writer.close(); // reports errors
```

```
function ignoreRejections(...promises) {
  for (const promise of promises) {
    promise.catch(() => {});
  }
}
```

One downside of this approach is that backpressure is ignored: We simply assume that the queue is big enough to hold everything we write.

10.4.1.3 Writing approach 3: awaiting `.ready` (handling backpressure efficiently)

In this writing approach, we handle backpressure efficiently by awaiting the Writer getter `.ready`:

```
await writer.ready; // reports errors
// How much room do we have?
console.log(writer.desiredSize);
writer.write('Chunk 1').catch(() => {});

await writer.ready; // reports errors
// How much room do we have?
console.log(writer.desiredSize);
writer.write('Chunk 2').catch(() => {});

await writer.close(); // reports errors
```

The Promise in `.ready` fulfills whenever the stream transitions from having backpressure to not having backpressure.

10.4.1.4 Example: writing to a file via a Writer

In this example, we create a text file `data.txt` via a WritableStream:

```
import * as fs from 'node:fs';
import {Writable} from 'node:stream';

const nodeWritable = fs.createWriteStream(
  'new-file.txt', {encoding: 'utf-8'}); // (A)
const webWritableStream = Writable.toWeb(nodeWritable); // (B)

const writer = webWritableStream.getWriter();
try {
  await writer.write('First line\n');
  await writer.write('Second line\n');
  await writer.close();
} finally {
  writer.releaseLock()
}
```

In line A, we create a Node.js stream for the file `data.txt`. In line B, we convert this stream to a web stream. Then we use a Writer to write strings to it.

10.4.2 Piping to WritableStreams

Instead of using Writers, we can also write to WritableStreams by piping ReadableStreams to them:

```
await readableStream.pipeTo(writableStream);
```

The Promise returned by `.pipeTo()` fulfills when piping finishes successfully.

10.4.2.1 Piping happens asynchronously

Piping is performed after the current task completes or pauses. The following code demonstrates that:

```
const readableStream = new ReadableStream({ // (A)
  start(controller) {
    controller.enqueue('First line\n');
    controller.enqueue('Second line\n');
    controller.close();
  },
});
const writableStream = new WritableStream({ // (B)
  write(chunk) {
    console.log('WRITE: ' + JSON.stringify(chunk));
  },
  close() {
    console.log('CLOSE WritableStream');
  },
});


console.log('Before .pipeTo()');
const promise = readableStream.pipeTo(writableStream); // (C)
promise.then(() => console.log('Promise fulfilled'));
console.log('After .pipeTo()');

// Output:
// 'Before .pipeTo()'
// 'After .pipeTo()'
// 'WRITE: "First line\n"'
// 'WRITE: "Second line\n"'
// 'CLOSE WritableStream'
// 'Promise fulfilled'
```

In line A we create a ReadableStream. In line B we create a WritableStream.

We can see that `.pipeTo()` (line C) returns immediately. In a new task, chunks are read and written. Then `writableStream` is closed and, finally, `promise` is fulfilled.

10.4.2.2 Example: piping to a WritableStream for a file

In the following example, we create a WritableStream for a file and pipe a ReadableStream to it:

```
const webReadableStream = new ReadableStream({ // (A)
  async start(controller) {
    controller.enqueue('First line\n');
    controller.enqueue('Second line\n');
    controller.close();
  },
});

const nodeWritable = fs.createWriteStream( // (B)
  'data.txt', {encoding: 'utf-8'});
const webWritableStream = Writable.toWeb(nodeWritable); // (C)

await webReadableStream.pipeTo(webWritableStream); // (D)
```

In line A, we create a ReadableStream. In line B, we create a Node.js stream for the file `data.txt`. In line C, we convert this stream to a web stream. In line D, we pipe our `webReadableStream` to the WritableStream for the file.

10.4.2.3 Example: writing two ReadableStreams to a WritableStream

In the following example, we write two ReadableStreams to a single WritableStream.

```
function createReadableStream(prefix) {
  return new ReadableStream({
    async start(controller) {
      controller.enqueue(prefix + 'chunk 1');
      controller.enqueue(prefix + 'chunk 2');
      controller.close();
    },
  });
}

const writableStream = new WritableStream({
  write(chunk) {
    console.log('WRITE ' + JSON.stringify(chunk));
  },
  close() {
    console.log('CLOSE');
  },
  abort(err) {
    console.log('ABORT ' + err);
  },
});

await createReadableStream('Stream 1: ')
```

```
  .pipeTo(writableStream, {preventClose: true}); // (A)
await createReadableStream('Stream 2: ')
  .pipeTo(writableStream, {preventClose: true}); // (B)
await writableStream.close();

// Output
// 'WRITE "Stream 1: chunk 1"'
// 'WRITE "Stream 1: chunk 2"'
// 'WRITE "Stream 2: chunk 1"'
// 'WRITE "Stream 2: chunk 2"'
// 'CLOSE'
```

We tell `.pipeTo()` to not close the WritableStream after the ReadableStream is closed (line A and line B). Therefore, the WritableStream remains open after line A and we can pipe another ReadableStream to it.

10.5 Turning data sinks into WritableStreams via wrapping

If we want to write to an external sink via a WritableStream, we can wrap it in an adapter object and pass that object to the `WritableStream` constructor. The adapter object is called the *underlying sink* of the WritableStream (queuing strategies are explained later, when we take a closer look at backpressure):

```
new WritableStream(underlyingSink?, queuingStrategy?)
```

This is the type of underlying sinks (feel free to skim this type and the explanations of its properties; they will be explained again when we encounter them in examples):

```
interface UnderlyingSink<TChunk> {
  start?(
    controller: WritableStreamDefaultController
  ): void | Promise<void>;
  write?(
    chunk: TChunk,
    controller: WritableStreamDefaultController
  ): void | Promise<void>;
  close?(): void | Promise<void>;;
  abort?(reason?: any): void | Promise<void>;
}
```

Explanations of these properties:

- `.start(controller)` is called immediately after we invoke the constructor of `WritableStream`. If we do something asynchronous, we can return a Promise. In this method, we can prepare for writing.
- `.write(chunk, controller)` is called when a new chunk is ready to be written to the external sink. We can exert backpressure by returning a Promise that fulfills once the backpressure is gone.

- `.close()` is called after `writer.close()` was called and all queued writes succeeded. In this method, we can clean up after writing.
- `.abort(reason)` is called if `writeStream.abort()` or `writer.abort()` were invoked. `reason` is the value passed to these methods.

The parameter `controller` of `.start()` and `.write()` lets them error the WritableStream. It has the following type:

```
interface WritableStreamDefaultController {
  readonly signal: AbortSignal;
  error(err?: any): void;
}
```

- `.signal` is an AbortSignal that we can listen to if we want to abort a write or close operation when the stream is aborted.
- `.error(err)` errors the WritableStream: It is closed and all future interactions with it fail with the error value `err`.

10.5.1 Example: tracing a ReadableStream

In the next example, we pipe a ReadableStream to a WritableStream in order to check how the ReadableStream produces chunks:

```
const readableStream = new ReadableStream({
  start(controller) {
    controller.enqueue('First chunk');
    controller.enqueue('Second chunk');
    controller.close();
  },
});
await readableStream.pipeTo(
  new WritableStream({
    write(chunk) {
      console.log('WRITE ' + JSON.stringify(chunk));
    },
    close() {
      console.log('CLOSE');
    },
    abort(err) {
      console.log('ABORT ' + err);
    },
  })
);
// Output:
// 'WRITE "First chunk"'
// 'WRITE "Second chunk"'
// 'CLOSE'
```

10.5.2 Example: collecting chunks written to a WriteStream in a string

In the next example, we create a subclass of `WriteStream` that collects all written chunks in a string. We can access that string via method `.getString()`:

```
class StringWritableStream extends WritableStream {
  #string = '';
  constructor() {
    super({
      // We need to access the `this` of `StringWritableStream`.
      // Hence the arrow function (and not a method).
      write: (chunk) => {
        this.#string += chunk;
      },
    });
  }
  getString() {
    return this.#string;
  }
}
const stringStream = new StringWritableStream();
const writer = stringStream.getWriter();
try {
  await writer.write('How are');
  await writer.write(' you?');
  await writer.close();
} finally {
  writer.releaseLock()
}
assert.equal(
  stringStream.getString(),
  'How are you?'
);
```

A downside of this approach is that we are mixing two APIs: The API of `WritableStream` and our new string stream API. An alternative is to delegate to the WritableStream instead of extending it:

```
function StringcreateWritableStream() {
  let string = '';
  return {
    stream: new WritableStream({
      write(chunk) {
        string += chunk;
      },
    }),
    getString() {
      return string;
    },
  };
```

```
}

const stringStream = StringcreateWritableStream();
const writer = stringStream.stream.getWriter();
try {
  await writer.write('How are');
  await writer.write(' you?');
  await writer.close();
} finally {
  writer.releaseLock()
}
assert.equal(
  stringStream.getString(),
  'How are you?'
);
```

This functionality could also be implemented via a class (instead of as a factory function for objects).

10.6 Using TransformStreams

A TransformStream:

- Receives input via its *writable side*, a WritableStream.
- It then may or may not transform this input.
- The result can be read via a ReadableStream, its *readable side*.

The most common way to use TransformStreams is to "pipe through" them:

```
const transformedStream = readableStream.pipeThrough(transformStream);
```

`.pipeThrough()` pipes `readableStream` to the writable side of `transformStream` and returns its readable side. In other words: We have created a new ReadableStream that is a transformed version of `readableStream`.

`.pipeThrough()` accepts not only TransformStreams, but any object that has the following shape:

```
interface ReadableWritablePair<RChunk, WChunk> {
  readable: ReadableStream<RChunk>;
  writable: WritableStream<WChunk>;
}
```

10.6.1 Standard TransformStreams

Node.js supports the following standard TransformStreams:

- Encoding (WHATWG standard) – `TextEncoderStream` and `TextDecoderStream`:
  - These streams support UTF-8, but also many "legacy encodings".
  - A single Unicode code point is encoded as up to four UTF-8 code units (bytes). In byte streams, encoded code points be be split across chunks. `TextDecoderStream` handles these cases correctly.

  - Available on most JavaScript platforms (`TextEncoderStream`, `TextDecoderStream`).
- Compression Streams (W3C Draft Community Group Report) – `CompressionStream`, `DecompressionStream`:
  - Currently supported compression formats: `deflate` (ZLIB Compressed Data Format), `deflate-raw` (DEFLATE algorithm), `gzip` (GZIP file format).
  - Available on many JavaScript platforms (`CompressionStream`, `DecompressionStream`).

10.6.1.1 Example: decoding a stream of UTF-8-encoded bytes

In the following example, we decode a stream of UTF-8-encoded bytes:

```
const response = await fetch('https://example.com');
const readableByteStream = response.body;
const readableStream = readableByteStream
  .pipeThrough(new TextDecoderStream('utf-8'));
for await (const stringChunk of readableStream) {
  console.log(stringChunk);
}
```

`response.body` is a ReadableByteStream whose chunks are instances of `Uint8Array` (TypedArrays). We pipe that stream through a `TextDecoderStream` to get a stream that has string chunks.

Note that translating each byte chunk separately (e.g. via a `TextDecoder`) doesn't work because a single Unicode code point is encoded as up to four bytes in UTF-8 and those bytes might not all be in the same chunk.

10.6.1.2 Example: creating a readable text stream for standard input

The following Node.js module logs everything that is sent to it via standard input:

```
// echo-stdin.mjs
import {Readable} from 'node:stream';

const webStream = Readable.toWeb(process.stdin)
  .pipeThrough(new TextDecoderStream('utf-8'));
for await (const chunk of webStream) {
  console.log('>>>', chunk);
}
```

We can access standard input via a stream stored in `process.stdin` (`process` is a global Node.js variable). If we don't set an encoding for this stream and convert it via `Readable.toWeb()`, we get a byte stream. We pipe it through a TextDecoderStream in order to get a text stream.

Note that we process standard input incrementally: As soon as another chunk is available, we log it. In other words, we don't wait until standard input is finished. That is useful when the data is either large or only sent intermittently.

10.7 Implementing custom TransformStreams

We can implement a custom TransformStream by passing a Transformer object to the constructor of `TransformStream`. Such has object has the following type (feel free to skim this type and the explanations of its properties; they will be explained again when we encounter them in examples):

```
interface Transformer<TInChunk, TOutChunk> {
  start?(
    controller: TransformStreamDefaultController<TOutChunk>
  ): void | Promise<void>;
  transform?(
    chunk: TInChunk,
    controller: TransformStreamDefaultController<TOutChunk>
  ): void | Promise<void>;
  flush?(
    controller: TransformStreamDefaultController<TOutChunk>
  ): void | Promise<void>;
}
```

Explanations of these properties:

- `.start(controller)` is called immediately after we invoke the constructor of `TransformStream`. Here we can prepare things before the transformations start.
- `.transform(chunk, controller)` performs the actual transformations. It receives an input chunk and can use its parameter `controller` to enqueue one or more transformed output chunks. It can also choose not to enqueue anything at all.
- `.flush(controller)` is called after all input chunks were transformed successfully. Here we can perform clean-ups after the transformations are done.

Each of these methods can return a Promise and no further steps will be taken until the Promise is settled. That is useful if we want to do something asynchronous.

The parameter `controller` has the following type:

```
interface TransformStreamDefaultController<TOutChunk> {
  enqueue(chunk?: TOutChunk): void;
  readonly desiredSize: number | null;
  terminate(): void;
  error(err?: any): void;
}
```

- `.enqueue(chunk)` adds `chunk` to the readable side (output) of the TransformStream.
- `.desiredSize` returns the desired size of the internal queue of the readable side (output) of the TransformStream.
- `.terminate()` closes the readable side (output) and errors the writable side (input) of the TransformStream. It can be used if a transformer is not interested in the remaining chunks of the writable side (input) and wants to skip them.
- `.error(err)` errors the TransformStream: All future interactions with it will fail with the error value `err`.

What about backpressure in a TransformStream? The class propagates the backpressure from its readable side (output) to its writable side (input). The assumption is that transforming doesn't change the amount of data much. Therefore, Transforms can get away with ignoring backpressure. However, it could be detected via `transformStreamDefaultController.desiredSize` and propagated by returning a Promise from `transformer.transform()`.

10.7.1 Example: transforming a stream of arbitrary chunks to a stream of lines

The following subclass of `TransformStream` converts a stream with arbitrary chunks into a stream where each chunk comprises exactly one line of text. That is, with the possible exception of the last chunk, each chunk ends with an end-of-line (EOL) string: `'\n'` on Unix (incl. macOS) and `'\r\n'` on Windows.

```
class ChunksToLinesTransformer {
  #previous = '';

  transform(chunk, controller) {
    let startSearch = this.#previous.length;
    this.#previous += chunk;
    while (true) {
      // Works for EOL === '\n' and EOL === '\r\n'
      const eolIndex = this.#previous.indexOf('\n', startSearch);
      if (eolIndex < 0) break;
      // Line includes the EOL
      const line = this.#previous.slice(0, eolIndex+1);
      controller.enqueue(line);
      this.#previous = this.#previous.slice(eolIndex+1);
      startSearch = 0;
    }
  }

  flush(controller) {
    // Clean up and enqueue any text we're still holding on to
    if (this.#previous.length > 0) {
      controller.enqueue(this.#previous);
    }
  }
}
class ChunksToLinesStream extends TransformStream {
  constructor() {
    super(new ChunksToLinesTransformer());
  }
}

const stream = new ReadableStream({
  async start(controller) {
```

```
    controller.enqueue('multiple\nlines of\ntext');
    controller.close();
  },
});
const transformStream = new ChunksToLinesStream();
const transformed = stream.pipeThrough(transformStream);

for await (const line of transformed) {
  console.log('>>>', JSON.stringify(line));
}

// Output:
// '>>> "multiple\n"'
// '>>> "lines of\n"'
// '>>> "text"'
```

Note that Deno's built-in `TextLineStream` provides similar functionality.

Tip: We can also make this transformation via an async generator. It would asynchronously iterate over a ReadableStream and return an asynchronous iterable with lines. Its implementation is shown in §9.4 "Transforming readable streams via async generators".

10.7.2 Tip: async generators are also great for transforming streams

Due to ReadableStreams being asynchronously iterable, we can use asynchronous generators to transform them. That leads to very elegant code:

```
const stream = new ReadableStream({
  async start(controller) {
    controller.enqueue('one');
    controller.enqueue('two');
    controller.enqueue('three');
    controller.close();
  },
});

async function* prefixChunks(prefix, asyncIterable) {
  for await (const chunk of asyncIterable) {
    yield '> ' + chunk;
  }
}

const transformedAsyncIterable = prefixChunks('> ', stream);
for await (const transformedChunk of transformedAsyncIterable) {
  console.log(transformedChunk);
}

// Output:
// '> one'
```

```
// '> two'
// '> three'
```

10.8 A closer look at backpressure

Let's take a closer look at backpressure. Consider the following pipe chain:

```
rs.pipeThrough(ts).pipeTo(ws);
```

`rs` is a ReadableStream, `ts` is a TransformStream, `ws` is a WritableStream. These are the connections that are created by the previous expression (`.pipeThrough` uses `.pipeTo` to connect `rs` to the writable side of `ts`):

```
rs -pipeTo-> ts{writable,readable} -pipeTo-> ws
```

Observations:

- The underlying source of `rs` can be viewed as a pipe chain member that comes before `rs`.
- The underlying sink of `ws` can be viewed as a pipe chain member that comes after `ws`.
- Each stream has an internal buffer: ReadableStreams buffers after their underlying sources. WritableStreams have buffers before their underlying sinks.

Let's assume that the underlying sink of `ws` is slow and the buffer of `ws` is eventually full. Then the following steps happen:

- `ws` signals it's full.
- `pipeTo` stops reading from `ts.readable`.
- `ts.readable` signals it's full.
- `ts` stops moving chunks from `ts.writable` to `ts.readable`.
- `ts.writable` signals it's full.
- `pipeTo` stops reading from `rs`.
- `rs` signals it's full to its underlying source.
- The underlying source pauses.

This example illustrates that we need two kinds of functionality:

- Entities receiving data need to be able to signal backpressure.
- Entities sending data need to react to signals by exerting backpressure.

Let's explore how these functionalities are implemented in the web streams API.

10.8.1 Signalling backpressure

Backpressure is signalled by entities that are receiving data. Web streams have two such entities:

- A WritableStream receives data via the Writer method `.write()`.
- A ReadableStream receives data when its underlying source calls the ReadableStreamDefaultController method `.enqueue()`.

In both cases, the input is buffered via queues. The signal to apply backpressure is when a queue is full. Let's see how that can be detected.

These are the locations of the queues:

- The queue of a WritableStream is stored internally in the WritableStreamDefaultController (see web streams standard).
- The queue of a ReadableStream is stored internally in the ReadableStreamDefaultController (see web streams standard).

The *desired size* of a queue is a number that indicates how much room is left in the queue:

- It is positive if there is still room in the queue.
- It is zero if the queue has reached its maximum size.
- It is negative if the queue has exceeded its maximum size.

Therefore, we have to apply backpressure if the desired size is zero or less. It is available via the getter `.desiredSize` of the object which contains the queue.

How is the desired size computed? Via an object that specifies a so-called *queuing strategy*. `ReadableStream` and `WritableStream` have default queuing strategies which can be overridden via optional parameters of their constructors. The interface `QueuingStrategy` has two properties:

- Method `.size(chunk)` returns a size for `chunk`.
  - The current size of a queue is the sum of the sizes of the chunks it contains.
- Property `.highWaterMark` specifies the maximum size of a queue.

The desired size of a queue is the high water mark minus the current size of the queue.

10.8.2 Reacting to backpressure

Entities sending data need to react to signalled backpressure by exerting backpressure.

10.8.2.1 Code writing to a WritableStream via a Writer

- We can await the Promise in `writer.ready`. While we do, we are blocked and the desired backpressure is achieved. The Promise is fulfilled once there is room in the queue. Fulfillment is triggered when `writer.desiredSize` has a value greater than zero.
- Alternatively, we can await the Promise returned by `writer.write()`. If we do that, the queue won't even be filled.

If we want to, we can additionally base the size of our chunks on `writer.desiredSize`.

10.8.2.2 The underlying source of a ReadableStream

The underlying source object that can be passed to a ReadableStream wraps an external source. In a way, it is also a member of the pipe chain; one that comes before its ReadableStream.

- Underlying pull sources are only asked for new data whenever there is room in the queue. While there isn't, backpressure is exerted automatically because no data is pulled.
- Underlying push sources should check `controller.desiredSize` after enqueuing something: If it's zero or less, they should exert backpressure by pausing their external sources.

10.8.2.3 The underlying sink of a WritableStream

The underlying sink object that can be passed to a WritableStream wraps an external sink. In a way, it is also a member of the pipe chain; one that comes after its WritableStream.

Each external sink signals backpressure differently (in some cases not at all). The underlying sink can exert backpressure by returning a Promise from method `.write()` that is fulfilled once writing is finished. There is an example in the web streams standard that demonstrates how that works.

10.8.2.4 A transformStream (`.writable` → `.readable`)

The TransformStream connects its writable side with its readable side by implementing an underlying sink for the former and an underlying source for the latter. It has an internal slot `.[[backpressure]]` that indicates if internal backpressure is currently active or not.

- Method `.write()` of the underlying sink of the writable side waits asynchronously until there is no internal backpressure before it feeds another chunk to the TransformStream's transformer (web streams standard: `TransformStreamDefaultSinkWriteAlgorithm`). The transformer may then enqueue something via its TransformStreamDefaultController. Note that `.write()` returns a Promise that fulfills when the method is finished. Until that happens, the WriteStream buffers incoming write requests via its queue. Therefore, backpressure for the writable side is signalled via that queue and its desired size.
- The TransformStream's backpressure is activated if a chunk is enqueued via the TransformStreamDefaultController and the queue of the readable side becomes full (web streams standard: `TransformStreamDefaultControllerEnqueue`).
- The TransformStream's backpressure may be deactivated if something is read from the Reader (web streams standard: `ReadableStreamDefaultReaderRead`):
  - If there is room in the queue now, it may be time to call `.pull()` of the underlying source (web streams standard: `.[[PullSteps]]`).
  - `.pull()` of the underlying source of the readable side deactivates the backpressure (web streams standard: `TransformStreamDefaultSourcePullAlgorithm`).

10.8.2.5 `.pipeTo()` (ReadableStream → WritableStream)

`.pipeTo()` reads chunks from the ReadableStream via a reader and write them to the WritableStream via a Writer. It pauses whenever `writer.desiredSize` is zero or less (web streams standard: Step 15 of `ReadableStreamPipeTo`).

10.9 Byte streams

So far, we have only worked with *text streams*, streams whose chunks were strings. But the web streams API also supports *byte streams* for binary data, where chunks are Uint8Arrays (TypedArrays):

- `ReadableStream` has a special `'bytes'` mode.
- `WritableStream` itself doesn't care if chunks are strings or Uint8Arrays. Therefore, whether an instance is a text stream or a byte stream depends on what kind of chunks the underlying sink can handle.
- What kind of chunks a `TransformStream` can handle also depends on its Transformer.

Next, we'll learn how to create readable byte streams.

10.9.1 Readable byte streams

What kind of stream is created by the `ReadableStream` constructor depends on the optional property `.type` of its optional first parameter `underlyingSource`:

- If `.type` is omitted or no underlying source is provided, the new instance is a text stream.
- If `.type` is the string `'bytes'`, the new instance is a byte stream:

```
const readableByteStream = new ReadableStream({
  type: 'bytes',
  async start() { /*...*/ }
  // ...
});
```

What changes if a ReadableStream is in `'bytes'` mode?

In default mode, the underlying source can return any kind of chunk. In bytes mode, the chunks must be ArrayBufferViews, i.e. TypedArrays (such as Uint8Arrays) or DataViews.

Additionally, a readable byte stream can create two kinds of readers:

- `.getReader()` returns an instance of `ReadableStreamDefaultReader`.
- `.getReader({mode: 'byob'})` returns an instance of `ReadableStreamBYOBReader`.

"BYOB" stands for "Bring Your Own Buffer" and means that we can pass a buffer (an ArrayBufferView) to `reader.read()`. Afterwards, that ArrayBufferView will be detached and no longer usable. But `.read()` returns its data in a new ArrayBufferView that has the same type and accesses the same region of the same ArrayBuffer.

Additionally, readable byte streams have different controllers: They are instances of `ReadableByteStreamController` (vs. `ReadableStreamDefaultController`). Apart from forcing underlying sources to enqueue ArrayBufferViews (TypedArrays or DataViews), it also supports ReadableStreamBYOBReaders via its property `.byobRequest`. An underlying source writes its data into the BYOBRequest stored in this property. The web streams standard has two examples of using `.byobRequest` in its section "Examples of creating streams".

10.9.2 Example: an infinite readable byte stream filled with random data

In the next example, create an infinite readable byte stream that fills its chunks with random data (inspiration: `example4.mjs` in "Implementing the Web Streams API in Node.js").

```
import {promisify} from 'node:util';
import {randomFill} from 'node:crypto';
const asyncRandomFill = promisify(randomFill);

const readableByteStream = new ReadableStream({
  type: 'bytes',
  async pull(controller) {
    const byobRequest = controller.byobRequest;
    await asyncRandomFill(byobRequest.view);
    byobRequest.respond(byobRequest.view.byteLength);
  },
});

const reader = readableByteStream.getReader({mode: 'byob'});
const buffer = new Uint8Array(10); // (A)
const firstChunk = await reader.read(buffer); // (B)
console.log(firstChunk);
```

Due to `readableByteStream` being infinite, we can't loop over it. That's why we only read its first chunk (line B).

The buffer we create in line A is transferred and therefore unreadable after line B.

10.9.3 Example: compressing a readable byte stream

In the following example, we create a readable byte stream and pipe it through a stream that compresses it to the GZIP format:

```
const readableByteStream = new ReadableStream({
  type: 'bytes',
  start(controller) {
    // 256 zeros
    controller.enqueue(new Uint8Array(256));
    controller.close();
  },
});
const transformedStream = readableByteStream.pipeThrough(
  new CompressionStream('gzip'));
await logChunks(transformedStream);

async function logChunks(readableByteStream) {
  const reader = readableByteStream.getReader();
  try {
```

```
      while (true) {
        const {done, value} = await reader.read();
        if (done) break;
        console.log(value);
      }
    } finally {
      reader.releaseLock();
    }
  }
```

10.9.4 Example: reading a web page via `fetch()`

The result of `fetch()` resolves to a response object whose property `.body` is a readable byte stream. We convert that byte stream to a text stream via `TextDecoderStream`:

```
const response = await fetch('https://example.com');
const readableByteStream = response.body;
const readableStream = readableByteStream.pipeThrough(
  new TextDecoderStream('utf-8'));
for await (const stringChunk of readableStream) {
  console.log(stringChunk);
}
```

10.10 Node.js-specific helpers

Node.js is the only web platform that supports the following helper functions that it calls *utility consumers*:

```
import {
  arrayBuffer,
  blob,
  buffer,
  json,
  text,
} from 'node:stream/consumers';
```

These functions convert web ReadableStreams, Node.js Readables and AsyncIterators to Promises that are fulfilled with:

- ArrayBuffers (`arrayBuffer()`)
- Blobs (`blob()`)
- Node.js Buffers (`buffer()`)
- JSON objects (`json()`)
- Strings (`text()`)

Binary data is assumed to be UTF-8-encoded:

```
import * as streamConsumers from 'node:stream/consumers';

const readableByteStream = new ReadableStream({
```

```
    type: 'bytes',
    start(controller) {
      // TextEncoder converts strings to UTF-8 encoded Uint8Arrays
      const encoder = new TextEncoder();
      const view = encoder.encode('"□"');
      assert.deepEqual(
        view,
        Uint8Array.of(34, 240, 159, 152, 128, 34)
      );
      controller.enqueue(view);
      controller.close();
    },
  });
  const jsonData = await streamConsumers.json(readableByteStream);
  assert.equal(jsonData, '□');
```

String streams work as expected:

```
  import * as streamConsumers from 'node:stream/consumers';

  const readableByteStream = new ReadableStream({
    start(controller) {
      controller.enqueue('"□"');
      controller.close();
    },
  });
  const jsonData = await streamConsumers.json(readableByteStream);
  assert.equal(jsonData, '□');
```

10.11 Further reading

All of the material mentioned in this section was a source for this chapter.

This chapter doesn't cover every aspect of the web streams API. You can find more information here:

- "WHATWG Streams Standard" by Adam Rice, Domenic Denicola, Mattias Buelens, and □ □ □ □ (Takeshi Yoshino)
- "Web Streams API" in the Node.js documentation

More material:

- Web streams API:
  - "Implementing the Web Streams API in Node.js" by James M. Snell
  - "Streams API" on MDN
  - "Streams—The definitive guide" by Thomas Steiner
- Backpressure:
  - "Node.js Backpressuring in Streams" by Vladimir Topolev
  - "Backpressuring in Streams" in the Node.js documentation

- Unicode (code points, UTF-8, UTF-16, etc.): Chapter "Unicode – a brief introduction" in "JavaScript for impatient programmers"
- Chapter "Asynchronous iteration" in "JavaScript for impatient programmers"
- Chapter "Typed Arrays: handling binary data" in "JavaScript for impatient programmers"

Chapter 11

Stream recipes

Contents

11.1 Writing to standard output (stdout)

These are three options for writing to stdout:

- We can write to it via `console.log()`.
- We can write to it via a Node.js stream.
- We can write to it via a web stream.

11.1.1 Writing to stdout via ‘console.log()

`console.log(format, ...args)` writes to stdout and always appends a newline `'\n'` (even on Windows). The first argument can include placeholders which are interpreted in the same way as they are by `util.format()`:

```
console.log('String: %s Number: %d Percent: %%', 'abc', 123);

const obj = {one: 1, two: 2};
```

```
console.log('JSON: %j Object: %o', obj, obj);

// Output:
// 'String: abc Number: 123 Percent: %'
// 'JSON: {"one":1,"two":2} Object: { one: 1, two: 2 }'
```

All arguments after the first one always show up in the output, even if there are not enough placeholders for them.

11.1.2 Writing to stdout via a Node.js stream

`process.stdout` is an instance of `stream.Readable`. That means that we can use it like any other Node.js stream – for example:

```
process.stdout.write('two');
process.stdout.write(' words');
process.stdout.write('\n');
```

The previous code is equivalent to:

```
console.log('two words');
```

Note that there is no newline at the end in this case because `console.log()` always adds one.

If we use `.write()` with large amounts of data, we should take backpressure into consideration, as explained in §9.5.2.1 "`writable.write(chunk)`".

The following recipes work with `process.stdout`: §11.4 "Node.js stream recipes".

11.1.3 Writing to stdout via a web stream

We can convert `process.stdout` to a web stream and write to it:

```
import {Writable} from 'node:stream';
const webOut = Writable.toWeb(process.stdout);
const writer = webOut.getWriter();
try {
  await writer.write('First line\n');
  await writer.write('Second line\n');
  await writer.close();
} finally {
  writer.releaseLock()
}
```

The following recipes work with `webOut`: §11.5 "Web stream recipes".

11.2 Writing to standard error (stderr)

Writing to stderr works similarly to writing to stdout:

- We can write to it via `console.error()`.
- We can write to it via a Node.js stream.

- We can write to it via a web stream.

See the previous section for more information.

11.3 Reading from standard input (stdin)

These are options for reading from stdin:

- We can read from it via a Node.js stream.
- We can read from it via a web stream.
- We can use module `'node:readline'`.

11.3.1 Reading from stdin via a Node.js stream

`process.stdin` is an instance of `stream.Writable`. That means that we can use it like any other Node.js stream:

```
// Switch to text mode (otherwise we get chunks of binary data)
process.stdin.setEncoding('utf-8');
for await (const chunk of process.stdin) {
  console.log('>', chunk);
}
```

The following recipes work with `webIn`: §11.4 "Node.js stream recipes".

11.3.2 Reading from stdin via a web stream

We first have to convert `process.stdin` to a web stream:

```
import {Readable} from 'node:stream';
// Switch to text mode (otherwise we get chunks of binary data)
process.stdin.setEncoding('utf-8');
const webIn = Readable.toWeb(process.stdin);
for await (const chunk of webIn) {
  console.log('>', chunk);
}
```

The following recipes work with `webIn`: §11.5 "Web stream recipes".

11.3.3 Reading from stdin via module `'node:readline'`

The built-in module `'node:readline'` lets us prompt users to enter information interactively – for example:

```
import * as fs from 'node:fs';
import * as readline from 'node:readline/promises';

const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout,
});
```

```
const filePath = await rl.question('Please enter a file path: ');
fs.writeFileSync(filePath, 'Hi!', {encoding: 'utf-8'})

rl.close();
```

For more information on module `'node:readline'`, see:

- §9.3.3 "Reading lines from readable streams via module `'node:readlines'`"
- Its official documentation.

11.4 Node.js stream recipes

Readable streams:

- §9.3.1.2 "`Readable.from()`: Creating readable streams from iterables"
- §9.3.2 "Reading chunks from readable streams via `for-await-of`"
  - §9.3.2.1 "Collecting the contents of a readable stream in a string"
- §9.3.3 "Reading lines from readable streams via module `'node:readlines'`"
- §9.4 "Transforming readable streams via async generators"
  - §9.4.1 "Going from chunks to numbered lines in async iterables"

Writable streams:

- §9.5.2 "Writing to writable streams"
- §9.5.2.2 "Piping readable streams to writable streams via `stream.pipeline()`"

11.5 Web stream recipes

Creating a ReadableStream from:

- Strings: §10.3.1 "A first example of implementing an underlying source"
- An iterable: §10.3.2.2 "Example: creating a ReadableStream from a pull source"

Reading from a ReadableStream:

- §10.2.1 "Consuming ReadableStreams via Readers"
- §10.2.2 "Consuming ReadableStreams via asynchronous iteration"
  - §10.2.2.2 "Example: assembling a string with the contents of a ReadableStream"
- §10.2.3 "Piping ReadableStreams to WritableStreams"

Transforming ReadableStreams:

- §10.6 "Using TransformStreams"
- §10.7.2 "Tip: async generators are also great for transforming streams"
- §10.7.1 "Example: transforming a stream of arbitrary chunks to a stream of lines"

Using WritableStreams:

- §10.4 "Writing to WritableStreams"
- §10.5.2 "Example: collecting chunks written to a WriteStream in a string"

Chapter 12

Running shell commands in child processes

Contents

In this chapter, we'll explore how we can execute shell commands from Node.js, via module `'node:child_process'`.

12.1 Overview of this chapter

Module `'node:child_process'` has a function for executing shell commands (in *spawned* child processes) that comes in two versions:

- An asynchronous version `spawn()`.
- A synchronous version `spawnSync()`.

We'll first explore `spawn()` and then `spawnSync()`. We'll conclude by looking at the following functions that are based on them and relatively similar:

- Based on `spawn()`:
  - `exec()`
  - `execFile()`
- Based on `spawnSync()`:
  - `execSync()`
  - `execFileSync()`

12.1.1 Windows vs. Unix

The code shown in this chapter runs on Unix, but I have also tested it on Windows – where most of it works with minor changes (such as ending lines with `'\r\n'` instead of `'\n'`).

12.1.2 Functionality we often use in the examples

The following functionality shows up often in the examples. That's why it's explained here, once:

- Assertions: `assert.equal()` for primitive values and `assert.deepEqual()` for objects. The necessary import is never shown in the examples:

  ```
  import * as assert from 'node:assert/strict';
  ```

- Function `Readable.toWeb()` converts Node's native `stream.Readable` to a web stream (an instance of `ReadableStream`). It is explained in §10 "Using web streams on Node.js". `Readable` is always imported in the examples.

- The asynchronous function `readableStreamToString()` consumes a readable web stream and returns a string (wrapped in a Promise). It is explained in the chapter on web streams. This function is assumed to available in the examples.

12.2 Spawning processes asynchronously: `spawn()`

12.2.1 How `spawn()` works

```
spawn(
  command: string,
  args?: Array<string>,
  options?: Object
): ChildProcess
```

`spawn()` asynchronously executes a command in a new process: The process runs concurrently to Node's main JavaScript process and we can communicate with it in various ways (often via streams).

Next, there is documentation for the parameters and the result of `spawn()`. If you prefer to learn by example, you can skip that content and continue with the subsections that follow.

12.2.1.1 Parameter: `command`

`command` is a string with the shell command. There are two modes of using this parameter:

- Command-only mode: `args` is omitted and `command` contains the whole shell command. We can even use shell features such as piping between multiple executables, redirecting I/O into files, variables, and wildcards.
  - `options.shell` must be `true` because we need an shell to handle the shell features.
- Args mode: `command` contains only the name of the command and `args` contains its arguments.
  - If `options.shell` is `true`, many meta-characters inside arguments are interpreted and features such as wildcards and variable names work.
  - If `options.shell` is `false`, strings are used verbatim and we never have to escape meta-characters.

Both modes are demonstrated later in this chapter.

12.2.1.2 Parameter: `options`

The following `options` are most interesting:

- `.shell: boolean|string` (default: `false`)
  Should a shell be used to execute the command?
  - On Windows, this option should almost always be `true`. For example, `.bat` and `.cmd` files cannot be executed otherwise.
  - On Unix, only core shell features (e.g. piping, I/O redirection, filename wildcards, and variables) are not available if `.shell` is `false`.
  - If `.shell` is `true`, we have to be careful with user input and sanitize it because it's easy to execute arbitrary code. We also have to escape meta-characters if we want to use them as non-meta-characters.
  - We can also set `.shell` to the path of a shell executable. Then Node.js uses that executable to execute the command. If we set `.shell` to `true`, Node.js uses:

    * Unix: `'/bin/sh'`
    * Windows: `process.env.ComSpec`
- `.cwd: string | URL`
  Specifies the *current working directory* (CWD) to use while executing the command.
- `.stdio: Array<string|Stream>|string`
  Configures how standard I/O is set up. This is explained below.
- `.env: Object` (default: `process.env`)
  Lets us specify shell variables for the child process. Tips:
  - Look at `process.env` (e.g. in the Node.js REPL) to see what variables exist.
  - We can use spreading to non-destructively override an existing variable – or create it if it doesn't exist yet:
    ```
    {env: {...process.env, MY_VAR: 'Hi!'}}
    ```
- `.signal: AbortSignal`
  If we create an AbortController `ac`, we can pass `ac.signal` to `spawn()` and abort the child process via `ac.abort()`. That is demonstrated later in this chapter.
- `.timeout: number`
  If the child process takes longer than `.timeout` milliseconds, it is killed.

12.2.1.3 `options.stdio`

Each of the standard I/O streams of the child process has a numeric ID, a so-called *file descriptor*:

- Standard input (stdin) has the file descriptor 0.
- Standard output (stdout) has the file descriptor 1.
- Standard error (stderr) has the file descriptor 2.

There can be more file descriptors, but that's rare.

`options.stdio` configures if and how the streams of the child process are piped to streams in the parent process. It can be an Array where each element configures the file descriptor that is equal to its index. The following values can be used as Array elements:

- `'pipe'`:
  - Index 0: Pipe `childProcess.stdin` to the child's stdin. Note that, despite its name, the former is a stream that belongs to the parent process.
  - Index 1: Pipe the child's stdout to `childProcess.stdout`.
  - Index 2: Pipe the child's stderr to `childProcess.stderr`.
- `'ignore'`: Ignore the child's stream.
- `'inherit'`: Pipe the child's stream to the corresponding stream of the parent process.
  - For example, if we want the child's stderr to be logged to the console, we can use `'inherit'` at index 2.
- Native Node.js stream: Pipe to or from that stream.
- Other values are supported, too, but that's beyond the scope of this chapter.

Instead of specifying `options.stdio` via an Array, we can also abbreviate:

- `'pipe'` is equivalent to `['pipe', 'pipe', 'pipe']` (the default for `options.stdio`).
- `'ignore'` is equivalent to `['ignore', 'ignore', 'ignore']`.
- `'inherit'` is equivalent to `['inherit', 'inherit', 'inherit']`.

12.2.1.4 Result: instance of `ChildProcess`

`spawn()` returns instances of `ChildProcess`.

Interesting data properties:

- `.exitCode: number | null`
  Contains the code with which the child process exited:
  - 0 (zero) means normal exit.
  - A number greater than zero means an error happened.
  - `null` means the process hasn't exited yet.
- `.signalCode: string | null`
  The POSIX signal with which a child process was killed or `null` if it wasn't. See the description of method `.kill()` below for more information.
- Streams: Depending on how standard I/O is configured (see previous subsection), the following streams become available:
  - `.stdin`
  - `.stdout`
  - `.stderr`
- `.pid: number | undefined`
  The *process identifier* (PID) of the child process. If spawning fails, `.pid` is `undefined`. This value is available immediately after calling `spawn()`.

Interesting methods:

- `.kill(signalCode?: number | string = 'SIGTERM'): boolean`
  Sends a POSIX signal to the child process (which usually results in the termination of the process):

  - The man page for `signal` contains a list of values.
  - Windows does not support signals, but Node.js emulates some of them – e.g.: `SIGINT`, `SIGTERM`, and `SIGKILL`. For more information, see the Node.js documentation.

  This method is demonstrated later in this chapter.

Interesting events:

- `.on('exit', (exitCode: number|null, signalCode: string|null) => {})`
  This event is emitted after the child process ends:
  - The callback parameters provide us with either the exit code or the signal code: One of them will always be non-null.
  - Some of its standard I/O streams might still be open because multiple processes might share the same streams. Event `'close'` notifies us when all stdio streams are closed after the exit of a child process.
- `.on('error', (err: Error) => {})`
  This event is most commonly emitted if a process could not be spawned (see ex-

ample later) or the child process could not be killed. An `'exit'` event may or may not be emitted after this event.

We'll see later how events can be turned into Promises that can be awaited.

12.2.2 When is the shell command executed?

When using the asynchronous `spawn()`, the child process for the command is started asynchronously. The following code demonstrates that:

```
import {spawn} from 'node:child_process';

spawn(
  'echo', ['Command starts'],
  {
    stdio: 'inherit',
    shell: true,
  }
);
console.log('After spawn()');
```

This is the output:

```
After spawn()
Command starts
```

12.2.3 Command-only mode vs. args mode

In this section, we specify the same command invocation in two ways:

- Command-only mode: We provide the whole invocation via the first parameter `command`.
- Args mode: We provide the command via the first parameter `command` and its arguments via the second parameter `args`.

12.2.3.1 Command-only mode

```
import {Readable} from 'node:stream';
import {spawn} from 'node:child_process';

const childProcess = spawn(
  'echo "Hello, how are you?"',
  {
    shell: true, // (A)
    stdio: ['ignore', 'pipe', 'inherit'], // (B)
  }
);
const stdout = Readable.toWeb(
  childProcess.stdout.setEncoding('utf-8'));

// Result on Unix
```

```
assert.equal(
  await readableStreamToString(stdout),
  'Hello, how are you?\n' // (C)
);

// Result on Windows: '"Hello, how are you?"\r\n'
```

Each command-only spawning with arguments requires `.shell` to be `true` (line A) – even if it's as simple as this one.

In line B, we tell `spawn()` how to handle standard I/O:

- Ignore standard input.
- Pipe the child process stdout to `childProcess.stdout` (a stream that belongs to the parent process).
- Pipe child process stderr to parent process stderr.

In this case, we are only interested in the output of the child process. Therefore, we are done once we have processed the output. In other cases, we might have to wait until the child exits. How to do that, is demonstrated later.

In command-only mode, we see more pecularities of shells – for example, the Windows Command shell output includes double quotes (last line).

12.2.3.2 Args mode

```
import {Readable} from 'node:stream';
import {spawn} from 'node:child_process';

const childProcess = spawn(
  'echo', ['Hello, how are you?'],
  {
    shell: true,
    stdio: ['ignore', 'pipe', 'inherit'],
  }
);
const stdout = Readable.toWeb(
  childProcess.stdout.setEncoding('utf-8'));

// Result on Unix
assert.equal(
  await readableStreamToString(stdout),
  'Hello, how are you?\n'
);
// Result on Windows: 'Hello, how are you?\r\n'
```

12.2.3.3 Meta-characters in `args`

Let's explore what happens if there are meta-characters in `args`:

```
import {Readable} from 'node:stream';
import {spawn} from 'node:child_process';
```

```
async function echoUser({shell, args}) {
  const childProcess = spawn(
    `echo`, args,
    {
      stdio: ['ignore', 'pipe', 'inherit'],
      shell,
    }
  );
  const stdout = Readable.toWeb(
    childProcess.stdout.setEncoding('utf-8'));
  return readableStreamToString(stdout);
}

// Results on Unix
assert.equal(
  await echoUser({shell: false, args: ['$USER']}), // (A)
  '$USER\n'
);
assert.equal(
  await echoUser({shell: true, args: ['$USER']}), // (B)
  'rauschma\n'
);
assert.equal(
  await echoUser({shell: true, args: [String.raw`\$USER`]}), // (C)
  '$USER\n'
);
```

- If we don't use a shell, meta-characters such as the dollar sign (`$`) have no effect (line A).
- With a shell, `$USER` is interpreted as a variable (line B).
- If we don't want that, we have to escape the dollar sign via a backslash (line C).

Similar effects occur with other meta-characters such as asterisks (`*`).

These were two examples of Unix shell meta-characters. Windows shells have their own meta-characters and their own ways of escaping.

12.2.3.4 A more complicated shell command

Let's use more shell features (which requires command-only mode):

```
import {Readable} from 'node:stream';
import {spawn} from 'node:child_process';
import {EOL} from 'node:os';

const childProcess = spawn(
  `(echo cherry && echo apple && echo banana) | sort`,
  {
    stdio: ['ignore', 'pipe', 'inherit'],
```

```
    shell: true,
  }
);
const stdout = Readable.toWeb(
  childProcess.stdout.setEncoding('utf-8'));
assert.equal(
  await readableStreamToString(stdout),
  'apple\nbanana\ncherry\n'
);
```

12.2.4 Sending data to the stdin of the child process

So far, we have only read the standard output of a child process. But we can also send data to standard input:

```
import {Readable, Writable} from 'node:stream';
import {spawn} from 'node:child_process';

const childProcess = spawn(
  `sort`, // (A)
  {
    stdio: ['pipe', 'pipe', 'inherit'],
  }
);
const stdin = Writable.toWeb(childProcess.stdin); // (B)
const writer = stdin.getWriter(); // (C)
try {
  await writer.write('Cherry\n');
  await writer.write('Apple\n');
  await writer.write('Banana\n');
} finally {
  writer.close();
}

const stdout = Readable.toWeb(
  childProcess.stdout.setEncoding('utf-8'));
assert.equal(
  await readableStreamToString(stdout),
  'Apple\nBanana\nCherry\n'
);
```

We use the shell command `sort` (line A) to sort lines of text for us.

In line B, we use `Writable.toWeb()` to convert a native Node.js stream to a web stream (for more information, see §10 "Using web streams on Node.js").

How to write to a WritableStream via a writer (line C) is also explained in the chapter on web streams.

12.2.5 Piping manually

We previously let a shell execute the following command:

```
(echo cherry && echo apple && echo banana) | sort
```

In the following example, we do the piping manually, from the echoes (line A) to the sorting (line B):

```
import {Readable, Writable} from 'node:stream';
import {spawn} from 'node:child_process';

const echo = spawn( // (A)
  `echo cherry && echo apple && echo banana`,
  {
    stdio: ['ignore', 'pipe', 'inherit'],
    shell: true,
  }
);
const sort = spawn( // (B)
  `sort`,
  {
    stdio: ['pipe', 'pipe', 'inherit'],
    shell: true,
  }
);

//==== Transferring chunks from echo.stdout to sort.stdin ====

const echoOut = Readable.toWeb(
  echo.stdout.setEncoding('utf-8'));
const sortIn = Writable.toWeb(sort.stdin);

const sortInWriter = sortIn.getWriter();
try {
  for await (const chunk of echoOut) { // (C)
    await sortInWriter.write(chunk);
  }
} finally {
  sortInWriter.close();
}

//==== Reading sort.stdout ====

const sortOut = Readable.toWeb(
  sort.stdout.setEncoding('utf-8'));
assert.equal(
  await readableStreamToString(sortOut),
  'apple\nbanana\ncherry\n'
);
```

ReadableStreams such as `echoOut` are asynchronously iterable. That's why we can use a `for-await-of` loop to read their *chunks* (the fragments of the streamed data). For more information, see §10 "Using web streams on Node.js".

12.2.6 Handling unsuccessful exits (including errors)

There are three main kinds of unsuccessful exits:

- The child process can't be spawned.
- An error happens in the shell.
- A process is killed.

12.2.6.1 The child process can't be spawned

The following code demonstrates what happens if a child process can't be spawned. In this case, the cause is that the shell's path doesn't point to an executable (line A).

```
import {spawn} from 'node:child_process';

const childProcess = spawn(
  'echo hello',
  {
    stdio: ['inherit', 'inherit', 'pipe'],
    shell: '/bin/does-not-exist', // (A)
  }
);
childProcess.on('error', (err) => { // (B)
  assert.equal(
    err.toString(),
    'Error: spawn /bin/does-not-exist ENOENT'
  );
});
```

This is the first time that we use events to work with child processes. In line B, we register an event listener for the `'error'` event. The child process starts after the current code fragment is finished. That helps prevent race conditions: When we start listening we can be sure that the event hasn't been emitted yet.

12.2.6.2 An error happens in the shell

If the shell code contains an error, we don't get an `'error'` event (line B), we get an `'exit'` event with a non-zero exit code (line A):

```
import {Readable} from 'node:stream';
import {spawn} from 'node:child_process';

const childProcess = spawn(
  'does-not-exist',
  {
    stdio: ['inherit', 'inherit', 'pipe'],
    shell: true,
```

```
    }
  );
  childProcess.on('exit',
    async (exitCode, signalCode) => { // (A)
      assert.equal(exitCode, 127);
      assert.equal(signalCode, null);
      const stderr = Readable.toWeb(
        childProcess.stderr.setEncoding('utf-8'));
      assert.equal(
        await readableStreamToString(stderr),
        '/bin/sh: does-not-exist: command not found\n'
      );
    }
  );
  childProcess.on('error', (err) => { // (B)
    console.error('We never get here!');
  });
```

12.2.6.3 A process is killed

If a process is killed on Unix, the exit code is `null` (line C) and the signal code is a string (line D):

```
import {Readable} from 'node:stream';
import {spawn} from 'node:child_process';

const childProcess = spawn(
  'kill $$', // (A)
  {
    stdio: ['inherit', 'inherit', 'pipe'],
    shell: true,
  }
);
console.log(childProcess.pid); // (B)
childProcess.on('exit', async (exitCode, signalCode) => {
  assert.equal(exitCode, null); // (C)
  assert.equal(signalCode, 'SIGTERM'); // (D)
  const stderr = Readable.toWeb(
    childProcess.stderr.setEncoding('utf-8'));
  assert.equal(
    await readableStreamToString(stderr),
    '' // (E)
  );
});
```

Note that there is no error output (line E).

Instead of the child process killing itself (line A), we could have also paused it for a longer time and killed it manually via the process ID that we logged in line B.

What happens if we kill a child process on Windows?

- `exitCode` is `1`.
- `signalCode` is `null`.

12.2.7 Waiting for the exit of a child process

Sometimes we only want to wait until a command is finished. That can be achieved via events and via Promises.

12.2.7.1 Waiting via events

```
import * as fs from 'node:fs';
import {spawn} from 'node:child_process';

const childProcess = spawn(
  `(echo first && echo second) > tmp-file.txt`,
  {
    shell: true,
    stdio: 'inherit',
  }
);
childProcess.on('exit', (exitCode, signalCode) => { // (A)
  assert.equal(exitCode, 0);
  assert.equal(signalCode, null);
  assert.equal(
    fs.readFileSync('tmp-file.txt', {encoding: 'utf-8'}),
    'first\nsecond\n'
  );
});
```

We are using the standard Node.js event pattern and register a listener for the `'exit'` event (line A).

12.2.7.2 Waiting via Promises

```
import * as fs from 'node:fs';
import {spawn} from 'node:child_process';

const childProcess = spawn(
  `(echo first && echo second) > tmp-file.txt`,
  {
    shell: true,
    stdio: 'inherit',
  }
);

const {exitCode, signalCode} = await onExit(childProcess); // (A)

assert.equal(exitCode, 0);
```

```
assert.equal(signalCode, null);
assert.equal(
  fs.readFileSync('tmp-file.txt', {encoding: 'utf-8'}),
  'first\nsecond\n'
);
```

The helper function `onExit()` that we use in line A, returns a Promise that is fulfilled if an `'exit'` event is emitted:

```
export function onExit(eventEmitter) {
  return new Promise((resolve, reject) => {
    eventEmitter.once('exit', (exitCode, signalCode) => {
      if (exitCode === 0) { // (B)
        resolve({exitCode, signalCode});
      } else {
        reject(new Error(
          `Non-zero exit: code ${exitCode}, signal ${signalCode}`));
      }
    });
    eventEmitter.once('error', (err) => { // (C)
      reject(err);
    });
  });
}
```

If `eventEmitter` fails, the returned Promise is rejected and `await` throws an exception in line A. `onExit()` handles two kinds of failures:

- `exitCode` isn't zero (line B). That happens:
  - If there is a shell error. Then `exitCode` is greater than zero.
  - If the child process is killed on Unix. Then `exitCode` is `null` and `signalCode` is non-null.
    - Killing child process on Windows produces a shell error.
- An `'error'` event is emitted (line C). That happens if the child process can't be spawned.

12.2.8 Terminating child processes

12.2.8.1 Terminating a child process via an AbortController

In this example, we use an AbortController to terminate a shell command:

```
import {spawn} from 'node:child_process';

const abortController = new AbortController(); // (A)

const childProcess = spawn(
  `echo Hello`,
  {
    stdio: 'inherit',
    shell: true,
```

```
    signal: abortController.signal, // (B)
  }
);
childProcess.on('error', (err) => {
  assert.equal(
    err.toString(),
    'AbortError: The operation was aborted'
  );
});
abortController.abort(); // (C)
```

We create an AbortController (line A), pass its signal to `spawn()` (line B), and terminate the shell command via the AbortController (line C).

The child process starts asynchronously (after the current code fragment is executed). That's why we can abort before the process has even started and why we don't see any output in this case.

12.2.8.2 Terminating a child process via `.kill()`

In the next example, we terminate a child process via the method `.kill()` (last line):

```
import {spawn} from 'node:child_process';

const childProcess = spawn(
  `echo Hello`,
  {
    stdio: 'inherit',
    shell: true,
  }
);
childProcess.on('exit', (exitCode, signalCode) => {
  assert.equal(exitCode, null);
  assert.equal(signalCode, 'SIGTERM');
});
childProcess.kill(); // default argument value: 'SIGTERM'
```

Once again, we kill the child process before it has started (asynchronously!) and there is no output.

12.3 Spawning processes synchronously: `spawnSync()`

```
spawnSync(
  command: string,
  args?: Array<string>,
  options?: Object
): Object
```

`spawnSync()` is the synchronous version of `spawn()` – it waits until the child process exits before it synchronously(!) returns an object.

The parameters are mostly the same as those of `spawn()`. `options` has a few additional properties – e.g.:

- `.input: string | TypedArray | DataView`
  If this property exists, its value is sent to the standard input of the child process.
- `.encoding: string` (default: `'buffer'`)
  Specifies the encoding that is used for all standard I/O streams.

The function returns an object. Its most interesting properties are:

- `.stdout: Buffer | string`
  Contains whatever was written to the standard output stream of the child process.
- `.stderr: Buffer | string`
  Contains whatever was written to the standard error stream of the child process.
- `.status: number | null`
  Contains the exit code of the child process or `null`. Either the exit code or the signal code are non-null.
- `.signal: string | null`
  Contains the signal code of the child process or `null`. Either the exit code or the signal code are non-null.
- `.error?: Error`
  This property is only created if spawning didn't work and then contains an Error object.

With the asynchronous `spawn()`, the child process ran concurrently and we could read standard I/O via streams. In contrast, the synchronous `spawnSync()` collects the contents of the streams and returns them to us synchronously (see next subsection).

12.3.1 When is the shell command executed?

When using the synchronous `spawnSync()`, the child process for the command is started synchronously. The following code demonstrates that:

```
import {spawnSync} from 'node:child_process';

spawnSync(
  'echo', ['Command starts'],
  {
    stdio: 'inherit',
    shell: true,
  }
);
console.log('After spawnSync()');
```

This is the output:

```
Command starts
After spawnSync()
```

12.3.2 Reading from stdout

The following code demonstrates how to read standard output:

```
import {spawnSync} from 'node:child_process';

const result = spawnSync(
  `echo rock && echo paper && echo scissors`,
  {
    stdio: ['ignore', 'pipe', 'inherit'], // (A)
    encoding: 'utf-8', // (B)
    shell: true,
  }
);
console.log(result);
assert.equal(
  result.stdout, // (C)
  'rock\npaper\nscissors\n'
);
assert.equal(result.stderr, null); // (D)
```

In line A, we use `options.stdio` to tell `spawnSync()` that we are only interested in standard output. We ignore standard input and pipe standard error to the parent process.

As a consequence, we only get a result property for standard output (line C) and the property for standard error is `null` (line D).

Since we can't access the streams that `spawnSync()` uses internally to handle the standard I/O of the child process, we tell it which encoding to use, via `options.encoding` (line B).

12.3.3 Sending data to the stdin of the child process

We can send data to the standard input stream of a child process via the options property `.input` (line A):

```
import {spawnSync} from 'node:child_process';

const result = spawnSync(
  `sort`,
  {
    stdio: ['pipe', 'pipe', 'inherit'],
    encoding: 'utf-8',
    input: 'Cherry\nApple\nBanana\n', // (A)
  }
);
assert.equal(
  result.stdout,
  'Apple\nBanana\nCherry\n'
);
```

12.3.4 Handling unsuccessful exits (including errors)

There are three main kinds of unsuccessful exits (when the exit code isn't zero):

- The child process can't be spawned.

- An error happens in the shell.
- A process is killed.

12.3.4.1 The child process can't be spawned

If spawning fails, `spawn()` emits an `'error'` event. In contrast, `spawnSync()` sets `result.error` to an error object:

```
import {spawnSync} from 'node:child_process';

const result = spawnSync(
  'echo hello',
  {
    stdio: ['ignore', 'inherit', 'pipe'],
    encoding: 'utf-8',
    shell: '/bin/does-not-exist',
  }
);
assert.equal(
  result.error.toString(),
  'Error: spawnSync /bin/does-not-exist ENOENT'
);
```

12.3.4.2 An error happens in the shell

If an error happens in the shell, the exit code `result.status` is greater than zero and `result.signal` is `null`:

```
import {spawnSync} from 'node:child_process';

const result = spawnSync(
  'does-not-exist',
  {
    stdio: ['ignore', 'inherit', 'pipe'],
    encoding: 'utf-8',
    shell: true,
  }
);
assert.equal(result.status, 127);
assert.equal(result.signal, null);
assert.equal(
  result.stderr, '/bin/sh: does-not-exist: command not found\n'
);
```

12.3.4.3 A process is killed

If the child process is killed on Unix, `result.signal` contains the name of the signal and `result.status` is `null`:

```
import {spawnSync} from 'node:child_process';

const result = spawnSync(
  'kill $$',
  {
    stdio: ['ignore', 'inherit', 'pipe'],
    encoding: 'utf-8',
    shell: true,
  }
);

assert.equal(result.status, null);
assert.equal(result.signal, 'SIGTERM');
assert.equal(result.stderr, ''); // (A)
```

Note that no output was sent to the standard error stream (line A).

If we kill a child process on Windows:

- `result.status` is 1
- `result.signal` is `null`
- `result.stderr` is `''`

12.4 Asynchronous helper functions based on `spawn()`

In this section, we look at two asynchronous functions in module `node:child_process` that are based on `spawn()`:

- `exec()`
- `execFile()`

We ignore `fork()` in this chapter. Quoting the Node.js documentation:

> `fork()` spawns a new Node.js process and invokes a specified module with an IPC communication channel established that allows sending messages between parent and child.

12.4.1 `exec()`

```
exec(
  command: string,
  options?: Object,
  callback?: (error, stdout, stderr) => void
): ChildProcess
```

`exec()` runs a command in a newly spawned shell. The main differences with `spawn()` are:

- In addition to returning a ChildProcess, `exec()` also delivers a result via a callback: Either an error object or the contents of stdout and stderr.
- Causes of errors: child process can't be spawned, shell error, child process killed.

  - In contrast, `spawn()` only emits `'error'` events if the child process can't be spawned. The other two failures are handled via exit codes and (on Unix) signal codes.
- There is no parameter `args`.
- The default for `options.shell` is `true`.

```
import {exec} from 'node:child_process';

const childProcess = exec(
  'echo Hello',
  (error, stdout, stderr) => {
    if (error) {
      console.error('error: ' + error.toString());
      return;
    }
    console.log('stdout: ' + stdout); // 'stdout: Hello\n'
    console.error('stderr: ' + stderr); // 'stderr: '
  }
);
```

`exec()` can be converted to a Promise-based function via `util.promisify()`:

- The ChildProcess becomes a property of the returned Promise.
- The Promise is settled as follows:
  - Fulfillment value: `{stdout, stderr}`
  - Rejection value: same value as parameter `error` of the callback but with two additional properties: `.stdout` and `.stderr`.

```
import * as util from 'node:util';
import * as child_process from 'node:child_process';

const execAsync = util.promisify(child_process.exec);

try {
  const resultPromise = execAsync('echo Hello');
  const {childProcess} = resultPromise;
  const obj = await resultPromise;
  console.log(obj); // { stdout: 'Hello\n', stderr: '' }
} catch (err) {
  console.error(err);
}
```

12.4.2 `execFile()`

```
execFile(file, args?, options?, callback?): ChildProcess
```

Works similarly to `exec()`, with the following differences:

- The parameter `args` is supported.
- The default for `options.shell` is `false`.

Like `exec()`, `execFile()` can be converted to a Promise-based function via `util.promisify()`.

12.5 Synchronous helper functions based on `spawnAsync()`

12.5.1 `execSync()`

```
execSync(
  command: string,
  options?: Object
): Buffer | string
```

`execSync()` runs a command in a new child process and waits synchronously until that process exits. The main differences with `spawnSync()` are:

- Only returns the contents of stdout.
- Three kinds of failures are reported via exceptions: child process can't be spawned, shell error, child process killed.
  - In contrast, the result of `spawnSync()` only has an `.error` property if the child process can't be spawned. The other two failures are handled via exit codes and (on Unix) signal codes.
- There is no parameter `args`.
- The default for `options.shell` is `true`.

```
import {execSync} from 'node:child_process';

try {
  const stdout = execSync('echo Hello');
  console.log('stdout: ' + stdout); // 'stdout: Hello\n'
} catch (err) {
  console.error('Error: ' + err.toString());
}
```

12.5.2 `execFileSync()`

```
execFileSync(file, args?, options?): Buffer | string
```

Works similarly to `execSync()`, with the following differences:

- The parameter `args` is supported.
- The default for `options.shell` is `false`.

12.6 Useful libraries

12.6.1 tinysh: a helper for spawning shell commands

tinysh by Anton Medvedev is a small library that helps with spawning shell commands – e.g.:

```
import sh from 'tinysh';

console.log(sh.ls('-l'));
console.log(sh.cat('README.md'));
```

We can override the default options by using `.call()` to pass an object as `this`:

```
sh.tee.call({input: 'Hello, world!'}, 'file.txt');
```

We can use any property name and tinysh executes the shell command with that name. It achieves that feat via a Proxy. This is a slightly modified version of the actual library:

```
import {execFileSync} from 'node:child_process';
const sh = new Proxy({}, {
  get: (_, bin) => function (...args) { // (A)
    return execFileSync(bin, args,
      {
        encoding: 'utf-8',
        shell: true,
        ...this // (B)
      }
    );
  },
});
```

In line A, we can see that if we get a property whose name is `bin` from `sh`, a function is returned that invokes `execFileSync()` and uses `bin` as the first argument.

Spreading `this` in line B enables us to specify options via `.call()`. The defaults come first, so that they can be overridden via `this`.

12.6.2 node-powershell: executing Windows PowerShell commands via Node.js

Using the library node-powershell on Windows, looks as follows:

```
import { PowerShell } from 'node-powershell';
PowerShell.$`echo "hello from PowerShell"`;
```

12.7 Choosing between the functions of module `'node:child_process'`

General constraints:

- Should other asynchronous tasks run while the command is executed?
  - Use any asynchronous function.
- Do you only execute one command at a time (without async tasks in the background)?
  - Use any synchronous function.
- Do you want to access stdin or stdout of the child process via a stream?
  - Only asynchronous functions give you access to streams: `spawn()` is simpler in this case because it doesn't have a callback that delivers errors and standard I/O content.
- Do you want to capture stdout or stderr in a string?
  - Asynchronous options: `exec()` and `execFile()`

  - Synchronous options: `spawnSync()`, `execSync()`, `execFileSync()`

Asynchronous functions – choosing between `spawn()` and `exec()` or `execFile()`:

- `exec()` and `execFile()` have two benefits:
  - Failures are easier to handle because they are all reported in the same manner – via the first callback parameter.
  - Getting stdout and stderr as strings is easier - due to the callback.
- You can pick `spawn()` if those benefits don't matter to you. Its signature is simpler without the (optional) callback.

Synchronous functions – choosing between `spawnSync()` and `execSync()` or `execFileSync()`:

- `execSync()` and `execFileSync()` have two specialties:
  - They return a string with the content of stdout.
  - Failures are easier to handle because they are all reported in the same manner – via exceptions.
- Pick `spawnSync()` if you need more information than `execSync()` and `execFileSync()` provide via their return values and exceptions.

Choosing between `exec()` and `execFile()` (the same arguments apply to choosing between `execSync()` and `execFileSync()`):

- The default for `options.shell` is `true` in `exec()` but `false` in `execFile()`.
- `execFile()` supports `args`, `exec()` doesn't.

Chapter 13

Where are the remaining chapters?