IN THIS CHAPTER

- Understanding Node.js
- Using Node.js from the Command Line with REPL
- Making modules
- Reading environment variables
- Accepting arguments
- Running Node.js programs
- Invoking the callback pattern

Chapter 1
Node.js Fundamentals

"Complexity that works is built up out of modules that work perfectly, layered one over the other."

—KEVIN KELLY

Browsers aren't the only places where JavaScript code can run. In Book 7, you learn about Node.js, which makes it possible for you to run JavaScript on web servers, in your local development environment, on single-board computers like the Raspberry Pi, and more.

In this chapter, I describe what Node.js is, explain how it works, and tell you what makes it so useful. I also spell out the basics of interacting with Node.js and running Node.js programs.

Learning What Makes Node.js Tick

Node.js is open-source software for running JavaScript outside of web browsers. Created by Ryan Dahl in 2009, it quickly became tremendously popular. Today, Node.js is used on many of the world's largest websites and by nearly every JavaScript programmer. It attracts a giant community of developers: The npm registry (where developers can make their work available to be downloaded and used by other developers) now lists over 1 million Node.js packages.

As a front-end developer (which you are now, if you've read Books 1–6), learning Node.js places you into the rarified — and potentially lucrative — position of being a *full-stack developer,* which is a programmer who works on front-end apps as well as back-end apps. Even if you're interested only in developing user interfaces, knowing how Node.js works can make you a better front-end developer.

Node.js is not a programming language

Every bit of code you write for Node.js is JavaScript code. Node.js contains no special syntax of its own. It does, however, have some conventions and best practices that all Node.js developers follow. You need to learn the standard Node.js conventions and a bit more about servers to be able to program for Node.js. Other than that, if you know JavaScript, you can know Node.js.

Node.js is not a framework

You might often hear programmers who aren't familiar with Node.js talk about the differences between Node.js and tools like ASP.net or Spring. Although ASP.net, Spring, and Node.js are all tools that can be used in the creation of server-side web applications, the similarities between them stop there. ASP.net and Spring are *frameworks* — collections of prebuilt code libraries for creating web applications. Node.js is a platform for server-side JavaScript applications, in the same way that a browser is a platform for client-side JavaScript applications. Comparing a platform with a framework is like comparing a road to a car.

Node.js is a runtime environment

Like any programming language, or any language at all, really, JavaScript is nothing but a syntax. Just as human languages must be in a readable format or spoken to be useful, programming languages must be compiled and run in a *runtime environment.*

Web browsers are runtime environments, too

In Book 2, I tell you about one runtime environment for JavaScript: the web browser. What makes JavaScript so useful in web browsers is that the web browser contains APIs for interacting with networking, storage, your computer's hardware, and more.

However, for safety and security reasons, there are things that JavaScript running in a web browser can't do. For example, a web browser can't read and write to databases on your computer; it can't create, change, or delete files on your computer; and it can't respond to HTTP requests from other web browsers. Instead, JavaScript in a browser runs in a protected environment that's sometimes referred to as a sandbox.

The browser's *sandbox* is a safe place where the browser can create its own little world involving cached files, cookies, and local storage. Though the browser can also access the device hardware outside the sandbox, it must specifically ask for permission from the user (which is why you see alerts from websites asking for permission to access your location or camera or to send you notifications). When you're using a web application running in a browser, you're acting as its parent.

REMEMBER

Because web browsers are client applications that access remote computers that they know nothing about (strangers!), it's beneficial that browsers run JavaScript in a sandbox or else we'd all be getting our hard drives erased or filled up with spam all the time.

Node.js lets JavaScript out of the sandbox

Node.js runs JavaScript directly in a computer's operating system, not in a sandbox. As a result, Node.js has full access to the operating system and to anything that any other installed program can access. Node.js programs can access the hard drive, make network requests, accept requests from the outside world, run other programs on your computer, and access hardware devices connected to the computer where they run. If a browser is like a sandbox, Node.js is like a house, and the programs you write to run in Node.js are fully grown adults.

Why developers need Node.js

Clients aren't useful much without servers. The way the web works is that clients (web browsers) request data from servers and send data to servers. The software on the server that listens for HTTP requests is called an *HTTP server.* HTTP servers may be able to access databases, read and write files, make requests to other servers, and even control your home thermostat, security camera, or toaster, as shown in Figure 1-1.

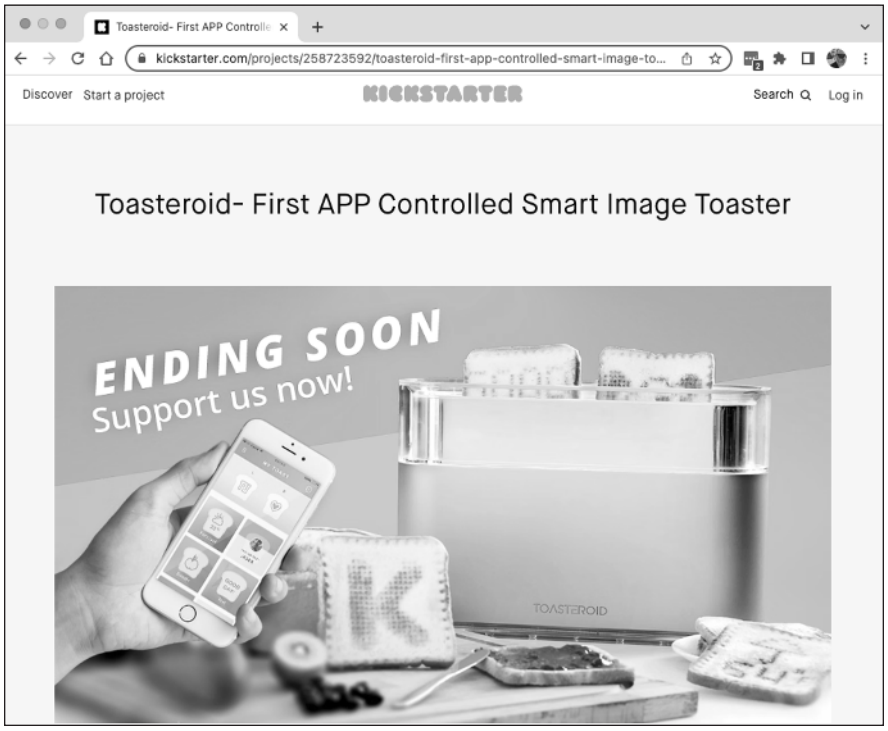

FIGURE 1-1: Unfortunately, Toasteroid never advanced past being a Kickstarter project.

Just as JavaScript interacts with your web browser, web servers interact with databases and toasters using APIs.

Before 2009, nearly all server-side software was written using languages other than JavaScript. What that meant is that if you were a JavaScript programmer who wanted to work on client-side applications as well as on server-side applications, you had to know and program in JavaScript on the client and in a separate language (such as Perl, PHP, Python, Ruby, or Java) for the server.

Learning the Parts of Node.js

Node.js is made up of several parts and several areas of functionality. In terms of the software components that make up Node.js, these are the most important parts:

- **V8:** This is the same JavaScript engine that's used by Google's Chrome browser.

- **libuv:** The libuv library provides support for asynchronous input and output (such as file and network operations) using the underlying operating system Node.js is running on.
- **Node.js bindings:** The bindings are methods that translate between JavaScript that can run in the V8 engine and the functions of libuv, which are written in C.
- **APIs:** Also known as the NodeJS standard library, the APIs are JavaScript interfaces that allow your programs to interact with the outside world.

Figure 1-2 shows a diagram of how these parts, and a few others, fit together.

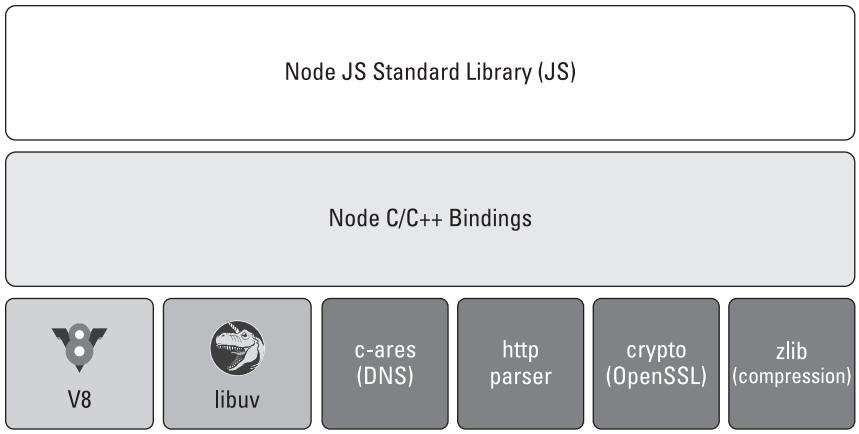

FIGURE 1-2: Illustrating the parts of Node.js.

The V8 engine

The JavaScript engine at the core of Node.js is the same one that parses and runs JavaScript code in Chrome. As a result of using the V8 engine, Node.js benefits from the constant innovations and improvements that are being made to V8 to make browsers faster. Using V8 also means that Node.js can use all the same features of JavaScript that the latest version of Google Chrome can use.

REMEMBER

The V8 JavaScript engine is covered in Chapter 1 of Book 2.

libuv

JavaScript uses events and callback functions to trigger asynchronous operations such as file operations, network requests, and database operations. The job of

libuv is to handle these asynchronous operations in a cross-platform way, which makes it possible for Node.js to run the same way on macOS, Windows, Linux, and other operating systems.

REMEMBER

The main function of libuv, the event loop, is covered in Chapter 10 of Book 1.

Node.js bindings

The Node.js bindings translate between code written in JavaScript (namely, the NodeJS APIs) and programs written in C or C++. For example, V8 by itself doesn't know how to access databases. However, libraries do exist for working with databases, though they're typically written in the native language of the operating system. As a result, a binding is necessary to bridge the communication gap.

The Node.js standard library

The Node.js standard library is a set of modules built into Node.js, which are known as the *core modules.* Figure 1-3 shows the names of the Node.js core modules. In the following section, I introduce a few of the Node.js core modules, and then I follow up in the rest of this book with detailed information about some of the most important ones.

assert	buffer	child_process	console	cluster
crypto	dgram	dns	events	fs
http	http2	https	net	os
path	perf_hooks	process	querystring	readline
repl	stream	string_decoder	timers	tls
tty	url	util	v8	vm
wasi	worker	zlib		

FIGURE 1-3: The Node.js core modules.

Introducing the Node.js Core Modules

The core modules are compiled files that live in the `lib` folder inside your global Node.js installation. Most Node.js programs make use of at least one of the core modules, which are installed as part of every Node.js installation.

Core modules can be imported using CommonJS or ECMAScript module syntax. These are the most important built-in modules to be aware of as you're starting to learn Node.js:

- **http** creates an HTTP server.
- **stream** provides tools for creating event-based classes for managing data.
- **assert** contains functions for making assertions for testing.
- **fs** handles file system operations.
- **path** deals with file paths.
- **process** provides information about the current Node.js process.
- **os** provides information about the operating system running Node.js.
- **querystring** contains utilities for working with URL query strings.
- **url** includes utilities for working with URLs.

Listing 1-1 shows how to import and use the os module using CommonJS.

LISTING 1-1: **Importing and Using a Core Module**

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

console.log('Host: ' + os.hostname());
console.log('OS: ' + os.type());
console.log('OS Version: ' + os.release());
console.log('Total Memory: ' + os.totalmem() + ' bytes');
console.log('Free Memory: ' + os.freemem() + ' bytes');
```

In this example, the os module is imported and made available as an object named `os`. The various methods made available by the os module can then be used to print information about the current operating system to the console.

You can run the program in Listing 1-1 by saving it in a file named `Listing0101.js` (for example) and then entering **node Listing0101** into the terminal.

Listing 1-2 shows how to import and use both the fs and http modules using ECMAScript module syntax and how to serve a file in response to a request.

LISTING 1-2: **Importing a Core Module Using ECMAScript Modules**

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

http.createServer((req, res) => {
  fs.readFile('importantInfo.html', function (err, data) {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.write(data);
    res.end();
  });
}).listen(8080);
```

To run the program in Listing 1-2, save it in a file, make sure there's a file named `importantInfo.html` in the same directory as the program file, and enter **node Listing0102.mjs** into your terminal.

Recognizing What Node.js Is Good For

Node.js's ability to handle many operations quickly is legendary. A server running in Node.js can handle thousands of concurrent requests per second. In fact, Node.js powers the back end of many of the largest websites, including, Netflix, PayPal, LinkedIn, Walmart, and GoDaddy. If it's good enough for them, you can be certain that it's good enough for you.

Why is Node.js so fast?

In Chapter 10 of Book 1, I tell you that JavaScript is single-threaded and that it uses an event loop to spawn asynchronous processes and to respond to messages. Other languages, such as Java, are multithreaded. They gain more power by running tasks in parallel.

Although it would seem that having only one thread would put JavaScript at a disadvantage, being single-threaded turns out to be ideal for servers. This is because most of what a server does is asynchronous — including network operations, file operations, and database operations.

In Node.js, asynchronous operations are always non-blocking. What this means is that, as far as Node.js is concerned, making a complex database request is no more difficult than running any other statement. The event loop hands off the

asynchronous task to the operating system's native API and sends the results back to your JavaScript program when it's complete. What makes Node.js so fast is that it doesn't wait. This is called *non-blocking.*

You can think of non-blocking code in the following way: Whenever you order a meal at a restaurant, the server takes your order and relays it to the kitchen. Once the order is delivered to the kitchen, the server doesn't wait for your order to be filled so that they can bring it to you — they run off and take other orders and deliver those to the kitchen. When your order is ready, the server picks it up and delivers it to you.

A blocking restaurant, in which the server must wait for each order to be completed before taking another one, would be extremely inefficient. The only way to make it more efficient would be to have multiple servers, which would be much more expensive.

What is Node.js not good at?

Node.js is not a good fit for certain types of tasks. In particular, Node.js is not a good choice for tasks that are computationally intensive. This is because tasks that aren't asynchronous, such as performing math operations, block Node.js.

To use the restaurant analogy from the preceding section: If a server in the non-blocking restaurant is also required to wash the dishes by hand (a task that can't be done asynchronously), the efficiency of the system goes out the window. The server would spend all their time between orders washing dishes and then they'd be less available to take orders.

Working with Node.js

Although Node.js was originally designed to run on web servers, it has also found a home in running tools used by software developers. If you do any modern JavaScript development, you're most likely making use of Node.js all the time.

In this book, however, I focus on showing you how to write server-side applications with Node.js.

Writing a Node.js program

Just like JavaScript programs you create to run in web browsers, you can write a program to run in Node.js using nothing but a text editor. However, code editors such as VS Code will make your job much easier.

Follow these steps to write your own web server with Node.js:

1. **Create a new file in VS Code, named** server.js**.**
2. **Enter the code from Listing 1-3 into the file and save it.**
3. **Open a terminal window and start the server by running the following command:**

```
node server
```

4. **Go to `http://localhost:3000` in your web browser.**

   You see the text `Hello World!` in your browser, as shown in Figure 1-4.

5. **When you're done checking out your server, press Ctrl+C in the terminal to stop the server.**

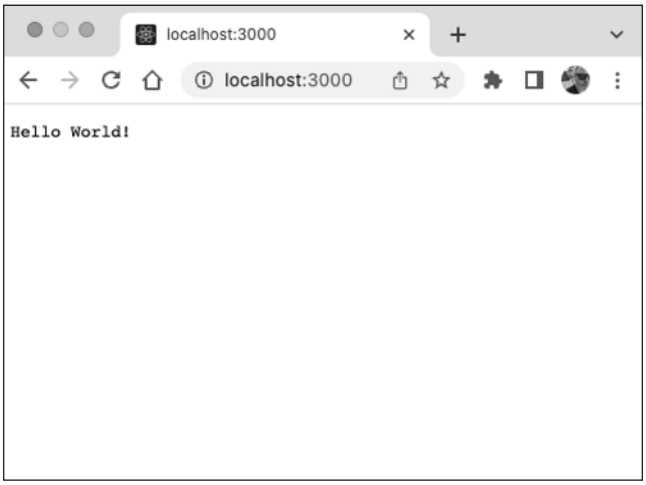

FIGURE 1-4: Running a simple Node.js web server.

LISTING 1-3: **Your First Node.js Web Server**

```
const http = require('http');
const server = http.createServer(function (req, res) {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello World!');
});
```

```
server.listen(3000);
console.log('Server running at http://localhost:3000/');
```

Monitoring your script

Node.js doesn't watch your files for changes. What this means is that if you make changes to your server while it's running, those changes won't show up in the currently running process. To see your changes, you need to stop the server and restart it.

Having to continually stop and restart your program to see your changes would be irritating, however. To solve this problem, you can use the Node.js package called nodemon, which wraps around your program in Node.js and automatically restarts it when you change files in your Node application.

Follow these steps to get started with using nodemon:

1. **Install nodemon globally by entering the following command into the terminal:**

```
npm install -g nodemon
```

WARNING

If the permissions on your computer aren't configured correctly, or if you don't have administrative access to your computer, you may get an error when you try to install a package globally. This is most common on macOS and Linux. Should this happen to you, preface the `npm install` command with `sudo`. The `sudo` command asks you for a password (the same password you log in to your computer with) and allows the global installation to proceed.

2. **Once you've installed nodemon globally, you can run any Node.js program by using `nodemon` instead of `node`. For example, enter the following command to start up the simple Node.js server from Listing 1-3:**

```
nodemon server
```

TIP

If nodemon doesn't work after you install it globally, you may have to close and reopen the terminal window or open a new terminal window.

3. **With your program running in nodemon, try changing the text that it returns from `Hello World!` to `Hello Node!`.**

TIP

On Windows, you may get an error in the terminal saying that nodemon can't be run because running scripts is disabled. If you search for the exact error message that you get on Google, you'll find an article that will tell you how to solve the problem.

When you refresh your browser window, you should now see the new text.

Running a code on the command line

The one essential tool for interacting with Node.js is a command-line interface. On macOS, this is known as Terminal. On Windows, it's `CMD.exe` or another terminal application. On Linux, it's the terminal.

VSCode has an integrated terminal window that gives you access to an underlying command-line interface on your computer. I used this integrated terminal in earlier parts of this book, and I'll continue to use it in this one.

In a web browser, you run a JavaScript program by opening it in a web browser. In Node.js, you run JavaScript from the command line. As you saw in the previous section, to run the program in Node.js, you type `node` or `nodemon` followed by the name of the program you want to run.

Since Node.js runs only JavaScript programs, it automatically infers that the file extension of your program is `.js`. Because of this, typing the `.js` extension when running a Node.js program is optional.

Using REPL

The command-line interface can be used to manage Node.js packages and run Node.js applications. It can also be used to gain direct access to Node.js for writing and running simple JavaScript code within Node.js. The interface where you can run code directly within Node.js without making files and modules is called REPL.

REPL stands for read-evaluate-print loop. REPL is the Node.js equivalent to the JavaScript console in a browser. Just as the browser console gives you access to the objects created by the browser (such as the window object and the navigator object), REPL gives you access to the global objects created by Node.

To access REPL, enter **node** into a terminal window. You see a welcome message and a REPL prompt like the one shown in Figure 1-5.

Playing with the Node.js REPL

After you're in REPL, try running some JavaScript and calling some Node.js functions. Follow these steps:

1. **Start by entering a simple JavaScript expression, like this:**

```
45+34
```

FIGURE 1-5: Starting REPL.

Node.js returns the result of the expression. There's nothing too interesting about that, of course.

2. **You can also call functions from within REPL — for example, try entering the following statement:**

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

Just as the browser's window object has a console object containing a `log()` method, Node.js has a global console object containing a number of methods that can be useful for debugging your programs, including these:

- `console.log()` outputs a standard debug message to the console.
- `console.error()` outputs an error to the console.
- `console.warn()` is an alias for `console.error()`.
- `console.dir()` logs the properties of an object to the console.

3. **Now that you know about Node.js's `console.dir()` method, use it to inspect the console object itself:**

```
console.dir(console);
```

Figure 1-6 shows the result of passing the console object to `console.dir()`.

TIP

Using `console.dir()` to inspect an object is a useful debugging tool to use within a Node.js program. When you're in REPL, you can just type the name of the object to print out its properties.

FIGURE 1-6: Inspecting the console object.

4. **You can create variables in the REPL, which will persist as long as the current REPL session is active:**

```
const myObject = {prop1:'test',prop2:'sandwich'};
```

When you create a new variable (or do any operation that has no return value) REPL returns `undefined`.

5. **Type the name of your new variable to make sure that it has been created:**

```
myObject
```

6. **Another useful built-in object in Node.js is the global object. Inspect the properties of global:**

```
global
```

You can use the global methods and properties without prefacing them with `global`.

As of Version 18 of Node.js, the global object contains a `fetch()` method that works like the browser's `window.fetch()` method. The `fetch()` method returns a Promise.

7. **Try out the fetch method by running the following command in REPL:**

```
fetch('https://www.example.com').then(
  (response)=>console.dir(response.body)
);
```

When you run the preceding snippet, Node.js does an HTTP GET request for the URL you specify and write the body of the response to the console. But, there's something strange about the response, as shown in Figure 1-7.

FIGURE 1-7: The body of an HTTP request from Node.js.

REMEMBER

The body of the response is a ReadableStream object. I tell you about streams in Chapter 2 of Book 7, including how to convert a ReadableStream object into something you can read and use in your programs.

TIP

After you execute the `fetch()` method mentioned in the preceding step list, you can press **Enter** to return to the REPL prompt.

Working with REPL commands

REPL doesn't provide you with a lot of built-in help, and it can be confusing to work in the REPL as a result. Knowing a few of the built-in commands and keyboard combinations you can use can make things much easier.

REPL has several commands that are available using a period followed by a keyword. The first to learn is `.help`. Entering `.help` at the REPL prompt brings up a list of the other commands you can use, as shown in Figure 1-8.

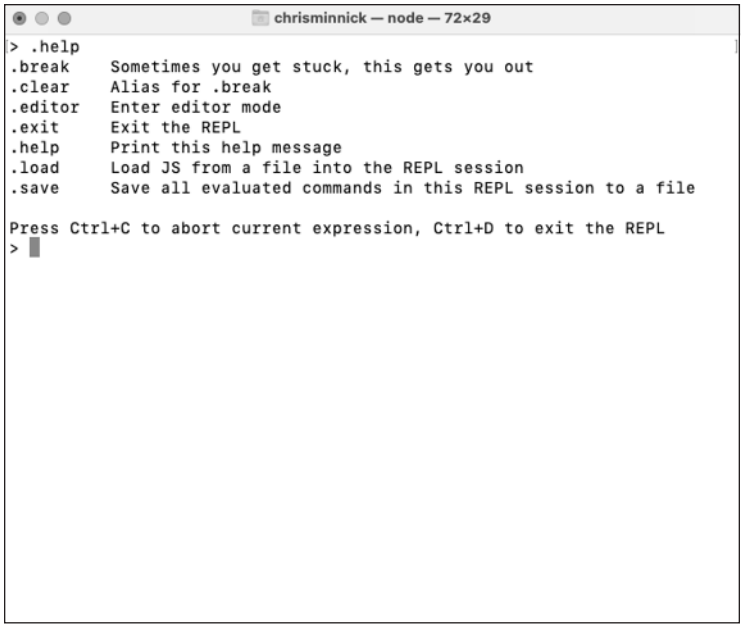

FIGURE 1-8: Finding out REPL's commands with the `.help` command.

Follow these steps to see how to save a REPL session and load JavaScript into the REPL:

1. **In a REPL session, run a couple of expressions to ensure that your current session has some data.**
2. **Enter** .save ./myREPL.js**.**

   The string after the `.save` command is the path and filename to use. You should see a message that the session was saved to your file.
3. **Quit the REPL by pressing Ctrl+D.**

   You return to the normal terminal.
4. **Enter** ls **to see the files in the current directory.**

   One of those files should have the same name as the one you just saved from REPL.

5. **Type** cat **followed by the name of your file.**

   Your saved REPL session will be output to the terminal.

6. **Start the REPL again:**

```
node
```

7. **Load your previous REPL session using the `.load` command:**

```
.load ./myREPL.js
```

Making and Using Node.js Modules

Each file you create in Node.js is treated as a separate module. Node.js programs are highly modular, and building Node.js programs involves importing modules from libraries to create your own modules, which you can then import and use in other modules.

Node.js supports JavaScript modules (also known as *ECMAScript* modules or just *ES* modules), using the same syntax I talk about in Chapter 12 of Book 1. However, many Node.js developers still use the original module syntax of Node.js, called CommonJS.

REMEMBER

You've seen CommonJS modules elsewhere in this book. In particular, the tools such as ESLint and Webpack that I describe in Book 6 make use of CommonJS modules.

Using CommonJS

To create a CommonJS module, use the `module.exports` property. Listing 1-4 shows the contents of a file, which I've named `metricConversions.js`; it exports several functions for converting imperial measurements to metric.

LISTING 1-4: **A File That Exports Several Modules**

```
//function for converting inches to centimeters
exports.inchesToCentimeters = (inches) => {
  return inches * 2.54;
};
//function for converting gallons to liters
exports.gallonsToLiters = (gallons) => {
  return gallons * 3.78541;
};
//function for converting pounds to kilograms
exports.poundsToKilograms = (pounds) => {
  return pounds * 0.453592;
};
//function for converting miles to kilometers
exports.milesToKilometers = (miles) => {
  return miles * 1.60934;
};
```

To import a module into another file, use the `require()` function. Listing 1-5 shows a module that imports and uses the exported modules from Listing 1-4.

LISTING 1-5: **Importing Modules**

```
const metricConversions = require('./metricConversions');

const convertToMetric = (value, unit) => {
  switch (unit) {
    case 'in':
      return metricConversions.inchesToCentimeters(value);
    case 'gl':
      return metricConversions.gallonsToLiters(value);
    case 'lb':
      return metricConversions.poundsToKilograms(value);
    case 'mi':
      return metricConversions.milesToKilometers(value);
    default:
      return value;
  }
};
console.log(`One inch is ${convertToMetric(1, 'in')}
  centimeters.`);
```

Using ES modules

Node.js modules can also be created and imported using the standard JavaScript module syntax. Because CommonJS is the default module format in Node.js (as of this writing), if you want to use JavaScript modules instead, you need to enable them.

To enable JavaScript modules, you can either

- Give the module a `.mjs` extension.
- Add `"type": "module"` as a top-level variable in `package.json` in the module's nearest parent folder.

Listing 1-6 shows an example of creating JavaScript modules.

LISTING 1-6: Creating JavaScript Modules

```
export function triangleArea(base, height) {
  return (base * height) / 2;
}

export function trianglePerimeter(side1, side2, side3) {
  return side1 + side2 + side3;
}

export function triangleHypotenuse(side1, side2) {
  return Math.sqrt(side1 * side1 + side2 * side2);
}

export function triangleLeg(hypotenuse, side) {
  return Math.sqrt(hypotenuse * hypotenuse - side * side);
}

export default {
  triangleArea,
  trianglePerimeter,
  triangleHypotenuse,
  triangleLeg
};
```

In the preceding listing, you export each function using a named export, and you provide a default export. This gives anyone who wants to make use of these modules maximum flexibility for how they'll import them.

Listing 1-7 shows how you can import and use JavaScript modules.

LISTING 1-7: Using JavaScript Modules

```
import { triangleArea, trianglePerimeter } from './Listing070106.mjs';

function triangleMaker(size) {
  const area = triangleArea(size, size);
  const perimeter = trianglePerimeter(size, size, size);

  const triangle = [];
  for (let i = 0; i < size; i++) {
    const row = [];
    for (let j = 0; j < size; j++) {
      if (j < size - i - 1) {
        row.push('  ');
      } else {
        row.push(' *  ');
      }
    }
    triangle.push(row.join(''));
  }
  return {
    area,
    perimeter,
    triangle,
  };
}

const { area, perimeter, triangle } = triangleMaker(18);
console.log(`Area: ${area}`);
console.log(`Perimeter: ${perimeter}`);
console.log(triangle.join('\n'));
```

To run the program in Listing 1-7, the code from both Listing 1-6 and Listing 1-7 must be saved in files named with `.mjs` extensions. Also note that in order to import modules from the file created from Listing 1-6, you must include the `.mjs` extension in the filename inside the import statement. If you don't use the `.mjs` extension,

Node.js assumes that the module should be loaded using CommonJS and you'll get an error. The same thing goes for running the code from Listing 1-7 in the console: Type the full name of the file to run it, like this:

```
node Listing070107.mjs
```

Figure 1-9 shows the output from running Listing 1-7 in the console.

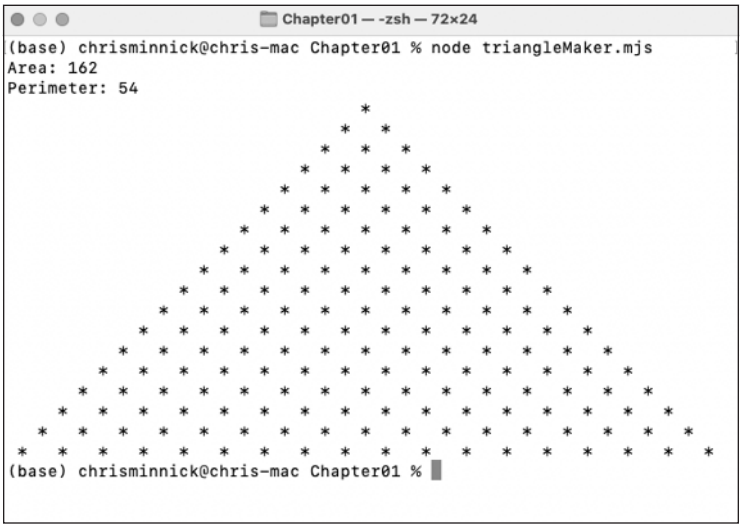

FIGURE 1-9: Drawing a triangle in the console.

Setting the module type in package.json

Setting the module type in `package.json` requires the Node.js modules you create to be part of a package, but the additional steps involved in creating a package will save you from having to remember to name your files with `.mjs`. Follow these steps to create a package and set the module type:

1. **Open a terminal in a directory containing Node.js modules, or create a new directory.**
2. **Enter** npm init **into the terminal window.**
3. **Answer the questions presented to you by the `npm init` script, or just press Enter for each question to accept the default values.**
4. **Add a new property to the top level of the JSON object in `package.json` named `type` and set its value to** "module"**.**

PACKAGES VERSUS MODULES

In Node, a module is a single JavaScript file or a directory that has some reusable functionality. Modules can be imported into other programs, which can use their functionality. A package is a file or directory that's described by a `package.json` file. Packages can be published to the npm registry.

Remember: Packages are generally made up of one or more modules, but not all modules are packages.

Listing 1-8 shows an example of a `package.json` file with ECMAScript modules support enabled.

LISTING 1-8: **Package.json with ECMAScript Modules Enabled**

```
{
  "name": "chapter01",
  "version": "1.0.0",
  "type": "module",
  "description": "",
  "main": "main.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

With ECMAScript modules enabled in `package.json`, you can simply name your Node.js files with a `.js` extension and import and run ECMAScript modules without having to type the file extension.

Getting Data to Node Modules

Many Node.js programs require or allow configuration arguments to be passed to them when they start up. You see this in many of the Node.js packages that I show you how to install and use for front-end development. For example, the npm program accepts many parameters, including the names of subcommands to

run (such as `npm run` and `npm install`) and options preceded by two hyphens, `--`, which are called *flags*, such as `--save-dev`.

You can provide input to Node.js programs in two ways upon execution: environment variables and passing arguments.

Environment variables

Node.js programs often need access to information about themselves and about where they're running. For example, a program that's running on a development machine will need to be configured differently from a version of the same program running in production. The global process object contains a wealth of information about the current Node.js process.

To view the process object, start up the REPL:

```
node
```

Once inside the REPL, enter **process**, and the process object will be returned, as shown in Figure 1-10. The output of the process object is much too long to fit into one figure, but you'll get the idea of it from the snippet shown in the figure.

FIGURE 1-10: Viewing the process object.

In addition to information about the node program and the computer, the process object also contains an object called `env`.

The `env` object contains environment variables. *Environment variables* are variables that reside in the operating system or the container of the running application. By default, the `env` object contains variables such as the username that's running the process, the path to the running program, and the `PATH` variable, which lists directories on the computer where executable programs are kept.

Setting environment variables from the command line

You can pass environment variables to a program when you start it by putting them before the command to run your program, like this:

```
USER_NAME=cminnick PASSWORD=supersecret node app.js
```

Setting environment variables with .env

You can also set environment variables by creating a file named `.env` at the root of your project directory. When working in a development environment, the `.env` file is commonly used to store configuration options, such as API keys and other details about your development environment.

WARNING

Do not store your `.env` file into your code repository. Instead, add `.env` to the `.gitignore` file and create a file, named something like `example.env`, that you commit to your repository (with no secret values set, of course). The `example.env` file can be used by any other developers who will be working on the code to create their own `.env` file.

Listing 1-9 shows an example of a `.env` file.

LISTING 1-9: A Sample .env File

```
API_HOST=api.openweathermap.org
API_KEY=YOUR_API_KEY
API_PATH=/data/2.5/weather
```

Before you can use the `.env` file, you need to import it. The library you can use to do that is called `dotenv`. Follow these steps to create a node package, install `dotenv`, and use it in a program:

1. **Initialize a node package and choose all the default settings:**

```
npm init –yes
```

2. **Install dotenv:**

```
npm i dotenv
```

3. **Make a new file named `app.mjs` and import dotenv:**

```
import dotenv from 'dotenv';
```

4. **Call the dotenv's `config()` method at the beginning of `app.mjs`:**

```
dotenv.config()
```

Listing 1-10 shows an example of a complete program that uses `.env`.

LISTING 1-10: **Using Environment Variables**

```
import dotenv from 'dotenv';
dotenv.config();

export function getCurrentWeather(city) {
  const apiKey = process.env.API_KEY;
  const apiHost = process.env.API_HOST;
  const apiPath = process.env.API_PATH;
  const apiUrl = `https://${apiHost}${apiPath}?q=${city}&appid=${a
   piKey}`;
  return fetch(apiUrl)
    .then((response) => response.json())
    .then((data) => {
      return {
        data: data,
      };
    });
}
getCurrentWeather('London').then((weather) => {
  console.log(weather);
});
```

Passing arguments

You can pass any number of arguments to a node program after the name of the program. These arguments must be separated by spaces — for example:

```
node myProgram.mjs oneThing 'something else' thirdThing
```

Arguments you pass to a node program become elements in the `process.argv` array. The `process.argv` array always has at least two elements:

- `process.argv[0]` contains the location of node.
- `process.argv[1]` contains the location of the JavaScript file.

The arguments that you pass to the program will be in order starting with `process.argv[2]`. Because the arguments you need start with the third element, it's common to declare a new constant in a file that receives arguments by removing the first two elements from the `argv` array, like this:

```
const args = process.argv.slice(2);
```

Node's Callback Pattern

One common pattern for invoking asynchronous APIs in Node.js's core modules and in most modules created by the community of Node.js developers is the callback pattern.

In the callback pattern, an asynchronous method always takes a callback function as its last argument. This function is called when the asynchronous operation is completed. The callback function takes the place of the `return` statement in the function, as in the following simple (synchronous) example:

```
function addNumbersSync(number1,number2,callback) {
  callback(number1 + number2);
}
```

To invoke this function, you pass two numbers followed by a function that indicates what to do with the result, like this:

```
addNumbersSync(2,2, function(result) {
  console.log(`The result is ${result}`);
});
```

In a Node.js module that performs asynchronous tasks, the callback function is used to pass along the results of the asynchronous tasks to the next step when it finishes running, as in the following example:

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

fs.readFile('someTextFile.txt', function (err, data) {
  if (err) {
    console.log(err);
  } else {
    console.log(data.toString());
  }
});
```

In the preceding example, the fs module's `readFile()` method reads the contents of a text file. When the reading of the file is completed, it passes the data from the file as the second argument to the callback function supplied to it. If the `readFile()` method returns an error, that is passed as the first argument to the callback function.

The callback convention is the original way that Node.js modules supported event-based asynchronous code. When Promises became a popular way to abstract callbacks to create less-confusing code, support for them was built into Node.js. Today, many Node.js functions return both a promise and a callback, which gives developers a choice of whether to use the old callback convention or the more up-to-date Promises techniques.

For example, the following is a function that supports the callback pattern:

```
function asyncTask(param1, param2, callback) {
  // do something asynchronous
  if (error) {
    return callback(new Error("oops. There's been an error"));
  }
  // do something else if there's no error
  callback(null, result);
}
```

To invoke this function, pass it the arguments specified by its parameters, followed by a callback function, like this:

```
asyncTask ( argument1, argument2, function ( err,
   returnValue ) {
```

```
  // insert the code to run after the async task is done.
});
```

Here's how the preceding example can be modified to support promises:

```
function asyncTask(param1, param2) {
  // do something asynchronous
  if (error) {
    return Promise.reject(new Error("oops. There's been an
   error"));
  }
  // do something else if there's no error
  return Promise.resolve(result);
}
```

Finally, Listing 1-11 shows how to write the function so that it can be used with a callback or with Promises.

LISTING 1-11: Supporting Both Callbacks and Promises

```
function asyncTask(param1, param2, callback) {
  // do something asynchronous
  if (error) {
    // if a callback is passed, call it with the error
    if (callback) {
      callback(new Error("oops. There's been an error"));
    }
    // otherwise, return a rejected promise
    return Promise.reject(new Error("oops. There's been an
  error"));
  }

  // do something else if there's no error

  // if a callback is passed, call it with the result
  if (callback) {
    callback(null, result);
  }
  // otherwise, return a resolved promise
  return Promise.resolve(result);
}
```