

Pragmatic NodeJS

Richard Krasso

First Edition

Table of Contents

<i>Chapter 1. Diving into Node.js: An Introduction</i>	<i>4</i>
Chapter Overview	4
Learning Objectives	4
Introduction to Node.js	4
Node.js Installation	5
Working with the CLI	8
Node Package Manager (NPM)	13
package.json	17
package-lock.json	22
Programming Exercises	23
<i>Chapter 2. Module System and Diagnostics</i>	<i>24</i>
Chapter Overview	24
Learning Objectives	24
Introduction to Node.js Module System	25
CommonJS	25
ES Modules	28
Debugging and Troubleshooting	31
Programming Exercises	35
<i>Chapter 3. Error Handling and Unit Testing</i>	<i>36</i>
Chapter Overview	36
Learning Objectives	37
Introduction to Test-Driven Development (TDD)	37
Unit Testing with Node.js Assert Library	39
Error Handling in Node.js	49
Introduction to Jest and Test Runners	53
Introduction to Test Coverage	60

Programming Exercises	64
Chapter 4. Node Event System	66
Chapter Overview	66
Learning Objectives	66
Introduction to Node.js Event-Driven Architecture	66
Understanding the EventEmitter API	67
Creating Custom Events with EventEmitter	70
Error Handling with Events	74
TDD with EventEmitter	76
Programming Exercises	83
Chapter 5. Process and OS	85
Chapter Overview	85
Learning Objectives	85
Introduction to Processes in Node.js	85
Understanding Process Events	91
Introduction to the OS Module	96
Event Loop and Process.nextTick	98
TDD with Process Events	103
Programming Exercises	113
Chapter 6. HTTP	114
Chapter Overview	114
Learning Objectives	114
Introduction to HTTP and HTTP Methods	115
Creating a Basic HTTP Server with Node.js	116
Working with HTTP Requests and Responses	121
Routing in Node.js HTTP Server	126
TDD with an HTTP Server	129
Programming Exercises	134
Chapter 7. Buffers and Streams	135
Chapter Overview	135
Learning Objectives	135
Introduction to Buffers in Node.js	135
Understanding Streams in Node.js	142

Reading, Writing, and Transforming Data with Streams	148
Piping and Chaining Streams	151
TDD and Streams	155
Programming Exercises	159
Chapter 8. File System	160
Chapter Overview	160
Learning Objectives	160
Introduction to the File System (fs) Module	161
Reading and Writing Files	164
Working with Directories	168
Streams and the File System	173
TDD with Reading and Writing Files	176
Programming Exercises	180
Chapter 9. Child Processes	182
Chapter Overview	182
Learning Objectives	182
Introduction to Child Processes in Node.js	182
Spawning Child Processes	184
Executing Shell Commands with Child Processes	189
Error Handling in Child Processes	191
TDD with Child Processes	192
Programming Exercises	201

Chapter 1. Diving into Node.js: An Introduction

Chapter Overview

Node.js is one of the most powerful JavaScript engines available today, revolutionizing web development by enabling the use of JavaScript on the server side. In this chapter, we will examine the key features of Node.js, starting with the command-line interface (CLI). The CLI is an essential tool for Node.js development, allowing developers to interact with their operating system and execute scripts directly from the terminal. It is a crucial skill for any Node.js developer. Next, we will dive into the Node Package Manager (npm), which is a vital tool for managing dependencies in Node.js projects.

Learning Objectives

By the end of this chapter, you should be able to:

- List the differences between the package.json and package-lock.json files.
- Explain npm.
- Experiment with third-party npm packages.
- Build a Node.js CLI program.

Introduction to Node.js

Node.js is an open-source, cross platform, JavaScript runtime environment built on top of Chrome's V8 JavaScript engine. Node.js allows developers to write code outside of a web browser. Server-side scripting is a technique utilized in web development that involves executing scripts on a web server to create a customized response to a user's request from a website. That is, before sending content to the user's web browser, the server executes the script. To elaborate, when a user requests a webpage, the server runs a script to gather data, possibly from a database, and creates HTML, CSS, and JavaScript, which are subsequently sent to the user's browser to be rendered. This differs from client-side scripting, where scripts, typically written in JavaScript, are executed directly in the user's browser after the page has been delivered.

Server-side scripting is used for generating dynamic web content, where the page changes depending on the user's input, access rights, and other factors. It is also used to handle form submissions, connect and interact with databases, and manage user sessions and cookies.

To understand server-side scripting consider the following, you walk into a bank to withdraw money, you cannot just walk into the vault to grab the money yourself. Instead, you give your request to a bank teller. The teller then goes into the vault (the server), gets the money (the data), and brings it back to you. In this scenario, you're like the user's web browser, the bank teller is the server-side script, and the vault is the server where the resource is being stored. Just like how the bank teller checks your account details and

verifies your identity before giving you the money, the server-side script performs operations like checking your identity, whether you have permission to access a resource, and retrieving information from a database before sending a response to the web browser. Think of server-side scripts as a peanut butter between two pieces of bread. It is an intermediary entity used to facilitate communications between a user's web browser and a company's web server. In the next section we will discuss how to install Node.js through a package manager.

Node.js Installation

There are several different ways to install Node.js. You can download and install Node.js directly through their [website](#) or through a package manager. The best approach is debatable, but there are several advantages of using a package manager:

- Handles multiple versions
- Allows easy swapping between versions
- Great for testing global npm packages
- Used heavily in enterprise development

Using a package manager is the optimal choice for working with Node.js because it allows us to easily switch between different Node.js versions. Why is this important? It is important because, not every project uses the latest version of Node.js. In fact, in industry, most projects “lag” severely behind the latest version. Using a package manager is the best way to work with multiple Node.js versions and environments. Not mention, Node.js follows a 6-month release cycle. As of this writing, the latest “Long Term Support” version of Node.js is 20.10.0. Even number versions move to Active LTS status and should be used by the general public. That means, the next upgradable version is Node.js 22, which is scheduled to be released in the summer of 2024 and becoming Active LTS in October of 2024. Given the rapid release of versions, it is very likely, by the time you read this textbook, there will be a new LTS version of Node.js. Imagine if you worked on a Node.js team for several years. How many versions of Node.js would have been released? How would you be able to make code changes to projects using an older version of Node.js? Uninstall and reinstall an older version of Node.js each time you had to work on an older solution?

Hopefully, you can acknowledge, the amount of time wasted and effort associated with this process would not be advisable. However, with a package manager, you can easily switch to an older version of Node.js with minimal effort. For this textbook we will be using the latest LTS version of Node.js, which is 20. Any other version (higher or lower) will not be supported in this textbook. There are two options for using a package manager to install Node.js. If you are on a Windows machine you use NVS (Node Version System). If you are on a Linux or macOS machine you use NVM (Node Version Manager). Instructions on how to install either package manager is provided through their GitHub repository.

- [Node Version Switcher \(NVS\)](#)

- [Node Version Manager \(NVM\)](#)

The preceding instructions have been taken directly from each of their corresponding setup instructions.

NVS (Windows only)

If you have a Windows machine, use the following instructions; otherwise, skip to the next section (nvm).

- Install NVS using their [MSI installation file](#).
- Once NVS is installed, you will need to open a new command prompt, PowerShell, or terminal window (VS Code) to access the CLI tool.

NVS Basic Commands ([source](#)):

Command	Description
<code>nvs help <command></code>	Get detailed help for a command
<code>nvs --version</code>	Display the NVS tool version
<code>nvs add [version]</code>	Download and extract a node version
<code>nvs rm <version></code>	Remove a node version
<code>nvs use [version]</code>	Use a node version in the current shell
<code>nvs ls [filter]</code>	List local node version

Given the above basic commands, to install the latest LTS (20.10.0) of Node.js, you would use the following command

```
nvs add 20
```

Alternatively, you could simply enter `nvs` from the CLI to open the interactive menu. In the menu, select node and then the version you want to install.

NVM (Linux and macOS only)

If you have a Linux or macOS machine, use the following instructions; otherwise, see NVS section. To install or update NVS, you should run the installation script. To do that, use the `cURL` or `Wget` command:

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.7/install.sh | bash
```

or

```
wget -qO- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.7/install.sh | bash
```

Downloading either of the above commands, downloads a script and runs it on your machine. If you are on macOS, you will need to install the XCode command line tools and Git before running the installation script; otherwise, it will fail. If you encounter the error “nvm: command not found” after running the installation script, you will need to restart the terminal instance (close and open a new terminal window). If this does not resolve the issue, refer to the troubleshooting [suggestions](#) provided in their GitHub repository. You can also try installing it [manually](#).

NVM Basic Commands ([source](#)):

Command	Description
nvm install --lts	Download and extract the LTS node version.
nvm install [version]	Download and extract a node version.
nvm uninstall [version]	Uninstalls the specified version.
nvm use [version]	Use a node version in the current shell.
nvm ls-remote	Displays a list of the available versions you can install from the remote listing.
nvm list	Displays a list of which versions you have installed.

Given the above basic commands, to install the latest LTS (20.10.0) of Node.js, you would use the following command:

```
nvm install --lts
```

Working with the CLI

Up until this point, JavaScript code had to be executed in a web browser, which is referred to as client-side scripting.

In client-side scripting, when a webpage is loaded in the browser, the JavaScript code is used to interact with the HTML elements (the webpages structure) and CSS properties (the webpages style) to make the website more interactive. Client-side scripting can be explained with the following steps:

1. A user requests a webpage, like <https://bellevue.edu>
2. The web server where Bellevue Universities website is hosted responds to the user's request by sending the HTML, CSS, and JavaScript for the page that was requested.
3. The user's browser first renders the HTML and CSS to display the requested webpage.
4. Then, the browser executes the JavaScript code, once a user invokes some type of action. For example, clicking on the login button.

It is important to note that in client-side scripting, the JavaScript code you write does not have access to the resources on the web server. Meaning, you cannot access the web server files, Operating System (OS), connected database, or other information stored on the computer.

CLI stands for Command Line Interface. It is a way of interacting with your computer by typing text commands into a terminal window (macOS/Linux) or command prompt (Windows). This is an alternative to traditional Graphical User Interface (GUI) windows. In a GUI, a mouse is used to “point and click” to invoke operations. For example, if I wanted to create a new folder on my computer, I would open a GUI and click on an option to “Create a new folder.” In a CLI environment, the same behavior is possible, but they are given as commands in a terminal window.

In the context of Node.js, the CLI is used to execute JavaScript files, install packages, start servers, and to perform tasks. It was discussed in the previous section (see Introduction) that “Node.js allows developers to write code outside of a web browser” and “server-side scripting is a technique utilized in web development that involves executing scripts on a web server to create a customized response to a user's request.” When you install Node.js it comes with a tool called node, which can be used to execute JavaScript files. Node.js also comes with a package manager (not to get confused with nvs and nvm) called the Node Package Manager (npm; discussed in the next section). With node we can create CLI programs using JavaScript. To create a Node.js CLI program, you will need some type of

text editor or Integrated Development Environment (IDE). Visual Studio Code (VS Code), is one of many text editors you can use to write Node.js CLI programs. In VS Code, to create a new JavaScript file, select File -> New File. Optionally, if you already have a folder created on your computer, you can open that folder to add new files to it. To do this, select File -> Open Folder.

The following code example was created in VS Code and demonstrates the basic structure of a CLI program that prints recipe data to the console.

```
// Array of recipes
const recipes = [
  { name: "Pasta", ingredients: "Noodles, Tomato Sauce" },
  { name: "Salad", ingredients: "Lettuce, Tomato, Cucumber" },
  { name: "Soup", ingredients: "Broth, Vegetables" }
];

// main function for the program;
// displays the recipe name and ingredients to the console
function main() {
  for (let i = 0; i < recipes.length; i++) {
    console.log("Recipe: " + recipes[i].name + "\nIngredients: " + recipes[i].ingredients);
    console.log(""); // new line
  }
}

// call the main function
main();
```

In this code example, an array of recipes is created with three different dishes. A main function is used to iterate over the recipes array and print the recipe name and ingredients to the console. To execute this CLI program, we use the Node.js built-in node tool.

Open a new terminal window or command prompt and navigate to the path where you saved the file on your computer. Optionally, you can open a new terminal window directly from VS Code. To do this, right-click the file and select "Open in Integrated Terminal." Next, enter the following command

node [file_name]

where file_name is the name of the file you want to execute. In the above example, I named the file recipes.js. So, for me, I would run:

node recipes.js

to execute the script, which will print the recipes the console window. Anytime you want to execute a Node.js script from the CLI, you always use the node command, followed by the file name. In the next code example, we will take a look at how to accept user input from process.argv. In Node.js, process.argv is an array that contains command-line arguments passed when the Node.js process is launched. The first element in the process.argv array is the path to the Node.js executable. The second element in the process.argv array is the path to the JavaScript file being executed. Take for example, the following:

```
// main function for the program; displays the node executable and file being executed  
function main() {  
  const nodeExecutable = process.argv[0];  
  const fileBeingExecuted = process.argv[1];  
  
  console.log("Node Executable: " + nodeExecutable);  
  console.log("File Being Executed: " + fileBeingExecuted);  
}  
  
main(); // call the main function
```

In this code example, a main function is created with two local variables. The first variable is assigned the value from the first element in the process.argv array, which is the path to your Node executable. And, the second variable is assigned the value from the second element in the process.argv array, which is the path file being executed.

Node Executable: /Users/rkrasso/.nvm/versions/node/v20.9.0/bin/node
File Being Executed: /Users/rkrasso/repos/buwebdev/web-340-solutions/week-1/cli-input-argv.js

As shown in the above printout, the first line is the path to the node version I am using (20.9.0) and the second line is the path to the file being executed (cli-input.js). Starting from process.argv[2], the elements are additional command-line arguments provided when a Node.js script is executed. For example, if I were to run the command node [file_name] [argument1] [argument2], argument1 would be located at element 3 in the process.argv array and argument2 would be located at element 4 in the process.argv array.

Consider the following program,

```
// File: recipe-cli.js  
  
function main() {  
  // check if the user entered a recipe name  
  if (process.argv.length !== 3) {  
    console.error("Usage: node recipe-cli.js <recipe>"); // display error message  
    process.exit(1); // exit with a non-zero error code  
  }
```

```

}

const recipe = process.argv[2]; // get the recipe name from CLI args

console.log("Recipe: " + recipe); // display the recipe name
}

main(); // call the main function

```

In the body of the main function an if statement is used to check the length of the process.argv array. This is done to ensure at least one argument is used when executing the Node.js script. For example, if you were to call this script using node recipe-cli.js, an error message would be printed to the console Usage: node recipe.js <recipe>. As previously mentioned, this is possible because the first and second elements in the process.argv array point to the Node executable file.

In the body of the if statement, you will notice two unique lines of code. The first line is printing an error message to the console window using console.error. You should always use console.error anytime you want to display an error message to the console. The second line calls process.exit with an actual parameter of the value 1. In Node.js, process.exit(1) is used to terminate the current process and signal the operating system that the process has ended with an error code. In subsequent chapters we will explore the process object in Node.js.

The correct usage is **node recipe-cli.js Pasta**, which would display **Recipe: Pasta** to the console window. We could extend this example, by accepting a second argument for the ingredients:

// File: recipe-cli-extended.js

```

function main() {
  // check if the user entered a recipe name and ingredients
  if (process.argv.length !== 4) {
    console.error("Usage: node recipe-cli-extended.js <recipe> <ingredients>"); //
display error message
    process.exit(1); // exit with a non-zero error code
  }

  const recipe = process.argv[2]; // get the recipe name from CLI args
  const ingredients = process.argv[3]; // get the ingredients from CLI args

  console.log("Recipe: " + recipe); // display the recipe name
  console.log("Ingredients: " + ingredients); // display the ingredients
}

```

main(); // call the main function

In this example, a new variable is introduced that captures the third element in the `process.argv` array, which is the second command line argument after the file name. The `if` statement has been updated to check for at least 4 elements in the array, which would look like this:

[“path/to/node”, “path/to/filename”, “Pasta”, “Noodles, Tomato Sauce”]

To call this script, you would use the following commands, making sure to enclose the ingredients in quotes if they contain spaces:

node recipe-cli-extended.js Pasta "Noodles, Tomato Sauce"

In addition, to executing Node.js scripts directly from the console, you can also execute JavaScript directly in the console. To do this, you use the `node` tool with either an `-e` or `-p` option, followed by the code to be executed.

- `node -e <code>`: The `-e` option stands for “evaluate.” It allows you to pass a string of JavaScript code that Node.js will execute. For example, `node -e “console.log(‘Using the CLI is great!’)”` will print “Using the CLI is great!”. The `-e` option does not print the results of an expression. Rather, you have to explicitly call `console.log` to print it.
- `node -p <code>`: The `-p` option stands for “print.” It works similar to the `-e` option, but it also prints the result of the expression to the console. For example, `node -p “15 * 20 + 10”` will print 310.

Use the `-p` when you want to quickly test a piece of JavaScript code and print the results of that expression. Let’s say for example, you are working on math homework and you need to find the square root of 81. You could use the `-p` option to do a quick calculation.

`node -p “Math.sqrt(81)”`

When you run this command, Node.js will calculate the square root of 81, which is 9, and print the results to the console. Use the `-e` option when you want to evaluate a piece of JavaScript code. Let’s say for example, suppose you want to check the formatting of a string in JavaScript. You can use the `-e` option to do this:

node -e "let favoriteFood = 'Steak Tacos'; console.log('My favorite food is: ' + favoriteFood);"

When you run this command, Node.js will evaluate the code and print “My favorite food is Steak Tacos”. Of the two options, the `-e` option is used more often, because it is a great

way to quickly write and test JavaScript code without having to test the entire project. For example, let's say you wanted to test a single Representational State Transfer Application Programming Interface (RESTful API). You could use the `-e` option to evaluate the code without having to test all of the RESTful APIs in the project (code that you did not touch). In the next section, we will explore the Node Package Manager (npm).

Node Package Manager (NPM)

Node Package Manager (npm), is a tool that helps you install and manage third-party packages and dependencies in a Node.js project. To understand the value of npm, consider the following analogy: Imagine you are working on a science fair project about the solar system. You need to create models for each planet in our solar system (earth, mars, the moon, the sun, etc.). Now, you could create all of these planets yourself from scratch, but that could take several hours and a lot of effort. What if there was a way for you to use premade models with minimal modifications. How much time would this save you?

This is where npm comes in. npm is a store or marketplace where other developers share programs (or “packages”) that perform specific tasks. These “packages” are generally available to the public and anyone can use them.

So, instead of creating all of the planets (or writing all the code) yourself, you can use npm to “order” packages of code (premade planet models) that others have already written (built). This gives you more time to spend on the core features of your science project (like arranging the planets in order to accurately represent the solar system), rather than spending time on creating things that have already been built by others.

You can find a list of available npm packages on the [npm registry](https://www.npmjs.com/), which is a marketplace of available packages that anyone can use. Each npm package has its own page with a description of the package (what it does) and instructions on how to install and use the package. Versions and licensing requirements are also available. In general, before you use a third-party package, you want to be sure it is actively maintained. The easiest way to ensure a package is active, is to check the last versions release date. If it has been longer than a year, then it is likely the package is either not being maintained or it is outdated. Always look for packages that follow a regular release cycle.

Not all packages are created equally. Some packages are maintained by large organizations with large development teams and others are maintained by individuals. Always do your research before selecting a package to use in your project. Not all packages are third-party, Node.js comes with its own set of “out-of-the-box” packages. Below is a list that highlights some of the most commonly used built-in packages:

1. `fs`: The `fs` module is a package that provides functions for working with the file system (covered in a later chapter).

2. `http`: The `http` module is a package that allows Node.js to transfer data over the HTTP protocol. With this package, you can create an HTTP server and build RESTful APIs (covered in a later chapter).
3. `path`: The `path` module is a utility package that provides access to file and directory paths on a machine (covered in a later chapter).
4. `os`: The `os` module is a package that provides functions for interacting with the operating system (covered in a later chapter).
5. `events`: The `events` module is a package that allows you to work with events. This package is core part of the asynchronous event-driven nature of Node.js (covered in a later chapter).
6. `util`: The `util` module is a package that provides utility functions, like `util.promisify`, which allows you to convert a callback function to a promise (covered in a later).
7. `readline`: The `readline` module is a package that provides an interface for reading data from a Readable stream (covered in a later chapter), like `process.stdin`. In the previous section we explored the `process.argv` array. You can use the `readline` package in a similar fashion to read user input from the console.

Built-in packages do not require installation, instead they can be imported directly in the Node.js program, by using a `require` statement. A `require` statement is how we import built-in and third-party npm packages. For example, let's convert the `recipe-cli` program to use Node.js's built-in `readline` package.

// File: recipe-input-cli.js

```
const readline = require("readline"); // import readline module
```

```
function main() {
```

```
  // create readline interface object
```

```
  const rl = readline.createInterface({
```

```
    input: process.stdin,
```

```
    output: process.stdout,
```

```
  });
```

```
  // prompt the user to input a recipe name and ingredients
```

```
  rl.question("Enter a recipe name: ", function (name) {
```

```
    rl.question("Enter the ingredients: ", function (ingredients) {
```

```
      console.log(` Recipe Name: ${name} `); // display the recipe name
```

```
      console.log(` Ingredients: ${ingredients} `); // display the ingredients
```

```
    rl.close(); // close the readline object
```

```
});  
});  
}
```

main(); // call the main function

In this code example, the readline package is imported using a require statement. Next, a main function is used for the program and in the body of the function a readline interface object is created using readline.createInterface. In the body of this object literal, input is set to process.stdin and output is set to process.stdout. Underneath the interface object, a call to rl.question is used to accept user input and an anonymous callback function is used to chain the second question and anonymous callback function. Finally, the inputted values are printed to the console and a call to rl.close() is used to close the readline object.

If you do not call rl.close(), the readline interface will remain open and the Node.js process will continue to run, waiting for more user input. This is because the readline interface is an instance of the EventEmitter (covered in a later chapter) and it keeps the Node.js event loop active as long as it is open. Most of the ideas covered in this program will be addressed in subsequent chapters, for now, the main focus is on how to use require statements to import built-in Node.js packages.

Let's expand on our recipe example and demonstrate how to convert measurements. We will create a Node.js script named recipe-converter.js that interacts with the CLI. This script will ask for an ingredient and its quantity in tablespoons, then convert that quantity to cups, a common need in cooking and baking recipes. Here is how we can do it:

// File: recipe-converter.js

const readline = require("readline"); // import readline module

```
function main() {  
  // create readline interface object  
  const rl = readline.createInterface({  
    input: process.stdin,  
    output: process.stdout,  
  });  
  
  // prompt the user to input an ingredient and its quantity in tablespoons  
  rl.question("Enter an ingredient: ", function (ingredient) {  
    rl.question("Enter the quantity in tablespoons: ", function (tablespoons) {  
      // check if the input is a number  
      if (isNaN(tablespoons)) {  
        console.error("Input must be a number."); // display error message  
        process.exit(1); // exit with a non-zero error code  
      }  
    });  
  });  
}
```

```

    }

    const cups = (tablespoons / 16).toFixed(2); // convert tablespoons to cups and
round to two decimal places

    console.log(
      ` For ${ingredient}, ${tablespoons} tablespoons is equivalent to ${cups} cups.`
    ); // display the conversion

    rl.close(); // close the readline object
  });
}

main(); // call the main function

```

To run this script, simply use the following command in your terminal:

node recipe-converter.js

The script will then prompt you to enter the ingredient and its quantity in tablespoons. If you enter a non-numeric measurement, it will display an appropriate error message. Not all npm packages are available natively (like readline), some require installation. To install a third-party package (located in the npm registry), you use an install statement. The command for an install statement is:

npm install <package_name>

where <package_name> is the package, you wish to install. Packages can be installed either locally (project level) or globally (computer level). For global packages, use the following command:

npm install -g <package_name>

where <package_name> is the name of the package, you wish to install. The -g option is used to specify that the package should be installed globally. To uninstall a package, you use the same syntax, but in reverse:

npm uninstall <package_name>
npm uninstall -g <package_name>

To view a list of globally install packages, use the following command:

npm list -g --depth=0

To view a list of locally installed packages, use the following command:

npm list

In order to install a npm package, you need a package.json file. In the next section we will explore what the package.json file is and how to install third-party npm packages.

package.json

The package.json file is a configuration file, in JSON format, that contains important information about your Node.js project. It maintains:

1. **Metadata:** Name of the project, version, description, author, and more.
2. **dependencies:** These are the third-party npm packages used in your project. The format is, project name and version number (covered below). To install dependencies, you use this command: `npm install`
3. **devDependencies:** These are similar to **dependencies**, but they are only used for development purposes. When you install a package as a “devDependency” it is added to this section. The distinction is, when you deploy your application to production, devDependencies are typically omitted from the installation. To save a npm package as a devDependency you use this command: `npm install --save-dev <package_name>`. To install only “production” dependencies you use the following command: `npm install --production`. Using this command, only the packages under the dependency section are installed. That is dependencies under devDependencies are ignored.
4. **scripts:** These are commands you can run to perform tasks, like executing a Node.js project.
5. **main:** Identifies the entry point to your Node.js program.
6. **license:** identifies the license type of your project.
7. **engines:** identifies the Node and npm versions required in the project.

There are two ways to generate a package.json file. The first is by manually creating a file and naming it package.json. The other option is to initialize a new project as a Node.js project. In either approach, you must first create a project folder for the program you are building. The manual approach is self-explanatory. Create a new folder and inside of that folder create a new file named package.json. Next, add the appropriate fields (see above) for your project. To initialize a new project, use the following command:

npm init -y

The `-y` option in the command stands for “yes”. It basically answers “yes” to all of the questions that would normally be asked if you omitted the `-y` option. The “questions” are basic configuration questions, like author, version, description, etc.,

Also, this command assumes you have already created a project folder and you have already navigated to the location on your computer where the new project folder was saved. In the below example, I created a new folder named `favorite-color` and ran the initialization command from that folder location, which generated the following `package.json` file.

```
{
  "name": "favorite-color",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

A fun package that demonstrates how to install third-party npm packages is `chalk`. The `chalk` package is a third-party npm package that allows you to change the color and style of the text that is printed to the console. To install this package in the `favorite-color` project use:

npm install chalk

Your `package.json` file should now resemble the following:

```
{
  "name": "favorite-color",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
```

```

"author": "",
"license": "ISC",
"dependencies": {
  "chalk": "^4.1.2"
}
}

```

Notice, because I omitted the `--save-dev` option, the dependency was added to the `dependencies` section. Had I added the option to the installation command, the package would have been added to the `devDependencies` section instead. The npm tool uses a system called Semantic Versioning (SemVer) for all packages. In Semantic Versioning, a version number is in the format of MAJOR.MINOR.PATCH, such as `^4.1.2` in the chalk dependency.

1. **MAJOR version:** Is updated anytime you make changes to a project that are incompatible with previous versions.
2. **MINOR version:** Is updated anytime you add functionality to the project that is backwards-compatible.
3. **PATCH version:** Is updated anytime you make bug fixes that are backwards-compatible.

The caret (^) before the version number tells npm to install any new minor or patch version of the MAJOR version. For example, 4.4.1 and 4.9.2 are accepted, but 5.1.9 is not because the MAJOR version has changed. This feature is there to automatically receive bug fixes and new features each time you run the command `npm update`. The tilde (~) before the version number of a package tells npm to only update the patch version of a MAJOR.MINOR version. For example,

- “chalk”: “~4.1.2”: This means npm can install any new patch version of chalk, but it cannot modify the MAJOR or MINOR version. 4.1.5 and 4.1.9 are accepted, but 4.2.8 is not, because the MINOR version has changed.

Once a package has been initialized and a `package.json` file created, you can begin adding JavaScript files to build your program. As previously mentioned, the `main` section in the `package.json` file points to the entry point of your project. That is, the first file to be executed when your program is started. Best practices encourage the naming convention of either `index.js` or `app.js` as the name of the file used as the entry point of your project. By default, npm will configure the entry point as `index.js` (see the `package.json` example for more details), but there are times when `app.js` should be used. For example, when you are building an HTTP server. Let’s add a new file to the favorite-color project named `index.js`.

To import the chalk package, you add a `require` statement to the `index.js` file

```
const chalk = require("chalk");
```

In this statement, a variable named chalk is assigned the chalk module, which effectively gives us access to the functions in the chalk package.

```
const chalk = require("chalk");  
const readline = require("readline");
```

```
// function to return the color message based on the color selected
```

```
function colorMessage(color) {  
  const output = "\nYour favorite color is "; // the output string
```

```
// switch statement to match the color to the color message
```

```
switch(color) {  
  case "1":  
    color = "blue";  
    return chalk.blue(output + color);  
  case "2":  
    color = "red";  
    return chalk.red(output + color);  
  case "3":  
    color = "green";  
    return chalk.green(output + color);  
  case "4":  
    color = "yellow";  
    return chalk.yellow(output + color);  
  case "5":  
    color = "orange";  
    return chalk.cyan(output + color);  
  default:  
    return "Invalid color";  
  }  
}
```

```
function menu() {  
  // display a menu with the available color options  
  // blue, red, green, yellow, and orange  
  console.log("\nAvailable colors:");  
  console.log("1. Blue");  
  console.log("2. Red");  
  console.log("3. Green");  
  console.log("4. Yellow");  
  console.log("5. Cyan");  
}
```

```

}

// main function for the program
function main() {
  // create a readline object
  const rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout
  });

  menu(); // display the menu

  // prompt the user to select a color
  rl.question("\nWhat is your favorite color? ", function(answer) {
    console.log(colorMessage(answer)); // display the color message
    rl.close();
  });
}

main(); // call the main function

```

Using any npm package (built-in or third-party), always starts with a require statement. It is how you access the module and its underlying functions. Without a require statement, the program will throw an error, which is covered in a later chapter.

In this program, three functions are created:

- **colorMessage:** This function uses a switch statement to determine, which color the user selected. Once the selected color is determined, the npm package chalk is used to call the appropriate function to change the color of the output variable to match the selection.
- **menu:** This function displays a menu to the user showing the selectable font colors.
- **main:** This is the main program function. It creates a new readline object, calls the menu function, prompts the user to select an option, and calls the colorMessage function to format and display the appropriate color message.

To execute this program, use this command:

node index.js

Optionally, you could add an execution script to the scripts section in the package.json file. To do this, add the following line to the scripts section of your package.json file.

```

{
  "name": "favorite-color",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "node index.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "chalk": "^4.1.2"
  }
}

```

In this version of the package.json file, a new script was added with the command “node index.js”. This is identical to how we would execute this script from the CLI. To run this script, use this command: npm start. What is great about this approach, is you no longer need to specify the file name to execute the program. Another advantage of using the scripts section is you can chain commands together to automate building, testing, and deploying a Node.js project. Another commonly used script is a “test” script. The “test” script is traditionally used to run unit tests against your application, which follows the format “test”: “node <test_file_name>” where test_file_name is the name of the test runner that executes all of your unit tests. For example, “test”: “node tester.js”. It is a best practice to use the scripts section, so in following chapters it will be used exclusively for executing all Node.js scripts.

Anytime you run the command npm install, a new folder is added to your project called node_modules. This folder contains all of the dependencies in your project (think mini programs) and it is needed in order to access the packages installed in your project; however, this folder should never be added to GitHub or left in a project when it’s deployed to production. As a Node.js project grows, the node_modules folder can become significantly larger. I have seen node_module folders reaching several Gigabytes in size. So do yourself a favor and everyone on your team, by not including the node_modules folder in your deployment and/or GitHub repository. In other words, keep it local. In the next section, we will explore the final topic of this chapter, package-lock.json.

package-lock.json

The package-lock.json file is automatically created when you run the command npm install. It lists all of your project’s dependencies and their exact versions at the time of installation. Here is why the package-lock.json file is important:

1. **Consistency:** it ensures that you always install the same dependencies each time you run npm install, regardless of the versions specified in your package.json file.
2. **Efficiency:** it speeds up the installation process, because npm can bypass recreating metadata.
3. **Security:** it includes checksums to ensure that you are getting the same code from when you originally installed a package. For example, let's say a malicious vendor decided to take over a third-party npm package and introduced a virus in the package to steal personal information. The checksums in the package-lock.json file will prevent your project from getting the new malicious codebase.

The package-lock.json file should always be added to GitHub (source control) and included in the deployment of your project. This will ensure that all developers on your team and the production version of your project have the exact same dependencies.

Programming Exercises

In this assignment, you will create a Node.js script that converts pounds to kilograms. The script will take one command line argument, which is the weight in pounds, and print the converted weight in kilograms to the console.

1. Create a new JavaScript file named weight-converter.js.
2. In this file, write a script that converts pounds to kilograms.
3. The script should take one command line argument, which is the weight in pounds.
4. The script should print the converted weight in kilograms to the console. The output should be rounded to two decimal places.
5. If the script is run without a command line argument, it should print the following message: stderr: 'Usage: node weight-converter.js <pounds>'
6. If the script is run with a non-numeric command line argument, it should print 'Input must be a number.' to stderr.

How to Run the Script

To run the script, use the following command in your terminal:

- `node weight-converter <pounds>`

Replace <pounds> with the weight you want to convert.

How to Test the Script

To test the script, use the `tester.js` file provided with the assignment. Run the tester with the following command:

- `node tester.js`

Hints:

- Use `process.argv` to access the command line arguments.
- Use `console.error` to print error messages to `stderr`
- Use `process.exit(1)` to exit the script with a non-zero status code when an error occurs.

Grading Criteria:

The assignment is worth a total of 60 points. Points are awarded as follows:

- Correct conversion: 20 points
- Correct error handling when no argument is provided: 20 points
- Correct error handling when a non-numeric argument is provided: 20 points.

Test Cases:

- Correct conversion: `node weight-converter.js 10` should output 4.54.
- Correct error handling with no argument is provided: `node weight-converter.js` should output `Usage: node weight-converter.js <pounds>`
- Correct error handling when a non-numeric number is provided: `node weight-converter.js not-a-number` should output `Input must be a number.`

Chapter 2. Module System and Diagnostics

Chapter Overview

In this chapter, we will be taking an in-depth look at the module system in Node.js. This aspect is crucial to understand as it forms the foundation for building complex Node.js applications. We will start by exploring the concept of modules and why they are necessary for building large-scale applications. We will then delve into the default module system in Node.js, CommonJS, and learn how to both create and use modules in our applications. Additionally, we will touch on a new standard that has emerged, ES Modules, and compare it to CommonJS. Bugs and errors are an inevitable part of the development process, so we will also cover the necessary skills you need for debugging and troubleshooting in Node.js. The goal of this chapter is to equip you with a comprehensive understanding of Node.js module system and essential debugging skills, which will serve as the foundation for contemporary Node.js development.

Learning Objectives

By the end of this chapter, you should be able to:

- List the differences between CommonJS and ES Modules.

- Explain how to create custom modules in Node.js.
- Identify strategies for debugging and troubleshooting errors in a Node.js program.
- Build a Node.js program using custom modules.

Introduction to Node.js Module System

The Node.js module system is a mechanism designed for code reuse. It is a process of creating reusable code segments that can be imported (require statement) into different JavaScript files. Each module can contain functions, objects, or values that can be exported and imported in another JavaScript file. To understand the value of this feature, consider two cooking techniques: sauteing and roasting.

Sauteing is a quick cooking process where ingredients are rapidly cooked in a small amount of oil or fat over high heat. This technique delivers a dish that's nicely seared on the outside while fully cooked on the inside. On the other hand, roasting is a slower cooking method that takes place in an oven. The food, often meat or vegetables, is left uncovered to allow the heat to caramelize the exterior, creating a flavorful crust.

Now, imagine you're compiling a cookbook and you want a recipe to employ the sauteing technique or the roasting technique. Instead of detailing these techniques each time a new recipe is introduced, wouldn't it be more efficient if you could refer to the new recipe's technique as "this recipe utilizes the sauteing technique" or "this recipe employs the roasting technique?" This way, you can incorporate these techniques into your recipe without having to reiterate them or explain how they work.

Modules work in a similar way; a module is like a cooking technique. It has certain "methods" that can accomplish various tasks. Much like a module has functions that carry out various tasks. When you want to use a "method" like sauteing, you don't have to explain how it works, you simply refer to the sauteing technique. Just like in a module that provides the square root of a number, you don't need to explain how to calculate the square root of a value, instead you simply call the module and it provides a function that calculates the square root of a value.

The ability to create code that is reusable is foundational to software development. As a developer, your goal is to write code that is DRY (Do not repeat yourself). Meaning, if you write a block of code that is used multiple times in your project, it is better suited in a module. Imagine, the maintenance nightmare that would be associated with having to manage the same block of code in 50 files and 20 projects. Is it reasonable to suggest, that a single Node.js module is a better approach? In the next section, we will explore CommonJS, which is the default module system in Node.js.

CommonJS

By now, you are probably aware of the fact that JavaScript has advanced to a stage where ES Modules are common place (covered in the next section). But, how was code made

reusable before ES Modules was a thing? Well, the answer in Node.js is CommonJS. CommonJS is a module system that is used in Node.js to manage dependencies and export modules in a Node.js program. It allows developers to reuse code blocks from separate files. As you learned in the previous chapter, modules (npm packages) can be imported into a file through a require statement. For the require statement to work, it meant that an object, function, or data object had to be exported in the imported package. Now, let's bring this concept to life with a practical example.

Imagine you are designing a gardening application that allows users to keep track of their favorite plants. You could create a module that provides a default list of plants and includes functionality that allows users to add more plants as their list of favorite plants grows. Consider the following module in a file named plant-list.js:

```
"use strict";

// Array of plants
const plants = [
  { name: "Rose", type: "Flower" },
  { name: "Oak", type: "Tree" },
  { name: "Cactus", type: "Succulent" },
];

// function that returns the plants array
function getPlants() {
    return plants;
}

// function that adds a plant to the plants array
function addPlant(name, type) {
    plants.push({ name: name, type: type });
}

// export the functions using the module.exports object which is part of CommonJS
module.exports = {
    getPlants: getPlants,
    addPlant: addPlant,
};
```

In this code example, an array of plants is created with three objects: Rose, Oak, and Cactus. Two functions are created to return the array of plants and add new plants to the array. Finally, module.exports is used to export the two functions, which makes them accessible in a separate JavaScript file. The power of this module can only be understood when you consider its usage in a “real-world” website. Imagine, you are building a website that keeps track of a user’s favorite plants and also has a list of plants they want to grow

each season. Without a module, you would have to include this code in two separate files: favorite plants page and seasonal plants page. But, with a module, you can use a single codebase to satisfy both webpage requirements. To use the functions in this module, you include require statements in the file that accesses the functions. For example, in a file named index.js you would use:

```
// import the getPlants and addPlant functions from the plant-list.js file  
const { getPlants, addPlant } = require("./plant-list");
```

The use of curly braces ({ }) in the require statement is using CommonJS module system to import functions from the plant-list JavaScript file.

- **const { getPlants, addPlant }:** this is using destructuring assignment to pull out the getPlants and addPlant functions from the object that the require statement returns. Destructuring is a handy technique for accessing properties from objects or arrays that are exported. Essentially, this line is requesting, “Extract the getPlants and addPlant functions from the object that the require statement is bringing in.”

Let’s take a look at the remaining code in the index.js file to illustrate how we can use the plant-list module in a Node.js program:

```
"use strict";
```

```
// import the getPlants and addPlant functions from the plant-list.js file  
const { getPlants, addPlant } = require("./plant-list");
```

```
// function that displays the plants array to the console  
function displayPlants() {  
  const plants = getPlants(); // call the getPlants function
```

```
  // loop over the plants array and output the results  
  for (let i = 0; i < plants.length; i++) {  
    console.log("Plant: " + plants[i].name + "\nType: " + plants[i].type);  
    console.log(""); // new line  
  }  
}
```

```
// main function for the program;  
function main() {  
  console.log("--Plant List--");
```

```
  displayPlants(); // call the displayPlants function
```

```
// display a message to the console indicating we are adding a new plant
```

```

console.log("Adding a new plant...");

// add a new plant to the plants array
addPlant("Tulip", "Flower");

// display a message to the console displaying the new plants list
console.log("\nPlant List");

displayPlants(); // call the displayPlants function
}

main();

```

The rest of the code outlines the `displayPlants` function, which presents the array of plants from the `plant-list` module. Additionally, a `main` function is used as the primary function of the program. This `main` function invokes the `displayPlants` function, appends a new plant to the list, and then calls the `displayPlants` function again. This sequence effectively showcases the use of both functions from the module. In the following section, we will delve into an alternative method for exporting and importing module, a module that has become the norm for client-side JavaScript applications.

ES Modules

ES Modules (ECMAScript) are the official standard format to package JavaScript code for reuse. Similar to CommonJS, its goal is to provide developers with a mechanism for sharing code segments across multiple JavaScript files. ES Modules were introduced in ES6 (ES2015) for use in the browser, and recently, support has been added to Node.js.

ES Modules use an `import` statement to load JavaScript files and an `export` statement to expose objects, functions, and values to other JavaScript files. This is in contrast to CommonJS's approach to importing and exporting code segments, which uses a `require` statement and `module.exports`. Starting with Node.js 13, official support for ES Modules was added and shipped with each new LTS release. Prior to this point, if you wanted to use ES Modules you had to use the `--experimental-module` option.

You can use ES Modules in a Node.js program by either using the `.mjs` file extension for your JavaScript files, or by setting `"type": "module"` in your `package.json` file. Both approaches have their pros and cons, and the best choice depends greatly on the project you are working on and the project's individual business requirements. In general:

- **.mjs extension:** This approach allows you to use both CommonJS and ES Modules in a Node.js project. Files with `.js` are treated as CommonJS and files with `.mjs` are treated as ES Modules. This is useful and flexible, especially if you are working with

third-party packages that only support ES Modules or you are in the process of transitioning a project to ES Modules.

- **“type”: “module”**: This approach treats all .js files as ES Modules. This is a similar approach and less confusing, but it is equally less flexible.

For the sake of flexibility, all examples in subsequent chapters will use the .mjs extension over configurations in the package.json file. Let's continue with the gardening application from the previous section, but this time utilizing ES Modules. The project will be structured in a folder named gardening-app-esmodules. Inside this folder, we will have two main files: plant-list.mjs, which will contain the module that exports the getPlants and addPlant functions, and index.mjs, which will import these functions and contain the main logic of our application.

```
// plant-list.mjs
```

```
"use strict";
```

```
// Array of favorite plants
```

```
const plants = [  
  { name: "Rose", type: "Flower" },  
  { name: "Oak", type: "Tree" },  
  { name: "Basil", type: "Herb" },  
];
```

```
// function that returns the plants array
```

```
function getPlants() {  
  return plants;  
}
```

```
// function that adds a plant to the plants array
```

```
function addPlant(name, type) {  
  plants.push({ name: name, type: type });  
}
```

```
// export the functions using the export keyword which is part of ES Modules
```

```
export { getPlants, addPlant };
```

This code mirrors the one we wrote in the previous section, but it uses ES Modules. To import the getPlants and addPlant functions, you use an import statement:

```
// import the getPlants and addPlant functions from the plant-list.mjs file
```

```
import { getPlants, addPlant } from "./plant-list.mjs";
```

Let's take a look at the remaining code in the index.mjs file to illustrate how we can use the plant-list ES Module in a Node.js program:

```
"use strict";

// import the getPlants and addPlant functions from the plant-list.mjs file
import { getPlants, addPlant } from "./plant-list.mjs";

// function that displays the plants array to the console
function displayPlants() {
  const plants = getPlants(); // call the getPlants function

  console.log("-- DISPLAYING PLANTS --"); // display a message to the console

  // loop over the plants array and output the results
  for (let i = 0; i < plants.length; i++) {
    console.log("Plant: " + plants[i].name + "\nType: " + plants[i].type);
    console.log(""); // new line
  }
}

function main() {
  displayPlants(); // call the displayPlants function

  // add a new plant
  addPlant("Lavender", "Herb");

  displayPlants(); // call the displayPlants function
}

main(); // call the main() function
```

Other than how the functions are exported and imported; the remaining code is identical to what we used in the CommonJS module example.

It's worth highlighting that, as of Node.js 20, not all third-party npm packages have adopted ES Modules and some core Node.js packages still lack full compatibility. For example, `__dirname` and `__filename`, which are global variables and part of the CommonJS system are not available in ES Modules.

- **`__dirname`:** A global variable that gives you the directory name of the current module. It is the absolute path to the directory where a file is being executed from.

- **__filename:** A global variable that gives you the absolute path to the file being executed.

In ES Modules, the latter can be achieved, but with more code:

```
import { fileURLToPath } from 'url';
import { dirname } from 'path';

const __filename = fileURLToPath(import.meta.url);
const __dirname = dirname(__filename);
```

This is just one example, but there are many more examples throughout the Node.js ecosystem. As of now, Node.js 20 defaults to the CommonJS module system and while there have been discussions about making ES Modules the default in the future, no definitive plans have been announcements. Changing the default module system in Node.js would be a significant change that could potentially break a lot of existing Node.js packages and tools. It is my opinion that it is very unlikely Node.js will ever default to ES Modules. Because of this, I recommend sticking with the CommonJS module system. However, you should be familiar with ES Modules and how to use them in a Node.js program. In the next section, we will explore Node.js debugging and diagnostics.

Debugging and Troubleshooting

In the context of programming, debugging is the process of identifying and fixing errors in a program. Learning how to properly troubleshoot and debug a program is extremely important in software development.

1. **Identify and Fix Errors:** Debugging is an essential tool to detect and correct errors in your code, including syntax errors that impede your code from running and logical errors that lead to incorrect output. By utilizing debugging, you can troubleshoot and solve these problems, ensuring that your code runs smoothly and produces accurate results.
2. **Understand Code Flow:** Debugging is a powerful tool that enables you to grasp how your code behaves, which is particularly advantageous when working with asynchronous code, a typical pattern in Node.js. By utilizing debugging, you can better understand the flow and execution of your code, allowing you to efficiently track down and resolve issues that arise. This can be immensely beneficial, particularly when dealing with complex code that requires a thorough comprehension of its operations.
3. **Optimize Performance:** Debugging tools can often be used to help you identify bottlenecks and performance issues, like slow functions and memory leaks.

4. **Learning and Improvement:** Debugging helps you learn from the mistakes you have made and it teaches you how to write better code. This improves your coding skills and understanding of the programming language you are using.

Most programming languages and frameworks provide strategies and tools to simplify the process. In Node.js you can use debugging tools like the built-in debugger, the Chrome DevTools, or an IDE/Text Editor (VS Code).

Node.js supports remote debugging through Google Chrome DevTools. To enter remote debugging mode, use the `--inspect` option when executing a Node.js script.

`node --inspect <file_name>`

where `file_name` is the name of the file you are executing. Another option for entering remote debugging mode is using the `--inspect-brk` option when executing a Node.js script.

`node --inspect-brk <file_name>`

where `file_name` is the name of the file you are executing. The difference between these two approaches is:

1. **`--inspect`:** Starts the Node.js application in debug mode and opens a WebSocket to display for debugging commands, but it does not pause execution. Meaning the application runs regardless if you have a breakpoint set.
2. **`--inspect-brk`:** Starts the Node.js application in debug mode and opens a WebSocket to display the debugging commands, but this time execution is paused at the first breakpoint. This ensures you will not miss any breakpoints and it gives you a chance to “inspect” the code you have written.

In either case, you can connect to the WebSocket either through Chrome DevTools or VS Code to interactively debug your Node.js application. To debug a Node.js program with Chrome DevTools, follow these steps:

1. Start your Node.js program with the `--inspect-brk` option.
2. Open Google Chrome and navigate to `chrome://inspect`.
3. In the DevTools window, click on Open dedicated DevTools for Node.
4. In the dialog window, you will see the executed JavaScript file, along with options for setting breakpoints.

5. To set a breakpoint, click on the line number of the line of code you want to inspect. If done correctly, the line number should be highlighted in blue.
6. From this dialog window, you can resume script execution, step over to the next function call, step into the next function call, and step out of the current function call.

To debug a Node.js program with VS Code, follow these steps:

1. Open your project in VS Code.
2. Click on the Run and Debug icon from the Activity Bar on the side of the editor. It looks like a play button with a bug underneath it.
3. Click on create a launch.json file link. This will launch a dropdown menu asking you to select an environment. Chose Node.js.
4. Doing so, will create a new folder and file in your project. The folder will be named .vscode and the file will be named launch.json. VS Code will use the configurations placed in this file to launch and debug your Node.js project. The folder and file are always placed at the root of your project and the configurations apply to every file in the project. Meaning, you do not need to create multiple configuration files for your project.
5. In the launch.json file, you will see a configuration array. Add a new property to this array: "runtimeArgs": ["--inspect-brk"].

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "node",
      "request": "launch",
      "name": "Launch Program",
      "skipFiles": [
        "<node_internals>/**"
      ],
      "program": "${workspaceFolder}/week-2/gardening-app-esmodules/index.mjs",
      "runtimeArgs": ["--inspect-brk"]
    }
  ]
}
```

6. Set a breakpoint in your code by clicking on the left of a line where you want to inspect the code. If done correctly, a red dot should appear to the left of the line number.
7. Once you have set your breakpoints, click on the green button titled Launch Program at the top of the Run and Debug sidebar tab.
8. If done correctly, the code will run and pause at the line where you set the breakpoint. You can inspect the values of variables, step through your code, set watches, view the call stack, and view exceptions. You can also set more breakpoints while the code is in debugger mode.
9. To inspect a variable, you can either view the variables under the Variables section in the left activity bar or you can hover directly over the variable to inspect its content. In either approach, the data content is interactive.

Stepping through is a term used in debugging that allows you to control the execution of your program, line by line. The common stepping operations are:

1. **Continue:** This command continues executing code without pausing until the next breakpoint is reached. This is a great way to jump to different sections in your program.
2. **Pause:** This command pauses the execution of the code.
3. **Step over:** This command executes the current line of code. If the line contains a function call, it executes the entire function and then pauses at the next line of the current function. This is useful when you want to skip over the details of a function.
4. **Step into:** This command also executes the current line of code. But, if the line contains a function call, it “steps into” the function and pauses at the first line of that function. This is useful when you want to see the details of a function.
5. **Step out:** This command continues executing code until the current function is finished, then pauses at the next line of the parent function. This is useful when you used the step into command and are finished with the function you are inspecting.
6. **Restart:** This command restarts the debugger session.
7. **Stop:** This command stops the debugger session.

The importance of learning how to properly debug and troubleshoot a program cannot be understated. It is your responsibility as a software developer to know how your program is behaving and to know where to look for answers if something is not working as intended.

Setting breakpoints and using the built-in tools within VS Code is just one of many ways you can fix common mistakes and teach yourself how to become a better software developer.

`console.log` and `console.error` are simple but powerful tools for debugging and troubleshooting Node.js programs. They allow you to output information to the console that can help you understand how your code is behaving. Here are the reasons why you might use them:

1. **Printing Variable Values:** You can use `console.log` to print the value of a variable and to verify whether the application is reaching certain parts of your code. For example, let's say you built a function, but the value being returned from a calculation within that function appears to be wrong. You can use `console.log` to check if the calculation is being performed correctly.
2. **Tracking Execution Order:** You can use `console.log` statements to track the execution of your code. This can help you understand the order in which your code is being executed and whether there is unexpected behavior that should be addressed. For example, let's say you are building a RESTful API and the response is a 404. You can use `console.log` statements to determine if the API call is reaching your code.
3. **Error Reporting:** You can use `console.error` to report errors. `console.error` works just like `console.log`, but it outputs to `stderr` instead of `stdout`. `console.error` and `stderr` should always be used for error messages. Most web servers include logging mechanism to respond and differentiate between `console.log` and `console.error`.
4. **Debugging Asynchronous Code:** You can use `console.log` to understand the order in which asynchronous code is being executed. This is especially useful when you have a requirement to control the execution order of asynchronous code segments.

`console.log` and `console.error` are great tools for debugging and troubleshooting simple programs, but they are not a replacement for dedicated debugging tools, like Chrome DevTools and VS Code's debugger. You should always prioritize dedicated testing tools over `console.log` and `console.error`.

Programming Exercises

In this assignment, you will be creating a simple recipe application using the Node.js CommonJS module system. You will need to implement the following tasks:

1. Create a `createRecipe` function (10 points): This function should take an array of ingredients and return a string in the format "Recipe created with ingredients: ingredient1, ingredient2".

2. Create a `setTimer` function (10 points): This function should take a number of minutes and return a string in the format: "Timer set for X minutes."
3. Create a `quit` function (10 points): This function should return the string "Program exited".
4. Use the `index.js` file to demonstrate the functionality of your modules (10 points).
5. Create a `package.json` file (10 points): This file should include a "start" script and a "test" script. The "start" script should execute the `index.js` file (see step 4) and the "test" script should execute the `tester.js` file. If you need help setting these up, refer to your reading materials.
6. Set up the "start" and "test" scripts correctly (10 points): The "start" script should run your `index.js` file, and the "test" script should run the `tester.js` script.
7. **Note:** Each of these functions should be placed in the `recipes.js` file and exported using Node.js's CommonJS module system.

Once you have completed your assignment, you can test your code by running the `tester.js` script. You can do this by using one of the following methods:

1. Run `node testers.js` in the terminal. This will run the script directly.
2. Run `npm test` in the terminal. This will run the "test" script from the `package.json` file, which should execute the `tester.js` script.

If all tests pass, you will see the message "All tests passed!". If any of the tests fail, you will see an error message with an explanation of why the test failed.

Chapter 3. Error Handling and Unit Testing

Chapter Overview

Programs can encounter an error for various reasons, such as programming errors, incorrect inputs, or network issues. To prevent programs from crashing or behaving unpredictably due to these errors, it is necessary to manage them properly. This process of managing errors in a program is known as error handling, which aims to handle errors in a controlled manner. Error handling involves providing users with an explanation of the error, writing error messages to a console window or sever log file, attempting to re-execute the problematic code, or terminating the program silently. One of the primary techniques for addressing application-level errors is using `try/catch` blocks.

Unit testing, which is a popular testing method in software development, is used to help developers validate the code they are writing. Tests are written in “small chunks” and ran against a segmented code block. Unit testing is very useful, because it can help prevent defects and identify issues early in the development process. Its main objective is to isolate a part of code in an application to check its validity.

In this chapter we will explore how to handle errors in a Node.js application and how to write basic unit tests using the Node.js assert library.

Learning Objectives

By the end of this chapter, you should be able to:

- List the steps in the TDD cycle.
- Explain the differences between the Node.js assert library and a testing framework.
- Build custom error objects.
- Write unit tests using the Node.js assert library.

Introduction to Test-Driven Development (TDD)

One of the software development methodologies that is widely used is Test-Driven Development (TDD). It involves an iterative process of building programs. The developer creates a unit test that is meant to fail, then updates the code to pass the test, and finally refactors and implements the code in the program. This cycle is commonly called “Red, Green, Refactor,” and it can be repeated multiple times during the development of an application or until the desired outcome is achieved.

To understand this process, let’s explore a real-life example. Imagine you are baking a cake for the first time for your friend’s birthday.

1. **First (Red):** First, you start by writing down the recipe (this is your test) for the cake. But, because you have never baked a cake before, you do not know if the recipe is going to be good and if your friend is going to like it. This is like writing a unit test that is going to fail.
2. **Second (Green):** Second, you follow the recipe and bake the cake (this is your code). You taste the cake and it’s exactly how you wanted it to come out. This is like writing the minimal amount of code possible to pass the test.
3. **Third (Refactor):** Third, you look at the recipe and think about how to make it better. Perhaps you can reduce the amount of sugar in the recipe or add more vanilla to make it sweeter. This is like refactoring your code, where the goal is to improve the quality of the code you have written.

In other words, TDD is like writing a recipe (test), baking the cake to match the recipe (passing the test), and then improving on the recipe (code refactoring). By following this iterative approach, you can ensure the cake is always consistently baked the same way and it is always delicious. If you want to modify the recipe, you update the recipe card, bake a new cake, taste it, and continue to repeat the process until you are happy with the results.

The benefits of TDD are:

- **Improved Quality:** By writing unit tests first, you are ensuring that every line of code you write in the program has been fully tested. This can lead to fewer bugs and a higher quality code base.
- **Better Design:** TDD forces you to think about the code you are writing before it is written. This “pre-thought-process” leads to code that is more maintainable, easier to scale, and more flexible.
- **Simpler Debugging:** Because you are writing “chunk’ size unit tests, if something fails it is easier to troubleshoot and debug. This makes debugging faster and more efficient.
- **Documentation:** Tests serve as a form documentation, illustrating to other developers how the code you are writing is supposed to behave.
- **Confidence:** You will be more confident in the changes you make to an application, because if something breaks you will know that the unit tests will catch the errors. This will ensure the application is free of errors when it is deployed to production and paying customers.

Let’s imagine we are developing a function to calculate the square root of a number using TDD in JavaScript.

1. **Red (failing test):**

```
function test_calculateSquareRoot() {  
  let result = calculateSquareRoot(9);  
  console.log(result)  
}
```

In this code example, the test will fail, because the code for the `calculateSquareRoot` function is missing.

2. **Green (minimum code to pass test):**

```
function calculateSquareRoot(num) {  
  // return square root of number  
}
```

```
test_calculateSquareRoot(); // run the test again; should pass
```

3. **Refactor:** In this case, the function is already simple enough, so it does not need to be refactored, but if it did, this would be the time for you to improve the code without modifying the behavior.
4. **Repeat:** Repeat the process for the next piece of functionality in your program. For example, handling use cases when the actual parameter is not a numeric value. You could start by writing a unit test for that use case and repeat the Red-Green-Refactor cycle.

In the next section, we will explore how to write unit tests using TDD with the Node.js assert library.

Unit Testing with Node.js Assert Library

The assert module, is a built-in Node.js module that provides a set of functions for writing simple unit tests that test the functionality of your application. In the context of software development, “assert” is a method used in writing test cases. It checks for conditions in your program to meet certain criteria’s that you have defined. If the criteria is met, the program will continue to execute as expected; however, if the criteria is not met, the program will throw an error and the execution is stopped.

Most programming languages and frameworks have an assert library (JavaScript, Python, Java, C#, Ruby, PHP, etc.,). However, it is important to note, that generally speaking, an assert library is very basic. For more complex use cases, it is always better to use a more powerful testing library (covered later in this chapter).

Here is a brief introduction to the assert library:

1. **Importing:** You can import the assert module using a require statement.

```
const assert = require("assert");
```

2. **Basic usage:** The most basic use of the assert library is the assert function, which test whether a value is truthy. If it is not, assert will throw an error.

```
const assert = require("assert");
```

```
assert(value, "This is an error message");
```

3. **Assert equal:** You can test the truthy of values using the `assert.equal` function.

```
const assert = require("assert");

assert.equal(actual, expected, "Error Message");
```

The `assert.equal` function is using the equality operator (`===`), to test if the two values match. The challenge with this approach is, the behavior may not be accurate with different data types. Take for example, the following:

```
const assert = require("assert");

assert.equal(1, "1", "Error Message");
```

This test will pass, because in JavaScript, loose equality (`==`) performs type coercion if the types of two variables being compared are different. This basically means, the string `"1"` will get converted to a numeric value before it is evaluated. For these situations, when you want to enforce a strict equality check, you should use `assert.strictEqual`.

4. **Assert strict equal:** You can test the truthy of two values are strictly equal.

```
const assert = require("assert");
assert.strictEqual(1, "1", "Error Message");
```

The `assert.strictEqual` function will do a strictly equal check, resulting in this unit test failing.

5. **Assert deep equal:** If you want to check if two objects, arrays, or other complex data types are equal, you use `assert.deepEqual`.

```
const assert = require("assert");

assert.deepEqual(actual, expected, "Error Message");
```

This function checks the structure and properties of the comparing objects for strict equality.

6. **Assert throws:** You can check if a function throws an error by using the `assert.throws` function.

```
const assert = require("assert");
```



```

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

```

For a more in-depth exploration of the assert module, refer to the [official documentation](#) for the Node.js module.

In the context of TDD, we can use the assert module to build unit tests following the Red-Green-Refactor cycle. Let's rewrite the gardening project from the previous chapter to conform to TDD. The project structure has been modified to conform to industry best practices:

```

gardening-app
  src
    index.js
    plant-list.js
  test
    plant-list.spec.js
package.json

```

```

// package.json
{
  "name": "gardening-app",
  "version": "1.0.0",
  "description": "",
  "main": "src/index.js",
  "scripts": {
    "start": "node src/index.js",
    "test": "node test/plant-list.spec.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}

```

Step 1 (Red): The first step is to write unit tests that will fail.

```

"use strict";

const assert = require("assert");

```

```

const plantModule = require("../src/plant-list");

function testGetPlants() {
  const plants = plantModule.getPlants();
  assert.strictEqual(plants.length, 3, "Initial plants array should have 3 elements");
  assert.strictEqual(plants[0].name, "Rose", "First plant should be Rose");
}

function testAddPlant() {
  plantModule.addPlant("Lavender", "Herb");
  const plants = plantModule.getPlants();
  assert.strictEqual(plants.length, 4, "Plants array should have 4 elements after adding Lavender");
  assert.strictEqual(plants[3].name, "Lavender", "Last plant should be Lavender");
}

testGetPlants();
testAddPlant();

console.log("All tests passed!");

```

Using require statements, the assert library and plant module are imported. Next, two test functions are created that test the functionality of our proposed functions. And, finally, the test functions are called and a message is printed to the console indicating the unit tests have passed.

The next step, as part of the red cycle, is to add a new file named plant-list.js under the src folder.

```

// src/plant-list.js
"use strict";

const plants = [
  { name: "Rose", type: "Flower" },
  { name: "Oak", type: "Tree" },
  { name: "Basil", type: "Herb" }
];

function getPlants() {
  // Implementation goes here
}

function addPlant(name, type) {
  // Implementation goes here
}

```

```
}
```

```
module.exports = {  
  getPlants: getPlants,  
  addPlant: addPlant  
};
```

This code should look familiar; it is identical to what we used in the previous chapter's ES Modules and CommonJS Modules examples, with the implementation details of each function missing. Remember, our goal in the red stage of the cycle is to create failing unit tests. Once this code is added, we need to run the program to confirm that the tests are failing. Enter the following command in a terminal window:

npm test

Step 2 (Green): Once you run the tests and see them fail, you add the code to implement the functions to make the tests pass.

```
"use strict";
```

```
// Array of plants
```

```
const plants = [  
  { name: "Rose", type: "Flower" },  
  { name: "Oak", type: "Tree" },  
  { name: "Cactus", type: "Succulent" },  
];
```

```
// function that returns the plants array
```

```
function getPlants() {  
  return plants;  
}
```

```
// function that adds a plant to the plants array
```

```
function addPlant(name, type) {  
  plants.push({ name: name, type: type });  
}
```

```
// export the functions using the module.exports object which is part of CommonJS
```

```
module.exports = {  
  getPlants: getPlants,  
  addPlant: addPlant,  
};
```

Rerun the test command (**npm test**) to ensure the tests are now passing. Only after the tests have passed, do we begin writing the code in the index.js file to interact with the plant-list module. This is the essence of how TDD works. Red, Green, Refactor.

```
// src/index.js
"use strict";

// import the getPlants and addPlant functions from the plant-list.js file
const { getPlants, addPlant } = require("./plant-list");

// function that displays the plants array to the console
function displayPlants() {
  const plants = getPlants(); // call the getPlants function

  // loop over the plants array and output the results
  for (let i = 0; i < plants.length; i++) {
    console.log("Plant: " + plants[i].name + "\nType: " + plants[i].type);
    console.log(""); // new line
  }
}

// main function for the program;
function main() {
  console.log("--Plant List--");

  displayPlants(); // call the displayPlants function

  // display a message to the console indicating we are adding a new plant
  console.log("Adding a new plant...");

  // add a new plant to the plants array
  addPlant("Lavender", "Herb");
  // display a message to the console displaying the new plants list
  console.log("\nPlant List");

  displayPlants(); // call the displayPlants function
}

main();
```

To execute this program, run the start command (**npm start**) from a terminal window. You should see a similar output to what was provided in chapter 2. Likewise, running the test command (**npm test**) should execute the unit tests and you should see an output of “All

tests passed!” In this example, there is an important item to pay close attention to and that is the folder structure of the project.

```
foldername
  src
    index.js
    module_file_name.js
  test
    test_file_name.spec.js
package.json
```

This is a standard best practice in Node.js and something you should always follow when building Node.js projects. For the remaining chapters in this book, this organizational structure is followed and assumed. For the test file, you can use either .spec.js or .test.js. I am using .spec.js because many popular frameworks like Angular use this naming convention and I prefer it over .test.js because it is easier to identify the test files. Technically, either approach is correct and perfectly acceptable. For the sake of consistency, this book will use .spec.js for all test files. Because learning TDD can be complicated in the beginning, let’s look at another example of TDD in action.

Exercise Overview:

Design a simple JavaScript function that takes a user’s numerical grade and converts it to a letter grade. Use the following grading criteria to determine the awardable grade:

- 100 – 90 (A): Grades between 90 and 100% are awarded a letter grade of an “A”.
- 80 – 89 (B): Grades between 80 and 89% are awarded a letter grade of a “B”.
- 70 – 79 (C): Grade between 70 and 70% are awarded a letter grade of a “C”.
- 60 – 69 (D): Grades between 60 and 69% are awarded a letter grade of a “D”.
- 59 – 0 (F): Grades between 0 and 59% are awarded a letter grade of a “F”.

Make sure your unit tests cover each gradable criteria, only numerical values are accepted as actual parameters, and only numbers in the range of 100 and 0 are accepted.

Folder structure:

```
gradebook
  src
    index.js
    grader.js
  test
```

```
grader.spec.js
package.json
```

Step 1 (Red): Write the unit tests to fail.

```
// test/grader.spec.ts
```

```
const assert = require("assert");
const graderModule = require("../grader");
```

```
function testGetLetterGrade() {
  assert.strictEqual(graderModule.getLetterGrade(100), "A");
  assert.strictEqual(graderModule.getLetterGrade(90), "A");
  assert.strictEqual(graderModule.getLetterGrade(80), "B");
  assert.strictEqual(graderModule.getLetterGrade(70), "C");
  assert.strictEqual(graderModule.getLetterGrade(60), "D");
  assert.strictEqual(graderModule.getLetterGrade(59), "F");
  assert.strictEqual(graderModule.getLetterGrade(0), "F");
}
```

```
function testGetGradeOnlyAcceptsNumericalValues() {
  assert.throws(() => graderModule.getLetterGrade("100", "Input must be a number."));
  assert.throws(() => graderModule.getLetterGrade(true, "Input must be a number."));
  assert.throws(() => graderModule.getLetterGrade({}, "Input must be a number."));
}
```

```
function testGetGradeOnlyAcceptsValuesBetween0And100() {
  assert.throws(() => graderModule.getLetterGrade(-1, "Input must be between 0 and 100."));
  assert.throws(() => graderModule.getLetterGrade(101, "Input must be between 0 and 100."));
}
```

```
testGetLetterGrade();
testGetGradeOnlyAcceptsNumericalValues();
testGetGradeOnlyAcceptsValuesBetween0And100();
```

```
console.log("All tests passed!");
```

```
// src/grader.js
```

```
function getLetterGrade(score) {
}
```

```
module.exports = {  
  getLetterGrade: getLetterGrade  
};
```

Step 2 (Green): Write the implementation code to pass the unit tests.

```
function getLetterGrade(score) {  
  if (typeof score !== "number") {  
    throw Error("Input must be a number.");  
  }  
  
  if (score < 0 || score > 100) {  
    throw Error("Input must be between 0 and 100.");  
  }  
  
  switch (score) {  
    case 100:  
    case 90:  
      return "A";  
    case 80:  
      return "B";  
    case 70:  
      return "C";  
    case 60:  
      return "D";  
    default:  
      return "F";  
  }  
}  
  
module.exports = {  
  getLetterGrade: getLetterGrade  
};
```

The `getLetterGrade` function has one formal parameter and, in the body, there are two one-way selections (if statements). The first one-way selection evaluates the expression `typeof score !== "number"`. If the type of the formal parameter `score` is a numerical value, the evaluation passes (true) and execution is returned to the next block of code. If the evaluation does not pass (false), an `Error` object is thrown (covered in the next section) with a message of "Input must be a number". The next one-way selection evaluates the expression `score < 0 || score > 100`. If the evaluation passes, an `Error` object is thrown with a message of "Input must be between 0 and 100". If the evaluation does not pass, execution is returned to the next block of code. In the switch statement, we are evaluating

the score formal parameter. For scores that are 100 and 90, a letter grade of an “A” is returned. For scores that are 80 a letter grade of a “B” is returned. For scores that are 70 a letter grade of a “C” is returned. For scores that are a 60 a letter grade of a “D” is returned. And, for all other scores a letter grade of an “F” is returned.

Let’s rerun the unit tests to see if they pass (npm test). The results should be “All tests passed!”.

Step 3 (Refactor): The code from this example is simple enough, but it does not handle a range of values (it only checks for specific values). Let’s refactor the program so the code can handle a range of values.

```
function getLetterGrade(score) {  
  if (typeof score !== "number") {  
    throw Error("Input must be a number.");  
  }  
  
  if (score < 0 || score > 100) {  
    throw Error("Input must be between 0 and 100.");  
  }  
  
  if (score >= 90) {  
    return "A";  
  } else if (score >= 80) {  
    return "B";  
  } else if (score >= 70) {  
    return "C";  
  } else if (score >= 60) {  
    return "D";  
  } else {  
    return "F";  
  }  
}  
  
module.exports = {  
  getLetterGrade: getLetterGrade  
};
```

Now, let’s rerun the tests to see if they pass (npm test). Assuming your code looks like mine, the tests will pass. Our last step is to build the index.js file to demonstrate the modules usage.

```
// src/index.js
```



```
const graderModule = require("./grader");

function main() {
  // demonstrate the usage of our module
  console.log("Your letter grade for 100 is: " + graderModule.getLetterGrade(100));
  console.log("Your letter grade for 89 is: " + graderModule.getLetterGrade(89));
  console.log("Your letter grade for 72 is: " + graderModule.getLetterGrade(72));
  console.log("Your letter grade for 61 is: " + graderModule.getLetterGrade(61));
  console.log("Your letter grade for 59 is: " + graderModule.getLetterGrade(59));
}

main();
```

In the next section, we will explore how to handle errors in Node.js and how to create custom error objects.

Error Handling in Node.js

In the context of software development, error handling is the process of responding to and managing errors in a program. There are several reasons why errors can happen, such as incorrect input, unforeseen circumstances, or unavailability of resources. JavaScript, like most programming languages, uses try/catch blocks to handle errors. The basic syntax for try/catch blocks is:

```
try {
  // riskyFunction represents a function that may throw an error
  const result = riskyFunction();
} catch (err) {
  console.error("An error occurred: " + err.message);
}
```

In this example, `riskyFunction` represents a function that may throw an error. As you can see from this code block, a try block is used to “try” the function call and a catch block is used to “catch” potential errors. `console.error` is used to print the error message to the process.stderr. Notice in the catch block there is a single formal parameter named `err`. This is an industry standard best practice and a naming convention used throughout the remaining chapters in this textbook. Anytime your code throws an error, the execution will immediately move to the catch block.

Error handling is important because it helps to ensure that the code you are writing continues to execute when unexpected conditions occur. It also provides feedback to developers and end-users, which makes troubleshooting and debugging the cause of an error easier.

Consider the following function:

```
function divide(x, y) {  
  if (y === 0) {  
    throw new Error("Cannot divide by zero");  
  }  
  return x / y;  
}
```

divide is a function with two formal parameters: x and y. In the body of the function a one-way selection statement is used to determine if the second formal parameter y is 0. If the evaluated expression passes, an anonymous error object is thrown. In JavaScript Error, is a built-in object that represents a runtime error. When you use the syntax new Error(), you are creating a new anonymous error object. The value used as the actual parameter during the creation of the error object becomes the error message: new Error("Cannot divide by zero". This message is accessible by calling the message property on the caught error object.

```
catch (err) {  
  console.error(err.message); // property of the caught error object.  
}
```

The Error object has two main properties:

1. **message:** A readable description of the error. This is set by passing a string to the Error constructor.
2. **name:** A string that represents the error type or the name of the constructor function that created the error. For example, in the case of an Error object, name would point to a value of "Error".

JavaScript also includes several other built-in error types:

1. **TypeError:** This error is thrown when an operation could not be performed on a value of an unexpected type. This can occur when you assign a variable as a number, but call it as a function. For example,
let x = 10;
x();

Calling x() will throw TypeError: x is not a function. This makes sense, because we defined x as a numerical value but it's being treated as a function.

2. **RangeError:** This error is thrown when a value is not within the expected range. This typically happens when you try create an array with a negative length or call a function with actual parameters that are valid but outside the expected range:

Consider the divide function we defined earlier. You could throw a `RangeError` instead of an anonymous `Error`, because 0 would be considered “outside the range” of acceptable values.

3. **ReferenceError:** This error is thrown when you try to access a variable or property on an object that either does not exist or has not been defined.

```
console.log(myName); // throws ReferenceError
```

This line of code throws: `ReferenceError: myName is not defined`. This makes sense, because the `myName` variable does not exist (not defined).

4. **SyntaxError:** This error is thrown when JavaScript encounters code that does not follow the correct syntax rules.

```
let while = 10; // SyntaxError: Unexpected token 'while'
```

This line of code throws: `SyntaxError: Unexpected token 'while'`. This occurs, because the variable `while` is a reserved keyword in JavaScript and thus this statement breaks the syntax rules in JavaScript.

Besides JavaScript’s built-in `Error` objects, you can also create custom error objects. You do this by defining a new class that extends the `Error` class.

```
class CustomError extends Error {  
  constructor(message) {  
    super(message);  
    this.name = "CustomError";  
  }  
}
```

```
throw new CustomError("This is a custom error");
```

In this code example, a class named `CustomError` is created that extends JavaScript’s built-in `Error` object. The constructor accepts a formal parameter named `message` (the error message to be thrown) and then it calls the base classes constructor (`Error`) through the `super` function with the passed in formal parameter. In the body of the constructor, the `name` property is set to `CustomError` to represent the error’s type. Running this code will throw: `CustomError: This is a custom error`. To see this in a real-life example, consider our divide function.

```

class DivideByZeroError extends Error {
  constructor(message) {
    super(message);
    this.name = "DivideByZeroError";
  }
}

function divide(x, y) {
  if (y === 0) {
    throw new DivideByZeroError("Cannot divide by zero");
  }
  return x / y;
}

function main() {
  try {
    const result = divide(10, 0);
    console.log("The result is: " + result);
  } catch (err) {
    console.error("An error occurred: " + err.message);
  }
}

main();

```

In this example, a class named `DivideByZeroError` is created that extends JavaScript's built-in `Error` object. Then, the `divide` function uses this class to throw a divide by zero error with the appropriate error message. Running this program will print: "An Error occurred: Cannot divide by zero".

Creating custom error types is most useful when you want to differentiate between the types of errors being caught in a catch block or when you need to add more information to a built-in JavaScript error object.

Unlike most programming languages, JavaScript does not support multiple catch blocks. Instead, if you want to handle errors differently, you can either use two-way conditional statements (if...else statements) or nested try/catch blocks.

```

function main() {
  try {
    const result = divide(10, 0);
    console.log("The result is: " + result);
  }
}

```

```

catch (err) {
  if (err instanceof DivideByZeroError) {
    console.log("Error: " + err.message);
  } else {
    console.error("An unexpected error occurred")
  }
}
}
}

```

In this version of the main function, a two-way conditional statement is used to determine if the caught error object is an instance of the `DivideByZeroError` object. If true, the error message is printed to the `process.stdout`. If false, the error is printed to the `process.stderr`. The ability to respond to errors differently is very powerful and useful in most programs. In general, you should stick to JavaScript's built-in Error objects for handling errors and use custom errors when you need more flexibility.

In the next section we will look at test runners and Jest, which is a more robust framework for writing unit tests using TDD.

Introduction to Jest and Test Runners

Jest is an open-source JavaScript testing framework developed by Facebook. It is a “zero-configuration” framework, built on the philosophy of simplicity and extensibility. To get started with Jest, you do not need to install or configure a lot of third-party tools. You simply add Jest and begin writing unit tests. The advantages of Jest are:

1. **Zero-configuration:** Unlike most JavaScript test frameworks, Jest works “out of the box” with minimal setup. The simplicity of Jest makes it a great choice for new and existing projects. Simply install and begin writing unit tests.
2. **Mocking support:** Jest includes built-in support for mocking. Mocking in the context of software testing is a process of replacing actual functionality and/or data with fake functionality and/or data. This is done by creating a “mock” object that simulates the behavior and/or data of the real object. Mocking is often used to mock databases, files, and APIs where you want to test the functionality of something without dealing with the side effects of using real data.
3. **Snapshot testing:** Jest can capture “snapshots” of your components and data structures and then use those “snapshots” to compare against future versions of your code. Snapshot testing is a strategy used primarily for User Interface (UI) development and testing. Jest takes a “snapshot” of the output of a component (think HTML div) and saves it for future tests. Snapshot testing is used heavily in client-side frameworks like Angular and React.

4. **Parallel test execution:** Tests written in Jest are executed using parallelization. This improves the speed in which unit tests are executed.
5. **Coverage report:** Jest includes built-in support for generating test coverage reports. Test coverage reports are a way for you to keep track of which lines of code are being executed when unit tests are run. The results are a quantitative metric that illustrates how your code is being covered in each unit test. Reports typically include the following types of coverage: line coverage, statement coverage, branch coverage, and function coverage. When coverages are broken down in such a granular level, developers can pinpoint the parts of their code that are not being tested.
6. **Timer mocks:** Testing code that uses `setTimeout`, `setInterval`, and the `Date` object are especially difficult in JavaScript. With Jest, you can test their functionality using timer mocks. Timer mocks is a built-in feature in Jest that allows you to control JavaScript timing functions. You can “fast forward” in specific points in time to test delayed functionality. Imagine you are building a notification system and needed to test notifications. You could use timer mocks to test the code blocks where there is a delay in the response from a notification.
7. **Async testing:** Testing asynchronous code can be extremely difficult. Jest simplifies this process by providing built-in support for testing asynchronous operations.
8. **Extensibility:** You can use Jest to test Node.js code, React/Angular/Vue components, and JavaScript code. In other words, you can test all parts of a full-stack JavaScript project.

Let's create a new folder named `jest-tests`. In this folder we will practice using Jest by creating several simple unit tests. Once the folder has been created, open a new terminal window and navigate to where you saved the folder and enter the command

`npm init -y`

Next, create a new file named `index.spec.js`. To install Jest, enter the following command:

`npm install --save-dev jest`

This command will add Jest as a `devDependency`. In your `package.json` file, we will need to configure Jest. The simplest way to do this is to add a “`jest`” section to the file. We will also need to update the test script to use Jest.

```
{  
  "name": "jest-tests",  
  "version": "1.0.0",
```

```

"description": "",
"main": "index.js",
"jest": {
  "verbose": true
},
"scripts": {
  "test": "jest"
},
"keywords": [],
"author": "",
"license": "ISC",
"devDependencies": {
  "jest": "^29.7.0"
}
}

```

With these configurations in place, we can begin writing Jest unit tests.

1. Testing a simple math operation:

```

test("adds 5 + 5 to equal 10", () => {
  expect(5 + 5).toBe(10); // toBe is a matcher
})

```

2. Testing a simple array operation:

```

// Testing a simple array operation
test("array contains admin", () => {
  const roles = ["admin", "user", "guest"];
  expect(roles).toContain("admin"); // toContain is a matcher
})

```

3. Testing a simple string operation:

```

// testing a simple string operation
test("converts a string to uppercase", () => {
  const str = "web 340 node.js";
  const uppercase = str.toUpperCase();
  expect(uppercase).toBe("WEB 340 NODE.JS");
})

```

4. Testing a simple object operation:

```

// testing a simple object operation

```

```
test("object assignment", () => {
  const book = {title: "Pragmatic Node.js"};
  book.author = "Professor Krasso";
  expect(book).toEqual({
    title: "Pragmatic Node.js",
    author: "Professor Krasso"
  });
})
```

5. Testing a simple Boolean operation:

```
// testing a simple boolean operation
test("boolean assignment", () => {
  const is2024 = true;
  expect(is2024).toBeTruthy();
})
```

6. Testing a custom function:

```
// testing a custom function
function add(a, b) {
  if (typeof a !== "number" || typeof b !== "number") {
    throw Error("Inputs must be a number");
  }
  return a + b;
}

test("adds 2 + 2 to equal 4", () => {
  expect(add(2, 2)).toBe(4);
})

test("throws an error when a non-number is used", () => {
  expect(() => add("2", 2)).toThrow("Inputs must be a number");
})
```

In this code example, the second test is testing whether the function is throwing an error message for non-number actual parameters. The first actual parameter in the call to the expect function is an anonymous function (callback function) and is needed so Jest can control the behavior of the function. If you were to omit the first actual parameter, Jest would not be able to catch the thrown error to determine if it matches what we are testing. Instead, the thrown error would halt the execution of the code. This is a common practice that is used in testing when you want to test a function that throws an error.

Running these tests (assuming you updated the test script in the package.json file) is as simple as entering the following command:

npm test

Which, produces the following results (7-unit tests and all 7-unit tests passed):

```
> jest-tests@1.0.0 test
> jest

PASS ./index.spec.js
  ✓ adds 5 + 5 to equal 10 (1 ms)
  ✓ array contains admin
  ✓ converts a string to uppercase
  ✓ object assignment (1 ms)
  ✓ boolean assignment
  ✓ adds 2 + 2 to equal 4
  ✓ throws an error when a non-number is used (2 ms)

Test Suites: 1 passed, 1 total
Tests:       7 passed, 7 total
Snapshots:   0 total
Time:        0.157 s, estimated 1 s
Ran all test suites.
```

In the initial example, we have a total of 7 unit tests, all of which have passed successfully. This is a good start, but it is also important to understand how our testing framework behaves when tests fail. This understanding can help us write better tests and help us debug when things go wrong. To illustrate this, let's intentionally introduce a failure in our tests. We will do this by changing the expected error message in one of our tests:

```
test("throws an error when a non-number is used", () => {
  expect(() => add("2", 2)).toThrow("bad input");
});
```

In this updated code example, we are still testing the add function with invalid input (a string and a number). However, we have changed the error message we expect to be thrown from "Inputs must be a number" to "bad input". Since our add function has not been changed and still throws the original error message, this test is now expected to fail.

Let's rerun our unit tests to see how Jest responds to this failure. We can do this by running the test command (**npm test**) in our terminal window. The expected output from Jest should now show that 6 tests have passed and 1 test has failed. Jest is also expected to provide a detailed explanation for the test failure, including the difference between the expected and actual outcomes. This feature of Jest is extremely helpful, because it shows us what went wrong and where.

Expected substring: "bad input"

Received message: "Inputs must be a number"

```
● throws an error when a non-number is used

expect(received).toThrow(expected)

Expected substring: "Bad input"
Received message: "Inputs must be a number"

   38 | function add(a, b) {
   39 |   if (typeof a !== "number" || typeof b !== "number") {
>  40 |     throw Error("Inputs must be a number");
      |           ^
   41 |   }
   42 |   return a + b;
   43 | }

at Error (index.spec.js:40:11)
at add (index.spec.js:50:16)
at Object.toThrow (index.spec.js:50:29)

   48 |
   49 | test("throws an error when a non-number is used", () => {
>  50 |   expect(() => add("2", 2)).toThrow("Bad input");
      |                               ^
   51 | })

at Object.toThrow (index.spec.js:50:29)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 6 passed, 7 total
Snapshots:   0 total
Time:        0.179 s, estimated 1 s
Ran all test suites.
```

Jest prints the code that failed, the failing line number, and the differences between the Expected and Received inputs. This extra level of detail makes it easier to discern why a unit test has failed. Understanding how to read and interpret these failure messages is a key skill in test-driven development. It allows us to quickly identify and fix issues in our code. With a basic understanding of Jest, let's take a look at a comparison between Jest and the assert module from Node.js:

1. **Functionality:** Jest is a full-fledged testing framework that includes features like mocking, snapshot testing, and test coverage reports. The assert module is a simple assertion library that allows you to test code segments. It does not include additional "test features" like mocking and test coverage reports.
2. **Ease of use:** Arguably, the assert module is easier to use, because you do not have to learn an entirely new JavaScript framework. Jest prides itself on being an "easy to use, out of the box" framework, but in my experience, it does take longer to master. And, personally, it is better to avoid using third-party frameworks, because if the creator of that framework abandons it, you are left "up the creek without a paddle."
3. **Community and ecosystem:** Jest has a large community, ecosystem, and is backed by Facebook. There are plenty of resources available and it is used heavily

in React development. The Node.js assert module is a built-in feature of Node, so as long as Node.js is around, the assert module will be around.

4. **Flexibility:** Jest is highly flexible and can be configured to satisfy a long list of testing requirements. Most full-fledged test framework provide this level of flexibility. On the other hand, the assert module is a simple module that has only be designed to handle assertion-level testing.
5. **Integration with other tools:** Jest integrates with many other open-source, third-party tools. Given its large backing and community support, many organizations have written plug-ins to use Jest with their frameworks. Angular, TypeScript, and Vue are just some examples of this. The Node.js assert package does not have these types of integrations and as of this writing (Node.js 20) no immediate plans have been communicated to include such integrations.

The decision on which tool to use depends heavily on the application you are budling and the individual requirements of your team. For simple unit tests, the assert module is a great tool to use. For large, more complex projects where you are testing large files and full-stack applications, a test framework like Jest is more appropriate.

Test runners, are tools that automatically discover and execute unit tests in a project. Test runners are used heavily in DevOps (Development Operations) and continuous integrations/continuous delivery (CI/CD) pipelines. They allow the execution of tests to be automatically ran each time a developer makes changes to a code base or deploys their application to a different environment. Their role in automation is as follows:

1. **Test Discovery:** Test runners are automated tools that can identify and execute tests in your project based on certain criteria, such as file names, locations, or code annotations. This eliminates the need for manual invocation of tests or test suites, saving valuable time.
2. **Test Execution:** Test runners can execute your unit tests in a predefined order, including sequential, asynchronous, or parallel. They are also flexible in that they allow you to decide how to handle failures, including setting the severity of the failure to determine how subsequent tests are handled.
3. **Result Reporting:** After tests have been run, test runners can generate detailed reports that outline all passes and failures, along with explanations of why failures occurred. These reports can be displayed in various formats or integrated in other tools or third-party applications.
4. **Integration with Other Tools:** Test runners can be integrated with other tools in your project or a DevOps pipeline. For instance, they can be set to execute code each time a developer pushes code to a branch in Git and can also be configured to

run tests before deploying your application to a production environment. Moreover, they can integrate other test suites and tools.

5. **Automation:** Test runners are essential for achieving automation in your application, particularly for building CI/CD pipelines and adopting a DevOps culture. The ability to run tests automatically based on a specific criterion promotes a high level of automation, which saves development resources and fixed costs.

Jest itself is an example of a test runner. In the next section, we will take a look at test coverages.

Introduction to Test Coverage

Test coverage is a metric that is used to evaluate the effectiveness of testing efforts in a project. It involves tracking various aspects of testing processes, such as a number of functions called, the number of program statements executed, the number of if...else statements in the project, the number of lines of code executed, and the number of objects in the project. Test coverage is crucial for several reasons:

1. **Identify Untested Parts of Code:** Test coverage aids in the identification of code sections in your project that are not subject to unit testing. This enables you to determine whether additional unit tests are necessary to cover the remaining code segments.
2. **Increase Confidence:** Having a high-test coverage instills confidence in developers due to increased project stability and reliability. It also ensures that any changes made to the project, such as code refactoring, do not negatively impact existing features.
3. **Improve Code Quality:** Writing unit tests often results in better code quality since it compels the developers to think more critically about the code they write and any potential errors that may arise as a result. In short, writing unit tests can enhance a developer's skills.
4. **Regression Detection:** Adequate test coverage facilitates early detection of bugs during the development phase, preventing errors and bugs from surfacing at deployment. This is especially true if developers are adhering to TDD principles.

While high test coverage does not guarantee a bug-free application, it does force developers to adopt a different approach to project development. It is crucial for developers to write meaningful unit tests that align with project requirements. Additionally, the accuracy and quality of unit tests are essential as poor-quality tests render test coverage meaningless.

To generate a test coverage report in Jest, use the following command:

`npx jest --coverage`

This will create a coverage report in a directory named `coverage` of your project's root directory. For example, if I ran this command on a directory named `gardening-app`, the folder would be placed at the root of the `gardening-app` directory. The report also includes an HTML file that can be opened in a web browser.

For test coverage to work in Jest, the code needs to be in a module and that module must be imported in the test file. For example,

Folder Structure:

```
favorite-composers
  src
    composer.js
  test
    composer.spec.js
package.json
```

```
// package.json
{
  "name": "favorite-composers",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "jest --coverage"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "jest": "^29.7.0"
  }
}

// composer.spec.js
"use strict";

test("should return a list of composers", () => {
  const composers = require("../src/composer");
  expect(composers.getComposers()).toEqual(
    expect.arrayContaining([
```

```

    expect.objectContaining({
      firstName: "Ludwig",
      lastName: "Beethoven",
      genre: "Classical"
    }),
    expect.objectContaining({
      firstName: "Wolfgang",
      lastName: "Mozart",
      genre: "Classical"
    }),
    expect.objectContaining({
      firstName: "Johann",
      lastName: "Bach",
      genre: "Classical"
    }),
    expect.objectContaining({
      firstName: "Joseph",
      lastName: "Haydn",
      genre: "Classical"
    })
  ])
)
})

test("should return a composer by last name", () => {
  const composers = require("../src/composer");
  expect(composers.getComposerByLastName("Beethoven")).toEqual(
    expect.objectContaining({
      firstName: "Ludwig",
      lastName: "Beethoven",
      genre: "Classical"
    })
  )
})

// composer.js
"use strict";

const composers = [
  {
    firstName: "Ludwig",
    lastName: "Beethoven",
    genre: "Classical",
  },

```

```

{
  firstName: "Wolfgang",
  lastName: "Mozart",
  genre: "Classical",
},
{
  firstName: "Johann",
  lastName: "Bach",
  genre: "Classical",
},
{
  firstName: "Joseph",
  lastName: "Haydn",
  genre: "Classical"
}
];

function getComposers() {
  return composers;
}

function getComposerByLastName(lastName) {
  console.log("getComposerByLastName: " + lastName);

  for (let composer of composers) {
    if (composer.lastName === lastName) {
      return composer;
    }
  }
}

module.exports = {
  getComposers: getComposers,
  getComposerByLastName: getComposerByLastName
};

// Coverage report (1 page)

```

All files

100% Statements 7/7 50% Branches 1/2 100% Functions 2/2 100% Lines 7/7

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

Filter:

File	Statements	Branches
composer.js	100%	50%

TDD (Red – Green – Refactor) principles were used to create this project. First, tests were written in the `composer.spec.js` file. Then a skeleton module was created with an array of composer objects and two empty functions. The tests were ran and they initially failed. Finally, the module was updated to pass the unit tests.

To generate a coverage report, run `npm test` with the coverage option. This will create a folder named `lcov-coverage`. Inside this folder, you will find an HTML file named `composer.js.html`. Open this file in your default browser to see the coverage report displayed in an HTML format. You can click on the `composer.js` link to drill down to a more granular level in the report. The report provides information about the number of executed statements, branches, functions, and lines. You can filter the report to see results by a specific criterion. Additionally, each column in the report allows you to sort the data in either ascending or descending order.

In conclusion, test coverage is a powerful tool for testing and debugging Node.js projects.

Programming Exercises

Welcome to the Interplanetary Science Fair! In this assignment, you will be creating a module that calculates the distance between two planets in our solar system. You will be using Test Driven Development (TDD) principles and the Node.js assert library to guide your development process.

For this assignment, we will be using Astronomical Units (AU) to measure distances. 1 AU is the average distance between the Sun to the Earth. You will need to research and find the approximate distance of the planets from the Sun in AU. Remember, these are average values, as the actual distances vary as the planets orbit the Sun.

Instructions:

1. Create a new folder for your project following this structure:

distance-calculator


```
src
  distance-calculator.js
test
  distance-calculator.spec.js
package.json
```

2. In your package.json file, add a test script. Remember to use strict mode.
3. Your distance-calculator.js module should include a function that calculates the distance between two planets in AU, given their distances from the sun in AU.
4. For the function in your module, write three (3) different unit tests in the distance-calculator.spec.js using the Node.js assert library. Each test should set up a scenario, call the function with specific inputs, and then assert that the function's output is correct.
5. Each test should be wrapped in a function named testFunctionDescription, where functionDescription is a brief description of what the test does. For example, a test that checks if the function correctly calculates the distance between Earth and Mars could be named testEarthToMars.
6. Inside each test function, use a try/catch block to run the test and catch any errors. If the test passes, print a message indicating it passed and return true. If the test fails, print a message indicating the test failed and return false. Below is an example to get you started:

```
function testEarthToMars() {
  try {
    assert.strictEqual(calculateDistance('Earth', 'Mars'), expectedValue);
    return true;
  } catch (error) {
    console.error(`Failed testEarthToMars: ${error.message}`);
    return false;
  }
}
```

7. Use TDD principles to guide your development process. Write your tests first, then write the code to make the tests pass.

Grading:

You will be awarded 20 points for each passing unit test, for a total of 60 points. If a test does not pass, you will not receive points for that test. Use the provide starter files to complete the assignment.

Chapter 4. Node Event System

Chapter Overview

EventEmitter is a core feature of Node.js that handles the communications and interactions between objects in a program. It is a module based on event-driven programming where events are used to signal some type of behavior based on some type of action. For example, clicking on a button in a web page or a web server responding to an API request. Through the use of the EventEmitter module, objects in a Node.js program can emit events and respond to them in an efficient manner.

Learning Objectives

By the end of this chapter, you should be able to:

- Define what event-driven programming is.
- Explain how Node.js uses events.
- Identify strategies for handling errors in an event-driven environment.
- Build Node.js programs using custom events with EventEmitter.

Introduction to Node.js Event-Driven Architecture

In event-driven programming, the program's flow is determined by system events such as user-generated actions, sensor outputs, or interactions from other programs. These events serve as triggers that dictate the execution of the program. The key points about event-driven programming are outlined below:

1. **Event Listeners:** Listeners are an essential part of event-driven applications as they are responsible for detecting the occurrence of specific system or user-generated events. These events can range from user actions such as mouse clicks or key presses to system-generated events like webpage loads or system ticks.
2. **Event Handlers:** In event-driven programming, an event handler is a specific block of code that is executed when a particular event occurs. This handler is responsible for handling events that are generated as a result of some action, such as clicking a button on a webpage. For instance, an event handler might execute a function that changes the color of a div when a button is clicked. The function that changes the color of the div is referred to as the event handler for the on-click event.
3. **Asynchronous Behavior:** Event-driven programming is asynchronous by nature, which makes it possible for programs to accommodate large numbers of users and multiple tasks simultaneously. With the help of event-driven programming, a program can respond to new events in a more efficient and precise manner. This results in the development of faster and more responsive applications.

4. **Non-Blocking:** Event-driven programming allows applications to execute tasks without blocking the program's execution while waiting for an event. In other words, the program can continue processing other tasks while events are taking place, resulting in a more efficient use of system resources.
5. **Common in Web Applications:** Event-driven programming is widely used in Graphical User Interfaces (GUI) and web applications, where events such as mouse clicks or mouse movements are common. Moreover, events are commonly utilized in sever-side development to efficiently handle requests and responses from an API.

The core architecture of Node.js relies heavily on the use of events, and numerous built-in modules produce objects that are instances of the EventEmitter class. But, before we explore the EventEmitter class, let's take a look at how Node.js uses events in its core architecture. Throughout the Node.js architecture, events are used to initiate and respond to actions.

Consider the following analogy (and, yes, I am showing my age and no, I am not an advocate for either position, it is just an analogy to explain events), imagine you are in school and the principal of your school uses a PA system to make morning announcements. When the principle wants to make an announcement, they turn on the microphone (this is like emitting an event in Node.js). Now, let's say you are in a classroom and the teacher says, when the principle begins the Pledge of Allegiance, everyone must stand up and place their hand over their hearts. Your teacher has effectively set up a rule or a response to the Pledge of Allegiance announcement (this is like setting up an event listener in Node.js).

In other words, when the principle makes the Pledge of Allegiance announcement (emits an event) your class stands up (the event listener responds).

In this way, Node.js uses events to handle things that happen in a Node.js program. This includes data coming from a database, clicking a button in a browser window, reading and writing data to a file, and responding to network requests.

In the next section will continue our exploration of the EventEmitter API by practicing creating instances of the EventEmitter class, emitting events, and listening for events.

Understanding the EventEmitter API

The EventEmitter API is a built-in class from the events module in Node.js. It is a core component of the Node.js architecture and is used heavily in Node.js to handle events. Here is a brief introduction:

1. **EventEmitter Class:** Node.js has a built-in module called events where the EventEmitter class is defined. This class is used to emit events that an event

listener (covered below) can respond to. The following syntax illustrates how to use the EventEmitter class.

```
const EventEmitter = require('events');  
const myEmitter = new EventEmitter();
```

In this code example, a require statement is used to import the EventEmitter from the events module. The second line of code creates a new instance of the EventEmitter object and assigns it to a variable named myEmitter.

2. **Event Emission:** Events can be emitted using the emit method off of the EventEmitter object. This method allows you to register predefined events and to pass actual parameters to a listener function (covered below). Consider the following code snippet:

```
myEmitter.emit("classStarted", "week-4", "Event Emitters");
```

In this code example, an event named “classStarted” is emitted with two actual parameters, “week-4” and “Event Emitters”.

3. **Event Listening:** The on method from the instance of the EventEmitter object is used to register an event listener.

```
// Listen for the classStarted event  
myEmitter.on("classStarted", (week, topic) => {  
  console.log(` This week is ${week} and the topic is ${topic}` );  
})
```

In this code example, the on method is used to listen for the classStarted event. Next, an anonymous function is used to handle the response from the emitted event. In this case, because two actual parameters were provided when the event was emitted, they are captured in the parenthesis of the anonymous function. Effectively, this gives us access to the actual parameter values in the body of the listener. Lastly, we print the week and topic to the console.

Always write the listener before you emit the event. In other words, in our example, the code for the listener would appear above the code for the emitter. Otherwise, when an event is triggered, there would be no listeners to respond to the event.

4. **Asynchronous Event Handling:** Node.js handles events asynchronously. This makes Node.js highly efficient for non-blocking event-driven programs. For example, conferencing and messaging applications. This is also useful for handling I/O tasks, like reading files, making database queries, or responding to network requests.

5. **Built-in Modules:** Many of the built-in modules in Node.js use events are/or return objects that are an instance of the EventEmitter class. Modules like fs and http (covered in later chapters) are just two examples of built-in modules that use the EventEmitter class.
6. **Error Events:** Node.js uses events for error handling. The “error” event is emitted when an error occurs during the execution of a Node.js program. You should always handle these events in your Node.js program. To do this, consider the following code snippet.

```
myEmitter.on("error", (err) => {  
  console.error("There was an error", err);  
})
```

In this code example, a new event listener is used that listens for “error” events. Next an anonymous function with an actual parameter of err is used to capture the thrown error event and in the body of this function a call to console.error is used to print an error message with the err object.

When an event is registered with the on method, that listener is invoked each time the event is emitted. Using the once method, we can control this behavior and only respond once to a system event. This is especially useful when an event is only expected to occur once during an operation in your application. For example, let’s say you are building a web server and you only want to log when the server starts for the first time, you can use the once method to set up a listener to only write the message to a log file once. Regardless of how many times the “listening” event occurs. Another example, can be seen in establishing a connection to a database. You could use the once method to ensure only a single connection to the database is active when a “connect” event is emitted. Below is a simple example of how to use the once method.

```
const EventEmitter = require("events");  
const myEmitter = new EventEmitter();  
  
myEmitter.once("turnAge21", () => {  
  console.log("You are now 21, welcome to the club!");  
});  
  
myEmitter.emit("turnAge21");  
myEmitter.emit("turnAge21");  
myEmitter.emit("turnAge21");  
myEmitter.emit("turnAge21");
```

in this code example, even though the “turnAge21” event is emitted four times, it will only be printed once in the console. That is, you can only turn 21 once in your lifetime (oh, to be young again).

In the next section, we will take a look at how to create custom events using the EventEmitter class.

Creating Custom Events with EventEmitter

Learning how to create and emit custom events is critical for the development of highly efficient programs in Node.js. In the previous section, it was explained that events are emitted using the emit method and listeners are registered using the on function. And, in order to pass data to an event, you use formal parameters when setting up the event emitter. Let’s take a look at three real-world examples, where events can be used in a Node.js program.

1. **Saving Account:** Suppose you have a banking application that allows users to deposit and withdraw money from their accounts. You can use the EventEmitter class to emit events for different types of transactions. For instance, you could create a deposit event and emit it whenever a user deposits money into their account. Similar, you could create a withdraw event and emit it whenever a user withdraws money from their account. In addition, you could also create an insufficient funds event and emit it whenever a user tries to withdraw money exceeding their balance. You could use event listeners to update the account balance, send a confirmation message to the user, or trigger alerts for insufficient funds.
2. **Ordering Coffee at a Coffee Shop:** Suppose you are developing a coffee shop application and you want to use the EventEmitter class to handle the coffee ordering process. For instance, when a customer orders a latte, you could emit an “orderPlaced” event. Event listeners could be used to update the order queue, notify the barista to prepare the coffee, or print the receipt. Additionally, you could emit “orderReady” event to inform the customer their order is ready.
3. **Preordering a Game at a Video Store:** Suppose you are building a game store application that allows customers to preorder upcoming games. Whenever a customer preorders a game, you could emit a “gamePreordered” event. Additionally, listeners for this event could reserve a copy of the game for the customer and charge their account for the price of the game. Once the game is released, you could emit another event named “gameAvailable” to notify customers that the game is read for pickup. Conversely, listeners of this event could send a text message or email to the customer.

Let's take a look at how we could implement each of these scenarios in a simple Node.js program.

```
// saving-account.js
const EventEmitter = require("events");

class BankAccount extends EventEmitter {
  constructor() {
    super();
    this.balance = 0;
  }

  deposit(amount) {
    this.balance += amount;
    this.emit("deposit", amount);
  }

  withdraw(amount) {
    if (amount > this.balance) {
      this.emit("insufficientFunds", amount);
    } else {
      this.balance -= amount;
      this.emit("withdraw", amount);
    }
  }
}

const account = new BankAccount(); // create a new BankAccount object

account.on("deposit", (amount) => {
  console.log(`Deposited ${amount}. New balance: ${account.balance}`);
});

account.on("withdraw", (amount) => {
  console.log(`Withdrew ${amount}. New balance: ${account.balance}`);
});

account.on("insufficientFunds", (amount) => {
  console.log(`Attempted to withdraw ${amount}, but only ${account.balance}
  available.`);
});

// perform some actions
account.deposit(100);
```

```
account.withdraw(50);  
account.withdraw(60);
```

This script creates a BankAccount class that extends the EventEmitter. The deposit and withdraw methods emit “deposit” and “withdraw” events. If a withdrawn exceeds the users available balance, an “insufficientFunds” event is emitted. Event listeners are set up to log a message whenever a deposit or withdraw is made. Or, when there is an insufficient funds request. Finally, a call to each method is made to illustrate the events and listeners.

```
// coffee-shop.js  
const EventEmitter = require("events");  
  
class CoffeeShop extends EventEmitter {  
  constructor() {  
    super();  
    this.orderQueue = [];  
  }  
  
  placeOrder(order) {  
    this.orderQueue.push(order);  
    this.emit("orderPlaced", order);  
  }  
  
  completeOrder() {  
    const completeOrder = this.orderQueue.shift();  
    this.emit("orderReady", completeOrder);  
  }  
}  
  
const coffeeShop = new CoffeeShop(); // create a new CoffeeShop object  
  
coffeeShop.on("orderPlaced", (order) => {  
  console.log(` Order placed: ${order}` );  
  console.log(` Order queue: ${coffeeShop.orderQueue.join(", ")}` );  
  // Notify the barista to prepare the coffee  
  // Print the receipt  
});  
  
coffeeShop.on("orderReady", (order) => {  
  console.log(` Order ready: ${order}` );  
  // Inform the customer their order is ready  
});  
  
// Place some orders
```



```
coffeeShop.placeOrder("Latte");
coffeeShop.placeOrder("Cappuccino");
```

```
// Complete an order
coffeeShop.completeOrder();
```

The script creates a CoffeeShop class that extends the EventEmitter. The placeOrder method adds an order to the queue and emits an “orderPlaced” event. The completeOrder method removes an order from the queue and emits an “orderReady” event. Event listeners are set up to log a message whenever an order is placed or ready.

```
// video-store.js
const EventEmitter = require("events");

class GameStore extends EventEmitter {
  constructor() {
    super();
    this.preorders = [];
  }

  preorderGame(game, customer) {
    this.preorders.push({ game, customer });
    this.emit("gamePreordered", game, customer);
  }

  releaseGame(game) {
    const customersToNotify = this.preorders.filter(preorder => preorder.game === game);
    this.emit("gameAvailable", game, customersToNotify);
  }
}

const gameStore = new GameStore();

// Set up listeners
gameStore.on("gamePreordered", (game, customer) => {
  console.log(`Game preordered: ${game} by ${customer}`);
  // Reserve a copy of the game for the customer
  // Charge the customer's account for the price of the game
});

gameStore.on("gameAvailable", (game, customersToNotify) => {
  console.log(`Game available: ${game}`);
  customersToNotify.forEach(({ customer }) => {
```

```

    console.log(` Notify ${customer} that ${game} is ready for pickup `);
    // Send a text message or email to the customer
  });
});

// Preorder some games
gameStore.preorderGame("Marvel Spider-Man 2", "Wolfgang Mozart");
gameStore.preorderGame("Marvel Spider-Man 2", "Richard Wagner");

// Release a game
gameStore.releaseGame("Marvel Spider-Man 2");

```

This script creates a `GameStore` class that extends the `EventEmitter`. The `preorderGame` method adds a preorder to the list and emits a “gamePreordered” event. The `releaseGame` method emits a “gameAvailable” event for each customer who preordered the game. Event listeners are set up to log messages when a game is preordered or becomes available.

In the next section we will review the importance of listening for error events and strategies for handling errors in an event-driven environment.

Error Handling with Events

Listening for events in a Node.js program is crucial for many reasons. Below outlines some of the reasons why listening for events is important:

1. **Error Handling:** It allows you to handle events gracefully. When an error is emitted, you can catch the error and decide how to respond. This gives you the flexibility to either write the error to a log file, halt the execution of the program, report a message to the user, or do nothing.
2. **Preventing Crashes:** In Node.js if an error event is emitted and there is no listener to handle the event, the default action is to exit the program (crash the service). By setting up event listeners to listen for error events, you can prevent the program from crashing.
3. **Debugging:** Error events provide helpful information that you can use to better troubleshoot and debug the code you are writing. This can be an invaluable resource when building large-scale applications.
4. **Reliability:** By handling error events in a consistent manner, the application will become more fault tolerant and reliable. If something goes wrong (program crashes) you can control how the application responds. This provides a more reliable experience to end users and gives you a pathway for deciding how your application should behave.

In an event-driven architecture like Node.js, there are several strategies you can follow for handling errors:

1. **Listen for “error” events:** In Node.js, if an “error” event is emitted you use a listener to handle the event. It is a good practice to always list for the “error” event when using the EventEmitter class or when using a built-in module that returns an instance of the EventEmitter class.

```
eventEmitter.on('error', (err) => {  
  console.error('An error occurred:', err);  
});
```

2. **Use try/catch in synchronous code:** If you are emitting events in response to synchronous operations, you can use try/catch blocks to catch these errors and handle them in a graceful way.

```
try {  
  // Synchronous operation that might throw an error  
} catch (err) {  
  eventEmitter.emit('error', err);  
}
```

3. **Handle errors in callbacks:** If you are working with callback functions and an “error” event occurs, you can emit the error in the if condition of that callback.

```
If (err) {  
  eventEmitter.emit(“error”, err);  
} else {  
  // continue  
}
```

Handling errors in callbacks will be covered in a later chapter.

4. **Use Promises and async/await:** Using promises and async/await is another way you can respond to “error” events. Rejections are emitted in the catch block of the operation.

```
async function asyncOperation() {  
  try {  
    // Asynchronous operation that might throw an error  
  } catch (err) {  
    eventEmitter.emit('error', err);  
  }  
}
```

```
}
```

Handling errors in Promises and async/await will be covered in a later chapter.

In the next section, we will look at how to use TDD with the EventEmitter class.

TDD with EventEmitter

In this section, we will explore how to create unit tests and test programs using the EventEmitter class. This will be accomplished through TDD principles and reusing the code from the BankAccount class.

We will use the following project structure for the revised version of the saving account program.

```
account-app
  src
    bank-account.js
  test
    bank-account-spec.js
package.json
```

To use TDD principals and the Node.js built-in assert library, we use the following approach:

1. Red – Write a failing test

```
// bank-account.spec.js
const assert = require("assert");
const BankAccount = require("../src/bank-account");

function testDeposit() {
  try {
    const account = new BankAccount();
    account.deposit(100);
    assert.strictEqual(account.balance, 100, "The balance should be 100.");

    console.log("The deposit() method passed.");
  } catch (err) {
    console.log("The deposit() method failed.");
  }
}

testDeposit();
```

2. Green – Make the test pass:

```
// src/bank-account.js
const EventEmitter = require("events");

class BankAccount extends EventEmitter {
  constructor() {
    super();
    this.balance = 0;
  }

  deposit(amount) {
    this.balance += amount;
    this.emit("deposit", amount);
  }
}

module.exports = BankAccount;
```

3. Refactor – Code clean up and revise

```
// src/bank-account.js
const EventEmitter = require("events");

class BankAccount extends EventEmitter {
  constructor() {
    super();
    this.balance = 0;
  }

  deposit(amount) {
    if (amount <= 0) {
      this.emit("error", new Error("Deposit amount must be greater than 0.));
      return;
    }
    this.balance += amount;
    this.emit("deposit", amount);
  }
}

// test/bank-account.spec.js
const assert = require("assert");
const BankAccount = require("../src/bank-account");
```

```

function testDeposit() {
  try {
    const account = new BankAccount();
    account.deposit(100);
    assert.strictEqual(account.balance, 100, "The balance should be 100.");

    console.log("The deposit() method passed.");
  } catch (err) {
    console.log("The deposit() method failed.");
    console.log(err);
  }
}

function testNegativeDepositThrowsError() {
  try {
    const account = new BankAccount();
    account.deposit(-100);
  } catch (err) {
    assert.strictEqual(err.message, "Deposit amount must be greater than 0.");
    console.log("The negative deposit amount test passed.");
  }
}

testDeposit();
testNegativeDepositThrowsError();

```

4. Repeat this process for the remaining operations: withdraw and insufficient funds

```

// src/bank-account.js
const EventEmitter = require("events");

class BankAccount extends EventEmitter {
  constructor() {
    super();
    this.balance = 0;
  }

  deposit(amount) {
    if (amount <= 0) {
      this.emit("error", new Error("Deposit amount must be greater than 0."));
      return;
    }
    this.balance += amount;
    this.emit("deposit", amount);
  }
}

```

```

    }

    withdraw(amount) {
      if (amount > this.balance) {
        const error = new Error("Insufficient funds.");
        error.withdrawAmount = amount;
        this.emit("insufficientFunds", error);
        return;
      }
      this.balance -= amount;
      this.emit("withdraw", amount);
    }
  }

  const account = new BankAccount();

  account.on("deposit", (amount) => {
    console.log(` Deposited ${amount}. New balance: ${account.balance}`);
  });

  account.on("withdraw", (amount) => {
    console.log(` Withdrew ${amount}. New balance: ${account.balance}`);
  });

  account.on("insufficientFunds", (error) => {
    console.log(` Attempted to withdraw ${error.withdrawAmount}, but only
    ${account.balance} available.`);
  });

  account.deposit(100);
  account.withdraw(50);
  account.withdraw(60);

  module.exports = BankAccount;

  // bank-account.spec.js
  const assert = require("assert");
  const BankAccount = require("../src/bank-account");

  function testDeposit() {
    try {
      const account = new BankAccount();
      account.deposit(100);
      assert.strictEqual(account.balance, 100, "The balance should be 100.");
    }
  }

```

```

    console.log("The deposit() method passed.");
  } catch (err) {
    console.log("The deposit() method failed.");
    console.log(err);
  }
}

function testNegativeDepositThrowsError() {
  try {
    const account = new BankAccount();
    account.deposit(-100);
  } catch (err) {
    assert.strictEqual(err.message, "Deposit amount must be greater than 0.");
    console.log("The negative deposit amount test passed.");
  }
}

function testWithdraw() {
  try {
    const account = new BankAccount();
    account.deposit(100);
    account.withdraw(50);
    assert.strictEqual(account.balance, 50, "The balance should be 50.");

    console.log("The withdraw() method passed.");
  } catch (err) {
    console.log("The withdraw() method failed.");
    console.log(err);
  }
}

function testInsufficientFundsWithdraw() {
  try {
    const account = new BankAccount();
    account.deposit(100);
    account.withdraw(200);
  } catch (err) {
    assert.strictEqual(err.message, "Insufficient funds.");
    console.log("The insufficient funds withdraw test passed.");
  }
}

testDeposit();

```



```
testNegativeDepositThrowsError();
testWithdraw();
testInsufficientFundsWithdraw();
```

To use TDD principals and Jest, we use a similar approach:
Folder structure: same as the previous example

1. Red – Write a failing test:

```
const BankAccount = require("../src/bank-account");

test("deposit should increase the balance by the deposit amount", () => {
  const account = new BankAccount();
  account.deposit(100);
  expect(account.balance).toBe(100);
});
```

2. Green – Make the test pass:

```
// src/bank-account.js
const EventEmitter = require("events");

class BankAccount extends EventEmitter {
  constructor() {
    super();
    this.balance = 0;
  }

  deposit(amount) {
    this.balance += amount;
  }
}

module.exports = BankAccount;
```

3. Refactor – Clean up the code:

```
// src/bank-account.js
const EventEmitter = require("events");

class BankAccount extends EventEmitter {
  constructor() {
    super();
    this.balance = 0;
  }
}
```

```

    }

    deposit(amount) {
      if (amount <= 0) {
        this.emit("error", new Error("Deposit amount must be greater than 0.));
        return;
      }
      this.balance += amount;
    }
  }
}

module.exports = BankAccount;

// test/bank-account.spec.js
const BankAccount = require("../src/bank-account");

test("deposit should increase the balance by the deposit amount", () => {
  const account = new BankAccount();
  account.deposit(100);
  expect(account.balance).toBe(100);
});

test("deposit should emit an error if the deposit amount is less than or equal to 0", () => {
  const account = new BankAccount();
  expect.assertions(1);
  account.on("error", (err) => {
    expect(err).toBeInstanceOf(Error);
  });
  account.deposit(-100);
});

```

4. Repeat the process for the remaining operations: withdraw and insufficient funds.

```

"use strict";

const BankAccount = require("../src/bank-account");

test("deposit should increase the balance by the deposit amount", () => {
  const account = new BankAccount();
  account.deposit(100);
  expect(account.balance).toBe(100);
});

```

```

test("deposit should emit an error if the deposit amount is less than or equal to
0", () => {
  const account = new BankAccount();
  expect.assertions(1);
  account.on("error", (err) => {
    expect(err).toBeInstanceOf(Error);
  });
  account.deposit(-100);
});

```

```

test("withdraw should decrease the balance by the withdraw amount", () => {
  const account = new BankAccount();
  account.deposit(100);
  account.withdraw(50);
  expect(account.balance).toBe(50);
});

```

```

test("withdraw should emit an error if the withdraw amount is greater than the
balance", () => {
  const account = new BankAccount();
  expect.assertions(2);
  account.on("insufficientFunds", (err) => {
    expect(err).toBeInstanceOf(Error);
    expect(err.withdrawAmount).toBe(150);
  });
  account.deposit(100);
  account.withdraw(150);
});

```

The code for the bank-account.js file is excluded from this step, because it is identical to what we used in the assert library example.

Programming Exercises

In this assignment you will use TDD principles to create a taco stand event emitter and a command-line interface program to interact with it.

Use the following folder structure for the project:

```

taco-stand-app
src
  taco-stand.js
  index.js
test
  taco-stand.spec.js

```

package.json

1. In your package.json file, add a test script (for your unit tests) and a start script (for the CLI program). Remember to use strict mode in all JavaScript files (including test files).
2. Create a TacoStandEmitter class module that extends the EventEmitter class from Node.js. This class should have the following methods:
 - a. `serveCustomer(customer: string)`: Emits a “serve” event with the customer as the actual parameter.
 - b. `prepareTaco(taco: string)`: Emits a “prepare” event with the taco as the actual parameter.
 - c. `handleRush(rush: string)`: Emits a “rush” event with the rush as the actual parameter.
3. For the class module, write three (3) different unit tests in `taco-stand-spec.js` using the Node.js assert library. Each test should register an event listener for the emitted event, call the class method, and print either a pass or fail message to the console.
4. Each test should be wrapped in a function named `testFunctionDescription`, where `functionDescription` is a brief description of what the test does. For example, a test for the `serveCustomer` method could be named `testServeCustomer`. Below is an example to get you started:

```
function testPerformAction() {  
  try {  
    // register an event listener for the 'action' event  
    // call the perform action method  
    console.log("Passed testPerformAction");  
    return true;  
  } catch(err) {  
    console.error(` Failed testPerformAction: ${err} `);  
    return false;  
  }  
}
```

5. Inside each test function, use a try-catch block to run the test and catch any errors. If the test passes, print a message indicating it passed and return true. If the test fails, print a message indicating the test failed and return false.

6. Create a CLI program (index.js) that uses the TacoStandEmitter class. The input format should be command followed by a space and the argument. The commands are “serve”, “prepare”, and “rush”.
 - a. Command “serve John”; prints “Taco Stand serves: John”
 - b. Command “prepare beef”; prints “Taco Stand prepares: beef taco”
 - c. Command “rush lunch”; prints “Taco Stand handles rush: lunch”
7. Use TDD principles to guide your development efforts. Write your tests first, then write the code to make the tests pass, then refactor (Red – Green – Refactor).

Grading:

The total score is 60 points, with 24 points for the TacoStandEmitter class (8 points per method), 24 points for the tests (8 points per test), and 12 points for the CLI program.

Chapter 5. Process and OS

Chapter Overview

In this chapter, we will be exploring how Node.js applications can interact with the underlying operating system through its Process and Operating System (OS) modules. We will start by providing a brief introduction to Processes and their Events in Node.js. Following that, we will introduce the OS module, which offers various utility methods and properties for operating system – related tasks, allowing you to interact with the hardware and software in your computer. Additionally, we will look at the Event Loop and the Process.nextTick operation. By the end of this chapter, you will have a better understanding of how Node.js interfaces with an operating system, providing you with the necessary tools to build more resilient and adaptable Node.js programs.

Learning Objectives

By the end of this chapter, you should be able to:

- Define processes in a Node.js environment.
- Demonstrate the use of process.cwd() and process.pid().
- Identify the differences between process.nextTick() and setImmediate().
- Write code that gathers information from an Operating System.

Introduction to Processes in Node.js

Processes in Node.js are global objects that provide information on the current processes running in a Node.js environment. With processes, you can glean information about current processes, manipulate active processes, and use them to produce varying results in a Node.js program. As global objects, Node.js processes are always available from

anywhere in a Node.js program. They do not require importing or setting up, they are simply called using the process object variable.

The process object is incredibly useful in most Node.js programs. It provides functionality in areas like:

1. **Process Information:** Through the process object, you can obtain information about the current process, including process ID, current working directory, Node.js version, OS version, and more.
2. **Environment Variables:** process.env is a global property that allows you to set and access environment variables. This is especially useful for configuration settings like database username and passwords, safe IP addresses, network settings, and user provisions.
3. **Command Line Arguments:** The process.argv property allows you to access command-line arguments from a Node.js script. This is especially useful when building CLI scripts for test automation (TDD), CI/CD pipelines, and deployment scripts.
4. **Exit Status:** You can use the process object to set exit codes to control the execution behavior of a script. process.exit(), which was explored in a previous chapter, is one way you can end a process immediately. For example, you can use process.exit(0) to end the execution of a successful script and process.exit(1) for failures. By doing so, you can programmatically control and respond to the execution of a Node.js script.
5. **Standard Input/Output:** process.stdin, process.stdout, and process.stderr provide access to the standard input, output, and error streams, which is used heavily in console applications and when working with streams (covered in a later chapter).
6. **Event Handling:** The process object emits events, which means you interact with these events in the same way that you used the EventEmitter to build event-driven Node.js programs.

Below are some of the most commonly used properties of the process object:

Process ID:

```
// print the process Id  
console.log(process.pid);
```

Process Version:

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

Process Platform:

```
// print the process platform  
console.log(process.platform);
```

Process Title:

```
// print the process title  
console.log(process.title);
```

Process Arguments:

```
// print the process arguments  
console.log(process.argv);
```

Process Environment Variables:

```
// set a process environment variable  
process.env.HELLO = "Hello World";  
console.log(process.env.HELLO);
```

Current Working Directory:

```
// print the current working directory  
console.log(process.cwd());
```

Memory Usage:

```
// print the memory usage  
console.log(process.memoryUsage());
```

Uptime:

```
// print the uptime  
console.log(process.uptime());
```

Executable Path:

```
// print the execPath  
console.log(process.execPath);
```

Let's take a look at a program that uses some of these properties in a real-world scenario.

Process Environment Variables:

```
// process-env.js  
// Set the environment  
process.env.NODE_ENV = process.env.NODE_ENV || "development";  
  
// Display a message based on the environment  
if (process.env.NODE_ENV === "development") {  
  console.log("Running in development mode");  
} else if (process.env.NODE_ENV === "production") {
```

```

    console.log("Running in production mode");
  } else {
    console.log(`Running in ${process.env.NODE_ENV} mode`);
  }
}

```

In this code example, the `process.env` property is used to set the current environment to “development” if one is not specified when the script is executed. To execute this script, run the following command:

node process-env.js

The output will be “Running in development mode”. If you want to set the environment to “production” we can run the script like this:

NODE_ENV=production node process-env.js

The output will be “Running in production mode”. This is a common pattern in Node.js applications, where different settings or behaviors are used, depending on the environment the application is executed in (production, qa, test, development). You can also use environment variables for storing username and passwords for a database connection string or for keys to a payment portal like Stripe. For example,

```

// process-env2.js
// Set the database username, password, and server as environment variables
process.env.DB_USERNAME || "admin";
process.env.DB_PASSWORD || "s3cret";
process.env.DB_SERVER || "devMongoServer";

// Display the database username, password, and server
console.log(`Database username: ${process.env.DB_USERNAME}`);
console.log(`Database password: ${process.env.DB_PASSWORD}`);
console.log(`Database server: ${process.env.DB_SERVER}`);

```

In this example, three environment variables are created to represent the credentials for a database server. To run this script, enter the following command:

node process-env2.js

The output for this command should resemble the following:

```

Database username: admin
Database password: s3cret
Database server: dbMongoServer

```


You can also set your own values for each environment variable by using the following command:

```
DB_USERNAME=prodadmin DB_PASSWORD=supersecret  
DB_SERVER=prodMongoServer node process-env2.js
```

The output for this updated command should resemble the following:

```
Database username: prodadmin  
Database password: supersecret  
Database server: prodMongoServer
```

As you can see from the examples, using the `process.env` property is a powerful and flexible way for setting up environment variables in a Node.js application. Another `process` property that is used heavily in Node.js programs is `process.execPath`. This property is used to return a string that represents the absolute pathname of the executable that started the Node.js process.

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

This code will print the absolute path of the Node.js executable to the console. This is especially useful when you are working with child processes (covered in a later chapter) in a Node.js program. Another useful property is `process.platform`, which is used to determine the operating system on which the Node.js process is running. This can be useful in scripts that need to perform certain actions based on the underlying operating system. For example, the format for a file path in Windows is different than the format for a file path in Linux. You could use the `process.platform` property to determine the underlying operating system and write code that formats the path string accordingly.

```
// process-platform.js  
"use strict";
```

```
let pathToConfig; // path to the configuration file
```

```
if (process.platform === 'win32') {  
  // Windows uses backslashes to separate path segments  
  pathToConfig = 'C:\\Users\\User\\AppData\\Local\\MyApp\\config.json';  
} else {  
  // macOS and Linux use forward slashes  
  pathToConfig = '/usr/local/etc/myapp/config.json';  
}
```

```
console.log(` Path to configuration file: ${pathToConfig}`);
```

Depending on your OS, you will either see the path for a Windows or Linux computer (back slashes versus forward slashes). This is a simple example, but there are numerous situations where the `process.platform` property is extremely useful. The last property we will take a look at is `process.memoryUsage`. Imagine you are writing a program that generates a large number of random numbers and stores them in an array. You could use `process.memoryUsage` to keep track of how much memory is being used as the array grows.

```
// process-memory.js
let numbers = [];

for (let i = 0; i < 1000000; i++) {
  numbers.push(Math.random());

  // Every 100,000 numbers, log the current memory usage
  if (i % 100000 === 0) {
    const { heapUsed } = process.memoryUsage();
    console.log(`Memory usage after ${i} numbers: ${heapUsed / 1024 / 1024} MB`);
  }
}
```

The script will print how much memory is used after x numbers, using the following format: “Memory usage after X numbers: {MemoryUsage}”. Tracking the memory usage of a program is important for the following reasons:

1. **Performance:** Higher memory usage means slower processing time and potential crashes. By monitoring the memory usage in an application, you can identify the parts of your application that are consuming large amounts of memory (performing slow) and respond to them accordingly. Knowing how your application is performing is one of the main ways you can improve performance and response times.
2. **Resource Management:** In a production environment, resources are limited. This is especially true when you are working on a limited budget, start-up company, or small development team. Knowing how much memory your application is using provides insight into forecasting server and resource costs.
3. **Debugging:** If your application has a memory leak and is crashing for unknown reasons, you can use the data from the memory usage property to determine the cause of the issue.

As a new developer it is likely you will not deal with application-level memory issues, but learning about them now will set the foundation for more advanced programming concepts later in your career. I have worked with a lot of experienced and highly accomplished developers who still did understand the importance of checking the memory utilization of

an application. Hopefully, it goes without saying, but you should always strive to write more performant code.

In the next section we will take a closer look at process events and how to work with them in a Node.js program.

Understanding Process Events

The process object returns an instance of the EventEmitter class, which means events can be emitted and listeners can be created to listen for the emitted events. There are several events that can be emitted by the process object. These include:

1. **exit:** In Node.js, the exit event is triggered when the process is about to exit due to one of two reasons: either the process.exit() method was called somewhere in the program, or the event loop in Node.js has no more tasks to perform, which causes the exit event to be emitted.
2. **uncaughtException:** This event is emitted when an uncaught JavaScript error occurs in the event loop. A listener can be created to listen for this event. In doing so, the default behavior (throwing an error with a stack trace) is ignored. The advantage of this approach is, you can decide how the error is handled in your application.
3. **unhandledRejection:** This event is emitted whenever a promise is rejected, but there was no error handling in place to handle the promise rejection.
4. **warning:** This event is similar to the error event; it is emitted whenever Node.js has a warning message that needs to be emitted.
5. **message:** This event is emitted when process.send() is called.
6. **beforeExit:** This event is emitted when the Node.js event loop is empty and there are no additional tasks scheduled. However, if you register a listener for beforeExit and initiate an asynchronous operation, Node.js will wait for the operation to either be fulfilled or rejected before the process exits. The advantage of this approach is, you can schedule cleanup tasks or operational tasks before exiting the process and event loop. To understand the significance of this event, let's consider a simple text-based video game running in Node.js. Before a player decides to exit the game, we want to make sure that their process is saved. We can use the beforeExit event to satisfy this requirement (see the example later in this chapter).
7. **multipleResolves:** This event is emitted when a Promise has resolved multiple times. This can occur when a promise is rejected then fulfilled, when a promise is

fulfilled then rejected, when a promise is rejected multiple times, or when a promise is fulfilled multiple times.

8. **rejectionHandled**: This event is emitted when a Promise is rejected, but error handling was in place to handle the promise rejection. This event is the opposite of the `unhandledRejection` event.
9. **Signal Events (SIGINT, SIGTERM)**: Signal events are emitted when the Node.js process receives a signal. The two mostly commonly emitted events are SIGINT and SIGTERM.

Let's take a look at a couple Node.js programs that leverage the most commonly used process events.

Program 1: Imagine we are building a program where the user is a gardener and they are tending to their garden. When the gardener is ready to return to the house (the program is about to exit), we want to print a message to the console. Here is how we could use the `exit` event to satisfy this requirement:

```
// file name: gardener-returns-home.js
"use strict";

let gardener = {
  name: "Art Moze",
  location: "Rose Garden"
};

function returnToHouse() {
  console.log(` ${gardener.name} has finished tending to the ${gardener.location}. `);
  process.exit();
}

process.on("exit", () => {
  console.log(` ${gardener.name} has returned to the house. Good job, Art! `);
});

setTimeout(returnToHouse, 2000);
```

In this code example, an object literal named `gardener` is created with two properties: `name` and `location`. A function named `returnToHouse` is created that prints a message to the console window and calls the `process.exit()` function to exit the program. Next, we register a listener for the emitted `exit` event and print a message to the console. A call to the `setTimeout()` function is used to simulate the time it the gardener to tend to their garden. Running this script should print:

**Art Moze has finished tending to the Rose Garden.
Art Moze has returned to the house. Good job, Art!**

There is a 2 second delay in the output, which is controlled by the `setTimeout()` function.

Program 2: For this program, we will reuse the code from the Gardener program to illustrate how to use the `uncaughtException` event.

```
"use strict";

// gardener-uncaught-exception.js
"use strict";

let gardener = {
  name: "Art Moze",
  location: "Rose Garden",
};

function returnToHouse() {
  console.log(
    `${gardener.name} has finished tending to the ${gardener.location}.`
  );
  nonexistentFunction();
}

process.on("uncaughtException", (err) => {
  console.log(`An error occurred: ${err.message}`);
  console.log(`${gardener.name} will return to the house to recover`);
  process.exit(1);
});

setTimeout(returnToHouse, 2000);
```

The code in this program is similar to the previous one, exception instead of calling `process.exit` from the `returnToHouse` function, we are calling a non-existent function. The purpose of this call is to introduce a JavaScript error, so you can see how the `uncaughtException` event works. Next, we register an event listener for the `uncaughtException` event, print two messages to the console window, and call `process.exit` with a non-zero status code to indicate an error has occurred in our program. Running this script from the CLI should print:

**Art Moze has finished tending to the Rose Garden.
An error occurred: nonexistentFunction is not defined
Art Moze will return to the house to recover**

Program 3: Let's modify our Gardener scenario a little bit to demonstrate how to use the `beforeExit` event. We will use the following workflow to guide the development of this program:

1. Art Moze finishes tending to the Rose Garden.
2. Art Moze returns to the house.
3. Art Moze is cleaning his tools.
4. Art Moze is going to rest.

```
"use strict";
```

```
// gardener-before-exit.js
```

```
"use strict";
```

```
let gardener = {  
  name: "Art Moze",  
  location: "Rose Garden"  
};
```

```
function returnToHouse() {  
  console.log(` ${gardener.name} has finished tending to the ${gardener.location}.` );  
  setTimeout(() => {  
    console.log(` ${gardener.name} returns to the house.` );  
    cleanTools();  
  }, 4000);  
}
```

```
function cleanTools() {  
  console.log(` ${gardener.name} is cleaning his tools.` );  
}
```

```
process.on("beforeExit", () => {  
  console.log(` ${gardener.name} is going to rest.` );  
})
```

```
setTimeout(returnToHouse, 2000);
```

The `beforeExit` event is difficult to simulate, because it is only emitted when the Node.js event loop has no additional tasks to complete. But, in this code example, we simulate it through calling the `setTimeout` function and then clearing the event loop. The logical order of events in this program are as follows:

1. The script starts and schedules the `returnToHouse` function to be executed after 2 seconds using the `setTimeout` function.
2. After a 2 second delay, the `returnToHouse` function is executed. The function logs a message to the console window and schedules a function to be executed in 4 seconds using a second `setTimeout` function.
3. After a 4 second delay (a total of 6 seconds from when the script originally started), the scheduled function inside the `returnToHouse` function is executed. This function logs a message to the console and calls the `cleanTools` function.
4. The `cleanTools` function is executed immediately after being called, which logs a message to the console window.
5. After the `cleanTools` function finishes executing, the Node.js event loop becomes empty, because there are no more scheduled tasks to complete. This causes the `beforeExit` event to be emitted.
6. When the `beforeExit` event is emitted, the registered handler is executed, which prints a message to the console window.

Of all the process events listed, `beforeExit` is one of the most difficult to understand. But it is equally one of the most useful events to use in a Node.js program. Below are a few scenarios where the `beforeExit` comes in handy.

1. **Cleanup operations:** If your application creates temporary files or holds system resources, you can use the `beforeExit` event to release the held resources and to delete the temporary files.
2. **Logging and analytics:** If your application uses logging, you can use the `beforeExit` event to log information about the applications execution and/or analytics such as, how long a task ran, what the task did, and when the task exited. This information is extremely helpful for debugging and gathering analytics about how your application is being used by its customer base.
3. **Sending notifications:** If your application uses background tasks or long running tasks, you can use the `beforeExit` event to send notifications (email or SMS) to notify users that the tasks have completed.
4. **Saving application state:** if your application maintains state, you can use the `beforeExit` event to save the progress of a user in the application. This is helpful, because the user can continue where they left off. Imagine working on a document and each time you exited the program you had to start over from scratch.

In the next section we will take a closer look at the OS module and how it can be used to access OS level resources.

Introduction to the OS Module

The OS module is a built-in Node.js module that provides utility methods for interacting with a computers operating system. Think of the OS module as a toolbox for working with a computers operating system. Just like a tool bag is used for working on a houses roof. The main functions of the OS module are:

1. **OS information:** You can use the OS module to obtain information about the computers operating system where the Node.js script is being executed. This includes, the OS type, platform, version etc., Similar to the process object, you can check if the computer is running Windows, macOS, or Linux.
2. **Check system resources:** You can use the OS module to learn about a computers resource. For example, how much memory is free and how much memory is being used. Or, how many CPUs the computer has. This information is useful for a variety of reasons, like: performance monitoring (discussed in the process section), whether the application is running on multiple servers (load balancing), to prevent crashes, to determine if the system meets requirements to run your application, and to troubleshoot errors/unexpected behaviors.
3. **Manage file paths:** In “Introduction to Processes” we talked about how certain operating systems manage file paths differently. The OS module provides utility methods for working with file paths that are specific to the operating system in which the application is hosted. This makes working with file paths trivial.
4. **Get user information:** You can use the OS module to obtain information about the current user of the computer where your script is executed. This includes the user’s home directory, username, user-group ID, the users shell program, and user identifier.

Let’s take a look at some Node.js programs that use the OS module to interact with the computers operating system.

OS information:

```
console.group(`\nThis computer's OS information is: `);
console.log(` This computer's host name is ${os.hostname()}. `);
console.log(` This computer's operating system is ${os.type()}. `);
console.log(` This computer's operating system platform is ${os.platform()}. `);
console.log(` This computer's operating system release is ${os.release()}. `);
console.log(` This computer's operating system architecture is ${os.arch()}. `);
console.log(` This computer's operating system version is ${os.version()}. `);
```



```
console.groupEnd();
```

System resources:

```
console.group(`\nThis computer's system resources are:`);  
console.log(` This computer has ${os.cpus().length} CPUs.`);  
console.log(` This computer has ${os.totalmem()} total memory.`);  
console.log(` This computer has ${os.freemem()} free memory.`);  
console.log(` This computer has been up for ${os.uptime()} seconds.`);  
console.groupEnd();
```

User information:

```
console.group(`\nThis computer's user information is:`);  
console.log(` This computer's user info is:`);  
console.log(os.userInfo());  
console.log(` This computer's user ID is ${os.userInfo().uid}.`);  
console.log(` This computer's group ID is ${os.userInfo().gid}.`);  
console.log(` This computer's user name is ${os.userInfo().username}.`);  
console.log(` This computer's home directory is ${os.userInfo().homedir}.`);  
console.log(` This computer's shell is ${os.userInfo().shell}.`);  
console.groupEnd();
```

Manage file paths:

```
const os = require("os");  
const path = require("path");  
  
// Get the user's home directory  
const homeDir = os.homedir();  
  
// Define a new file in the user's home directory  
let file = "plants.txt";  
  
// Use the path module to join the home directory and the file name  
let filePath = path.join(homeDir, file);  
  
console.log(` The file path is ${filePath}.`);
```

In this code example, the path module is used to join the file name plants.txt to the home directory, which was obtained by calling os.homedir().

Let's take a look at a simple program that uses the os module to get user info. Imagine you are building a desktop application and you want to save user specific data or settings (think VS Code with multiple user accounts on a computer). You can use the os module to obtain the username and their home directory to save the settings file.

```
// os-user-config.js
const os = require("os");
const path = require("path");

const userInfo = os.userInfo();
const homeDir = userInfo.homedir;

let settingsFile = `${userInfo.username}_settings.json`;

let settingsFilePath = path.join(homeDir, settingsFile);

console.log(` Hello, ${userInfo.username}! Your settings file will be stored in this file:
${settingsFilePath}.` );
```

In this example, we are simulating the saving of a user setting file in the active user's home directory. This way, each user on the system can have their own settings file in their home directory. The file name is {username}_settings.json. Where {username} is the actual username of the user signed into the system. For example, smith_settings.json.

In the next section we will take a look at the Node.js event loop and process.nextTick.

Event Loop and Process.nextTick

JavaScript is a single-threaded programming language, where operations are processed one at a time. This design choice was largely influenced by the need to design a programming language that avoided concurrent execution issues, like race conditions. Its simplicity makes it extremely useful for simple tasks, like user interactions in a web browser. Tasks in a web browser are typically short, fast, and executed “one at a time.” Think about how a user clicks on a button in a webpage, uses their mouse to scroll over a <div>, or submits a registration form.

A thread in computer science is the smallest sequence of programmable instructions that are executed independently by some type of schedule. In a single-threaded environment, like JavaScript, if two operations are needed to perform some type of task, they cannot run at the same time. Regardless, if both operations are ready, they still have to be expected one at a time. JavaScript uses asynchronous callbacks and the event loop to handle long running tasks or tasks that are memory intensive, like reading data from a file, querying a database, awaiting a response from an API call, etc.,

These tasks are given to the system to be executed and when they are complete, their callbacks are added to what is called an event queue. The JavaScript runtime, takes the tasks from the event queue (one at a time), which gives the illusion of multitasking behavior, despite JavaScript still only being a single-threaded environment.

The event queue and event loop are two very important components of the JavaScript runtime environment, as they allow us to execute asynchronous operations. The event queue is a data structure (you can think of an array if it's easier) that holds the callbacks from asynchronous operations that are ready to be executed. While this is not entirely accurate, you can think of the event queue like an array of functions waiting to be called in your JavaScript program. Once an asynchronous operation completes (timer, network request, database query, etc.,) its callback function is added to the event queue.

The event loop is a continuous process that checks the event queue for callbacks that are ready to be executed. If there is a callback that is ready to be executed and the call stack is empty, the event loop takes the callback out of the event queue and adds it to the call stack, so it can be executed.

Think of this process as two lanes (call stack and event queue) in a highway with a police officer (event loop) directing traffic. This process allows JavaScript, which is single-threaded, to manage many asynchronous operations. While the JavaScript thread is executing code, the event loop is continually adding and removing callbacks from the event queue. Then, once the thread is free, the runtime can start executing the callbacks from the queue.

Let's use a restaurant example to explain this process. Think of JavaScript as a Taco Stand Manager. The manager can only perform one task at a time (single-threaded), but he does it so quickly that it appears to everyone watching that he is doing multiple tasks at once.

The tasks that the Manager needs to do are like function calls in JavaScript. When an order comes in (prepare a taco), the Manager handles it immediately, if he is not busy. If he is busy, the order gets added to the ticket holder, which is like the event queue. Now, imagine the Taco Stand is in the middle of their lunch rush, receiving multiple orders at a time (two order takers). The Manager cannot handle all of the orders by himself, instead, his employees help with the orders. The employees are like system operations in JavaScript that handle tasks like reading data from a file or awaiting a response from an API call.

When an employee is done preparing an order, they cannot interrupt the Manager to tell him the order is complete (he is too busy to notice). They mark the order ticket as complete (callback) and add it to the receipt spindle. The Manager checks the receipt spindle whenever his current order is completed to know which order to work on next. This is how JavaScript can handle many tasks efficiently, even though it can only do one thing at a time. The event loop in JavaScript is like the Manager constantly checking the receipt spindle to determine the next order to prepare.

"use strict";

// taco-stand-simulation.js

```

console.log("The Taco Stand Manager starts his day.");

// An order comes in to prepare a taco
setTimeout(() => {
  console.log(
    "An employee prepared a taco. The order ticket is marked as complete and added to
the receipt spindle."
);
}, 3000);

// Another order comes in to prepare a burrito
setTimeout(() => {
  console.log(
    "An employee prepared a burrito. The order ticket is marked as complete and added
to the receipt spindle."
);
}, 2000);

// Yet another order comes in to prepare a quesadilla
setTimeout(() => {
  console.log(
    "An employee prepared a quesadilla. The order ticket is marked as complete and
added to the receipt spindle."
);
}, 1000);

console.log(
  "The Manager is busy with the current order. He will check the receipt spindle when
he's done."
);

```

This program is simple, but it does show how the event queue and event loop work in JavaScript. The code is executed immediately, that is the main thread of the program has finished its execution, but the timers set by the `setTimeout` calls are still counting down in the background. When each timer expires, its callback function (code inside of the `setTimeout` function) is added to the event queue. The event loop is constantly checking the event queue and the call stack.

1. The third `setTimeout` (quesadilla) has a timer of one second. Its callback function is added to the event queue first and therefore is the first operation to be executed by the event loop.
2. The second `setTimeout` (burrito) is added to the event queue next and executed next by the event loop.

3. Finally, the first `setTimeout` (taco) is added to the event queue next and executed by the event loop.

In Node.js, when you want to delay the execution of an operation until the next pass of the event loop, you can use the function `process.nextTick()`. This function is quite similar to `setTimeout` and `setImmediate`, but with one key difference: it will execute before any I/O events are fired, whereas `setTimeout` and `setImmediate` will allow I/O events to run first.

This difference is significant because:

1. **Efficiency:** `process.nextTick` allows you to handle asynchronous operations in a more efficient manner, because callback functions are executed sooner.
2. **Recursion:** `process.nextTick` is a great tool to use when you are working with recursive functions, which are functions that call themselves until some condition is met. There are two types of recursions, tail recursion and non-tail recursion. In non-tail recursion, the recursive call is made at the beginning of a function and in tail recursion, the recursive call is made at the end of the function. `process.nextTick` can handle recursive function calls without blocking the I/O.
3. **Order of operations:** `process.nextTick` can be used to control the order in which functions are executed in your code base. This flexibility is great, especially when you are working with asynchronous code and you want to explicitly state in the order in which some operations are executed.

Let's take a look at an updated example of our Taco Stand JavaScript program that uses `process.nextTick` to immediately execute an asynchronous operation.

// Filename: taco-stand-next-tick.js

```
"use strict";
```

```
console.log("The Taco Stand Manager starts his day.");
```

```
function tacoOrder() {  
  setTimeout(() => {  
    console.log("An employee prepared a taco. The order ticket is marked as complete  
and added to the receipt spindle.");  
  }, 2000);  
}
```

```
function burritoOrder() {  
  process.nextTick(() => {
```

```

    console.log("An employee prepared a burrito. The order ticket is marked as
complete and added to the receipt spindle.");
  });
}

function quesadillaOrder() {
  setTimeout(() => {
    console.log("An employee prepared a quesadilla. The order ticket is marked as
complete and added to the receipt spindle.");
  }, 1000);
}

console.log("The Manager is busy with the current order. He will check the receipt
spindle when he's done.");

tacoOrder();
burritoOrder();
quesadillaOrder();

```

In this code example, we are using the `process.nextTick` function to simulate the preparation of a burrito order. Even though it is the second function called it is still the first to be executed. This is possible because the `process.nextTick` function adds the task to the event queue as soon as the current operation completes. The other functions (`tacoOrder` and `quesadillaOrder`) are controlled by the timers set in the `setTimeout` functions. The output after running this script should resemble the following:

The Taco Stand Manager starts his day.

The Manager is busy with the current order. He will check the receipt spindle when he's done.

An employee prepared a burrito. The order ticket is marked as complete and added to the receipt spindle.

An employee prepared a quesadilla. The order ticket is marked as complete and added to the receipt spindle.

An employee prepared a taco. The order ticket is marked as complete and added to the receipt spindle.

As you can see from the output, even though I called `tacoOrder` first, then `burritoOrder`, and then the `quesadillaOrder`, the output was different. The `process.nextTick` function (`burritoOrder`) printed first, then the `quesadillaOrder` function printed (timer was set to 1 second), and finally the `tacoOrder` function printed (timer was set to 2 seconds).

In the final section of this chapter, we will take a look at how to use Jest and TDD principles to test process events.

TDD with Process Events

Our first TDD example will use the `process.nextTick` to show how it could be rewritten using TDD principles and Jest.

Project Structure:

```
gardener-app
  src
  tasks.js
  test
  tasks.spec.js
  package.json
  jsconfig.json
```

```
// jsconfig.json
{
  "compilerOptions": {
    "types": ["jest"]
  }
}
```

The `jsconfig.json` file is a configuration file that we will use to enable Jest syntax highlighting and intellisense recommendations. Without this file, VS Code will not recognize the Jest commands you use in the unit tests.

```
// package.json
{
  "name": "gardener-app",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "jest"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "@types/jest": "^29.5.11",
    "jest": "^29.7.0"
  }
}
```

1. Write failing tests

```
// test/tasks.spec.js
"use strict";

const tasks = require("../src/tasks");

test("Gardener waters the plants", done => {
  const log = jest.spyOn(console, 'log');
  tasks.waterPlants();
  process.nextTick(() => {
    expect(log).toHaveBeenCalledWith('Gardener: The plants have been
watered.');
```

```
    log.mockRestore();
    done();
  });
});

test("Gardener prunes the trees", done => {
  const log = jest.spyOn(console, 'log');
  tasks.pruneTrees();
  setTimeout(() => {
    expect(log).toHaveBeenCalledWith("Gardener: The trees have been pruned.");
    log.mockRestore();
    done();
  }, 1000);
});

test("Gardener mows the lawn", done => {
  const log = jest.spyOn(console, 'log');
  tasks.mowLawn();
  setTimeout(() => {
    expect(log).toHaveBeenCalledWith("Gardener: The lawn has been mowed.");
    log.mockRestore();
    done();
  }, 2000);
});
```

2. Run the test and see it fail. To do this, run the command: **npm test** from the directory where the project is located.
3. Implement the code to make the tests pass:

```
// src/tasks.js
"use strict";
```



```

function mowLawn() {
  setTimeout(() => {
    console.log("Gardener: The lawn has been mowed.");
  }, 2000);
}

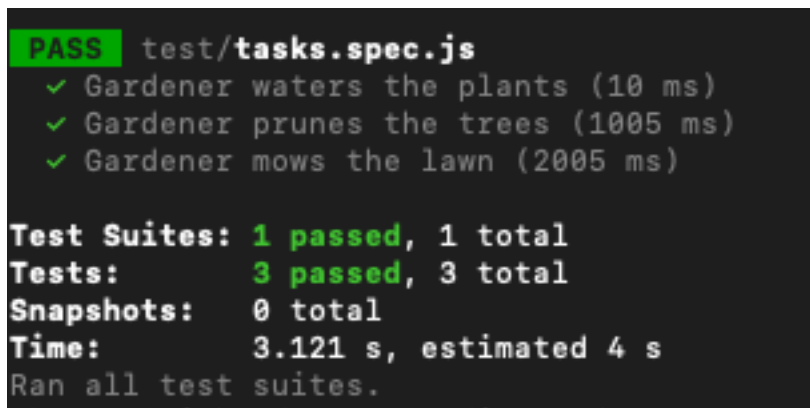
function waterPlants() {
  process.nextTick(() => {
    console.log("Gardener: The plants have been watered.");
  });
}

function pruneTrees() {
  setTimeout(() => {
    console.log("Gardener: The trees have been pruned.");
  }, 1000);
}

module.exports = { mowLawn, waterPlants, pruneTrees };

```

4. Run the tests to see them pass. To do this, run the command **npm test** from the directory where the project is located.



```

PASS test/tasks.spec.js
  ✓ Gardener waters the plants (10 ms)
  ✓ Gardener prunes the trees (1005 ms)
  ✓ Gardener mows the lawn (2005 ms)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        3.121 s, estimated 4 s
Ran all test suites.

```

This next program is designed to display a message about a botanist's favorite plant. The plant's name will be determined by an environment variable we set: `FAVORITE_PLANT`. By default, we will set the plants name to "Sunflower." If an environment variable is set, we will print that name instead of "Sunflower." The output will be, "The botanist's favorite plant is {Plant}!" The development of this program follows TDD principles. There will be a total of two test cases, one that checks if the environment variable is used correctly and another that checks the default value of the environment variable.

Project Structure:

botanist-app

src

botanist.js

test

botanist.spec.js

package.json

// jsonconfig.json

```
{
  "compilerOptions": {
    "types": ["jest"]
  }
}
```

// package.json

```
{
  "name": "botanist-app",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "jest"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "@types/jest": "^29.5.11",
    "jest": "^29.7.0"
  }
}
```

1. Write the failing unit tests first:

// test/botanist.spec.js

"use strict";

const botanist = require("../src/botanist");

describe("favoritePlant", () => {

let log;

beforeEach(() => {

```

    log = jest.spyOn(console, 'log');
  });

  afterEach(() => {
    log.mockRestore();
  });

  test("uses favorite plant from environment variable", () => {
    process.env.FAVORITE_PLANT = "Rose";
    botanist.favoritePlant();
    expect(log).toHaveBeenCalledWith("The botanist's favorite plant is Rose!");
  });

  test("uses default favorite plant if environment variable is not set", () => {
    delete process.env.FAVORITE_PLANT;
    botanist.favoritePlant();
    expect(log).toHaveBeenCalledWith("The botanist's favorite plant is Sunflower!");
  });
});

```

The following is a brief explanation of what each operation in the test file is doing:

- a. **describe()**: This is how you set up a test suite in Jest. A test suite is a collection of related unit tests. To understand this concept, consider the following, imagine you are building an application that allows users to manage their playlist. Typical behavior would consist of, adding new songs to the playlist, updating the playlist, and removing songs from the playlist. In a testing environment, you might have

```

describe("Playlist operations", () => {
  describe("addSongToPlaylist", () => {
    test("adds a song to a playlist", () => {
      // Your test goes here
    });

    test("should not add the song if it already exists in the playlist", () => {
      // Your test goes here
    });
  });

  describe("removeSongFromPlaylist", () => {
    test("removes a song from a playlist", () => {

```

```

});

test("should not remove the song if it does not exist in the playlist", () => {

});

});

describe("shufflePlaylist", () => {
  test("shuffles the playlist", () => {

  });

  test("should not shuffle the playlist if it has only one song", () => {

  });
});
});
});

```

- b. **beforeEach():** This is called a hook. The beforeEach hook runs before each test in the describe block is called. You use this hook to setup conditions that should be applied to unit tests before they run. A typical example is mocking data.
 - c. **log = jest.spyON(console, "log"):** This line of code is replacing the default console.log with a Jest mock function and assigns it to log. This is needed so Jest can check if the console.log in your module is being called correctly.
 - d. **afterEach():** This is also a hook. The afterEach hook runs after each test in the describe block is called. You can use this hook to release resources or clean up testing conditions that were added during the test execution.
 - e. **log.mockRestore():** This is a function that restores the console.log to its original state. We are using this to make sure the default behavior of console.log is restored after each unit test is executed. Otherwise, we run the risk of other tests in our test suite to be impacted by the mock we created in the beforeEach hook.
2. Run the test command (**npm test**) in the terminal window where the project saved to confirm they are failing.

```

Test Suites: 1 failed, 1 total
Tests:      2 failed, 2 total
Snapshots:  0 total
Time:       0.108 s, estimated 1 s
Ran all test suites.

```

3. Write the code to pass the failing unit tests.

```

// src/botanist.js
"use strict";

function favoritePlant() {
  const plant = process.env.FAVORITE_PLANT || "Sunflower";
  console.log(` The botanist's favorite plant is ${plant}!` );
}

module.exports = { favoritePlant };

```

4. Rerun the test command (**npm test**) to ensure they are now passing.

```

PASS test/botanist.spec.js
  favoritePlant
    ✓ uses favorite plant from environment variable (8 ms)
    ✓ uses default favorite plant if environment variable is not set (1 ms)

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        0.157 s, estimated 1 s
Ran all test suites.

```

5. Refactor, if necessary.

In this final example, we will take a look at how to use TDD principles to test the `process.exit` event. Consider the following program:

Write a program that checks if a plant is a flower. The program should take the plant's name as an input. If the plant is a flower, the program should print a message saying, "[Plant] is a flower!" If the plant is not a flower, the program should print a message saying, "[Plant] is not a flower!" and then exit with a status code of 1. We will use the following steps to guide our development efforts:

1. Write a test that checks if the program correctly identifies a flower and prints the correct message.

2. Write a test that checks if the program correctly identifies a non-flower plant, prints the correct message, and exits with a status code of 1.
3. Write the code to make the tests pass.

Project Structure:

plant-checker

src

plant.js

test

plant.spec.js

package.json

jsonconfig.json

// jsonconfig.json

```
{  
  "compilerOptions": {  
    "types": ["jest"]  
  }  
}
```

// package.json

```
{  
  "name": "botanist-app",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "test": "jest"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "devDependencies": {  
    "@types/jest": "^29.5.11",  
    "jest": "^29.7.0"  
  }  
}
```

1. Failing unit tests:

"use strict";

```

const plant = require("../src/plant");

const exit = jest.spyOn(process, 'exit').mockImplementation((code) => code);

describe("isFlower", () => {
  let log;

  beforeEach(() => {
    log = jest.spyOn(console, "log");
  });

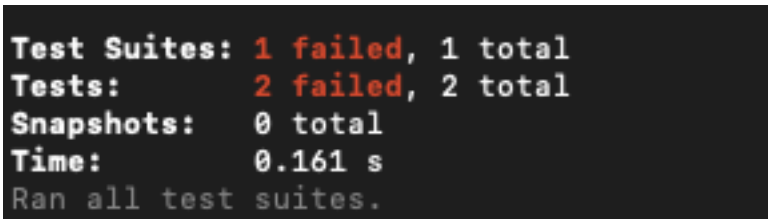
  afterEach(() => {
    log.mockRestore();
  });

  test("identifies a flower", () => {
    plant.isFlower("Rose");
    expect(log).toHaveBeenCalledWith("Rose is a flower!");
    expect(exit).not.toHaveBeenCalled();
  });

  test("identifies a non-flower plant", () => {
    plant.isFlower("Fern");
    expect(log).toHaveBeenCalledWith("Fern is not a flower!");
    expect(exit).toHaveBeenCalledWith(1);
  });
});

```

2. Run the test command (**npm test**) in the terminal window where the project saved to confirm they are failing.



```

Test Suites: 1 failed, 1 total
Tests:       2 failed, 2 total
Snapshots:   0 total
Time:        0.161 s
Ran all test suites.

```

3. Implement the code so the tests pass:

```

"use strict";

function isFlower(name) {
  const flowers = [

```

```

    "Rose",
    "Tulip",
    "Orchid",
    "Sunflower",
    "Daisy",
    "Lily",
    "Daffodil",
    "Geranium",
    "Iris",
    "Violet",
    "Jasmine",
    "Lavender",
    "Poppy",
    "Pansy",
    "Peony",
    "Hyacinth",
    "Marigold",
    "Petunia",
    "Carnation",
    "Chrysanthemum",
    "Aster",
    "Begonia",
    "Dahlia",
    "Zinnia",
    "Snapdragon"
  ];

  if (flowers.includes(name)) {
    console.log(` ${name} is a flower!` );
  } else {
    console.log(` ${name} is not a flower!` );
    process.exit(1);
  }
}

module.exports = { isFlower };

```

4. Rerun the test command (npm test) to confirm the unit tests are now passing.


```
PASS test/plant.spec.js
  isFlower
    ✓ identifies a flower (9 ms)
    ✓ identifies a non-flower plant (1 ms)

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        0.172 s, estimated 1 s
Ran all test suites.
```

5. Refactor, if necessary.

Programming Exercises

In this assignment, you will create a module that simulates a simple pie baker. You will use TDD principles and Jest for unit testing.

Instructions

1. Create a new folder for your project. Name it pie-baker
2. Inside the pie-baker folder create two subfolders: src and test
3. Inside the test folder, create a file named pie.spec.js
4. In the pie.js file, you will write a function named bakePie that takes a type of pie and an array of ingredients. The function should return a message indicating whether the pie was successfully baked or not. If an essential ingredient is missing, the function should log a warning message and call `process.exit(1)`.
 - a. The essential ingredients are: flour, sugar, and butter.
5. In the pie.spec.js file, you will write unit tests for the bakePie function. You should write at least three tests.
6. In your package.json file, add a test script that runs jest.
7. Use TDD principles to guide your development process. Write your tests first, then write the code to make the tests pass, then refactor (Red – Green – Refactor).

Grading

You will earn 20 points for each passing unit test, for a total of 60 points. If a test does not pass, you will not receive points for that test.

Hints

- Remember to use `module.exports` to export your function from `pie.js` and require to import it into `pie.spec.js`
- Use `jest.fn()` to create mock functions in your tests. Follow the format used in the `plant-checker` program.
- Use `toBe` or `toEqual` for your assertions.
- Remember to call your functions in each test with the necessary arguments.
- Use `npm install --save-dev` to install `jest`.
- Use `npm test` to run your tests.
- If you are testing a function that calls `process.exit`, be aware that this will terminate the Node.js process immediately. This means, that any unit tests following a check to `process.exit` will not be executed and will fail. Structure your unit tests in a way that the test to check `process.exit` is the last one in the test suite list. For example, unit test 1 (condition that does not call `process.exit`), unit test 2 (condition that does not call `process.exit`), unit test 3 (condition that does call `process.exit`).

Chapter 6. HTTP

Chapter Overview

The HTTP module is a built-in module that enables communications over the Hyper Text Transfer Protocol (HTTP). It allows for the creation of an HTTP server and the exchange of transactions (request and responses) from other servers. In this chapter, we will take a look at Node.js's built-in HTTP module, including how to create a basic HTTP server with Node.js, how to respond to server requests and responses, how to create routes in an HTTP server, and how to render HTML documents in a Node.js HTTP server.

Learning Objectives

By the end of this chapter, you should be able to:

- List the differences between HTTP methods.
- Demonstrate how to create a basic HTTP server with Node.js
- Explain routing.
- Build unit tests in Node.js to send and receive data from an HTTP server.

Introduction to HTTP and HTTP Methods

HTTP stands for Hyper Text Transfer Protocol and it is the protocol used to transmit hypertext over the web. At its core, HTTP more or less is a messaging definition that specifies how data should be formatted and transmitted. It also defines how web servers and browsers should respond to transactions sent across its protocol. These “methods” are known as “server requests” or “request methods” that request some type of action to be performed by a resource on the server.

To differentiate between request types, the protocol uses HTTP methods. These methods are intended to provide a separation between the allowable operations that can be invoked or applied to a resource on a server. The most common HTTP methods are:

1. **GET:** The GET method is an HTTP request that represents retrieving data from a server. This request method is often used to retrieve information/resources from a server. It does not change the state of the server (adding or updating a resource). For example, returning a list of records from a database or returning a single record from a database by a filtered ID.
2. **POST:** The POST method is an HTTP request that represents sending data to a server. This request method is often used to add a new resource on the server (i.e., changing the servers state). For example, creating a new record in a database.
3. **PUT:** The PUT method is an HTTP request that replaces the current state of a resource with an updated version. This request method is often used when you want to update a resource on a server. For example, updating a record in a database.
4. **DELETE:** The DELETE method is an HTTP request that deletes a resource. This request method is often used when you want to delete a resource on the server. For example, deleting a record from a database.
5. **PATCH:** The PATCH method is similar to the PUT method, but instead of replacing the entire resource, the PATCH method is used to apply modifications to an existing resource. The request method is often used when you only want to update certain fields in a resource. For example, only updating the email address of a user record in a database.
6. **HEAD:** The HEAD method is identical to the GET method, except there is no response body.
7. **OPTIONS:** The OPTIONS method is used to specify the communication options for the request method being invoked. This method allows you to specify communication requirements for a request. For example, supported HTTP methods.

All of these methods can be used to differentiate between different types of behaviors in a web application.

Imagine HTTP as a communication system in a taco stand:

1. **GET:** This is like a customer looking at the menu to see what tacos are for sale. They are not changing anything in the menu, rather they are just looking over the menu to decide what type of taco they should order.
2. **POST:** This is like a customer placing an order for a taco. They are adding something new to the ticket holder (new order ticket).
3. **PUT:** This is like the taco stand updating their menu by replacing their cubed pork taco with green salsa, lettuce, and queso with a carnitas taco with cilantro. They are updating a resource (cubed pork taco) by replacing the existing one with an updated version of that resource (new carnitas taco).
4. **PATCH:** This is like a customer who has already ordered a carne asada taco with cilantro and onions, wanting to update their order to include queso fresco. They are not replacing the entire taco (switching to a different taco altogether), but rather modifying a part of their taco order.
5. **DELETE:** This is like a taco stand removing the steak taco from their menu because they ran out of steak meat. They are deleting a resource.
6. **OPTIONS:** This is like asking the taco stand what toppings they can add to their tacos. This is how the OPTIONS method works. You are finding out the capabilities of a resource (i.e., the tacos toppings).

In the next section we will take a look at how to create a basic HTTP server with Node.js

Creating a Basic HTTP Server with Node.js

To create an HTTP server in Node.js, you use the built-in http module. This module allows you to create, listen, and respond to HTTP requests. Here is an example of how you can create a basic HTTP server.

```
"use strict";

const http = require("http");

const server = http.createServer((req, res,) => {
  res.statusCode = 200;
```

```
res.setHeader("Content-Type", "text/plain");
res.end("Learning Node.js is fun!");
});

server.listen(3000, "localhost", () => {
  console.log("Server running at http://localhost:3000/");
});
```

In this code example, the http module is imported using a require statement. Next, a server object is created that is an instance of an HTTP server. This call has two formal parameters: req and res.

1. **req (Request):** This is an object that represents the information being sent to a resource on your server. Information detailed in this object includes: request headers, query parameters, payload, URL, etc.
 - a. **payload:** In the context of HTTP, the term “payload” is used to describe data that is transmitted and intended to be used by the server. Think of this like when you packed your lunch box for school. You put a drink in the lunch box, a sandwich, a bag of chips, a cookie, and maybe an apple. These are the main things you need to eat for lunch. So, they are the “payload” of your lunch box. The lunch box itself, the zipper, cartoon logo, and handle on the outside – these are all important parts of your lunch box, but they are not what you are going to eat for lunch. In the context of HTTP, these parts are like the headers and metadata – they are important for delivering the payload, but they are not what you interact with (food items you eat during lunch).
 - b. **query parameters:** Query parameters are part of the URL that provide additional information your server needs for the incoming request. You use this as a mechanism for sending data or instructions to the server. Imagine you are at school and you are ordering your lunch through a kiosk. You choose a sandwich, but you have some specific preferences. You want to add extra cheese, bacon, no tomatoes, no pickles, and you want the bread toasted. In this example, the sandwich is the main resource you are requesting, but the extra cheese, bacon, no tomatoes, no pickles, and toasted bread are additional things you are including to customize your request. In the context of HTTP and URLs with query parameters, the customizations (extra cheese, bacon, no tomatoes, no pickles, and toasted bun) are sent using query parameters. The URL might look like the following: `http://lunchapp.com/order?sandwich=standard&extra=cheese&no=tomatoes&no=pickles&toasted=true`. So, in query parameters, data is passed through the URL using a key/value pair format. And, each key-value pair is

separated by an ampersand (&). This is a common approach to including multiple query parameters in a single HTTP request.

2. **res (Response):** This is an object that represents the information you send back to the client, which includes: response headers, status codes, and the actual data (response body). Using the lunch app example, imagine after you send the order, the app responds back with a confirmation message (the response). In this example, the response body could be a message indicating, “Your order for a sandwich with extra cheese, bacon, no tomatoes, no pickles, on a toasted bun has been received and is being prepared.” In other words, it is the main information the app is sending back to you from the request. Conversely, headers are like the instructions you give the app when you place the order. For example, you specify that you want the order to be delivered to your classroom. Status codes are like the response you get from the lunch app. For example, a status code of 200 is similar to the app telling you, “Your order was successfully placed and is in the process of being prepared. A status code of 404 would be like the app telling you, “The sandwich you ordered is not available or is out of stock.”

Each status code in the HTTP response tells you something about the result of an HTTP request. The most commonly used status codes are:

1. **200 OK:** This status code represents a successful request. In our lunch app, this could be the response that the order was received and is in the process of being prepared.
2. **201 Created:** This status code indicates a resource was created successfully. In our lunch app, this could be the response when you create a new account.
3. **400 Bad Request:** This status code indicates the server does not understand the request. Typically, this occurs for invalid syntax. In the lunch app, this could be a response when you try to order a sandwich but forget to specify the ingredients.
4. **401 Unauthorized:** This status code indicates that the request requires authentication. In the lunch app, this could be the response if you try to place an order for a sandwich, but you are not signed into your account.
5. **403 Forbidden:** This status code is similar to 401, because they are both about permissions. However, in this status code, the request has been authenticated, but it does not have the right permissions to access the resource. In the lunch app, this could be the response if you tried to access another customer’s order history.
6. **404 Not Found:** This status code indicates that the resource you requested is missing. In the lunch app, this could be the response if you try to order a sandwich that is not on the menu or is out of stock.

7. **500 Internal Server Error:** This status code indicates that the server had an error and could not fulfill your request. In the lunch app, this could be a response if there was a problem with the app's server when you tried to order a sandwich.

In our case, we set the HTTP status code to 200 and set the response header to "Content-Type" "text/plain". And, finally, we send "Learning Node.js is fun!" in the response body. Next, we listen on port 3000 of "localhost" and in the body we write "Server is running at http://localhost:3000/".

While this example is simple, it does illustrate the important parts of how an HTTP server is created using Node.js and the http module. Running this code (node sever.js) will start an HTTP server listening on port 3000. To test this server, open a new browser window and enter the URL http://localhost:3000/. You should see the message "Learning Node.js is fun!" displayed in the browser window. Setting the "Content-Type" to "text/plain" is why the message was printed as plain text. You can specify various content types through the response header. The mostly commonly used Content-Type values are:

1. **text/plain:** This value is used to send plain text to the client.
2. **text/html:** This value is used to send HTML to the client.
3. **application/json:** This value is used to send JSON to the client.
4. **application/javascript:** This value is used to send JavaScript to the client.
5. **image/jpeg:** This value is used to send a JPEG image to the client.
6. **image/png:** This value is used to send a PNG image to the client.
7. **multipart/form-data:** This value is used for form data.

Let's take a look at an example that returns data using application/json.

```
// File name: taco-stand-server.js
"use strict";

const http = require("http");

const server = http.createServer((req, res,) => {
  res.statusCode = 200;
  res.setHeader("Content-Type", "application/json");
  const tacoStandMenu = {
    "menu": [
```

```

    "Carne Asada Taco",
    "Chicken Taco",
    "Veggie Taco",
    "Fish Taco"
  ]
}

res.end(JSON.stringify(tacoStandMenu));
});

server.listen(3000, "localhost", () => {
  console.log("Server running at http://localhost:3000/");
});

```

Similar to the previous example, this code example creates a JSON object with a property that is an array of menu items. Next, the JSON object is converted into a string and sent to the client. To run this script, use the following command in the directory where the file is saved:

node taco-stand-server.js

To test the server, open a new terminal window and enter the following Node.js command:

```
node -e "http.get('http://localhost:3000/', (res) => { res.on('data', (chunk) => {
process.stdout.write(chunk); }); });"
```

This will send an HTTP GET request to **http://localhost:3000**, which is the URL for the web server we just created. The output should be similar to the following:

```
{"menu":["Carne Asada Taco","Chicken Taco","Veggie Taco","Fish Taco"]}
```

To stop an active terminal window, press Ctrl + C.

One thing to note is how the data is being sent back to the client. You will notice that we convert the JSON object into a string using `JSON.stringify`, which is a utility function that converts JSON objects into strings. This is necessary, because all data that is sent to the client must be in a string format. Now, you might be thinking, what is the point of setting the Content-Type in the header of the response. When you set the Content-Type, you are telling the client that whatever data was sent to it should be parsed to the specified value. For example, by setting the Content-Type to JSON, we are giving the client instructions that the data should be converted (parsed) back into a JSON object.

In others words, while the data is always sent as a string from your server, the Content-Type gives the client instructions on how to interpret the string being sent. In the next section,

we will take a look at how we can extend this example to work with HTTP request and responses.

Working with HTTP Requests and Responses

As discussed in the previous section, an HTTP server receives a request from a client and returns a response. Effectively, creating a two-way communication channel between a client. In the context of HTTP, the “client” is any program that sends an HTTP request to your server. This could be a web browser, a device (mobile app), another web server, or desktop application. Really, anything that can make an HTTP request is considered a “client.”

First, let’s take a look at some simple examples that illustrate how to work with HTTP requests.

Logging the request method and URL:

```
"use strict";
```

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

```
const server = http.createServer((req, res) => {  
  console.log(` Received a ${req.method} request for ${req.url}`);  
  res.end();  
});
```

```
server.listen(3000, () => console.log("Server started (http://localhost:3000/) !"));
```

In this example, we create a new HTTP server that is listening on port 3000. Port 3000 is the standard port a Node.js server uses to listen on for HTTP requests. In the body of the `createServer` call, `console.log` is used to display the request method and the request URL. If you run this script and open a new browser window and enter `http://localhost:3000`, the console will print:

Received a GET request for /

If we change the URL to `http://localhost:3000/tacos/chcken`, the console will print

Received a GET request for /tacos/chicken

```
// request2.js  
const http = require("http");  
const url = require("url");  
  
// Reading query parameters
```

```
const server = http.createServer((req, res) => {
  const query = url.parse(req.url, true).query;
  console.log(` Name: ${query.name}, Age: ${query.age} `);
  res.end();
});
```

```
server.listen(3000, () => console.log("Server started (http://localhost:3000/) !"));
```

in this code example, we use `url.parse`, which uses Node.js's built-in `url` module to parse the URL into an object. We do this, so we can access the objects properties (name and age). Otherwise, this code is similar to the previous example. Open a new browser window and enter the URL: `http://localhost:3000?name=Art&age=52`, the console will print

Name: Art, Age: 52

In this next example, we will take a look at how to process form data in an HTTP POST method.

Project Structure

```
taco-stand-server
  order-form.html
  order-taco.js
```

```
// File name: order-taco.js
"use strict";
```

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

```
const server = http.createServer((req, res) => {
  let body = "";
  req.on("data", chunk => {
    body += chunk.toString();
  });
```

```
  req.on("end", () => {
    console.log(` Order: ${body} `);
    res.end();
  });
});
```

```
server.listen(3000, () => console.log("Taco Stand Server started
(http://localhost:3000/) !"));
```

```
<!-- File name: order-form.html -->
<!DOCTYPE html>
```

```

<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Order Taco</title>
</head>
<body>
  <h1>Order Taco</h1>
  <form action="http://localhost:3000" method="post">
    <label for="taco">What type of taco would you like?</label>
    <input type="text" name="taco" id="taco">
    <input type="submit" value="Submit">
  </form>
</body>
</html>

```

This code creates an HTTP server that listens for incoming requests. For each request, it sets up two event listeners. This is possible because the request object is a descendant of the EventEmitter class. Two points should be made:

1. The data event is emitted whenever a chunk of the request body is received. Data sent from an HTTP request is sent as a buffer (covered in a later chapter). This is why chunk is parsed using the toString method.
2. The end event is emitted after all chunks of data are received. In the body of this listener, we write the request body to the console.

The code in the index.html is used to simulate a form submission. There is a single input field that accepts a response to “What type of taco would you like?”

To test this solution, start the server using the Node.js execute script command (node order-taco.js) from the directory where the file is saved. Next, open the order-form.html in your browser window. Fill in the input field and submit the form. You should see a message similar to the following in the server’s console window:

Order: taco=Carne+Asada+Taco

At the beginning of this section, we explored how to send plain text as an HTTP response, how to send JSON data as an HTTP response, and how to set and send status codes as an HTTP response. In the final example of this section, we will look at how to send an HTML file as a response.

Project Structure
 favorite-composers

app.js
index.html

```
// app.js
"use strict";

const http = require("http");
const fs = require("fs");

const server = http.createServer((req, res) => {
  fs.readFile("index.html", (err, data) => {
    if (err) {
      res.writeHead(500);
      return res.end("Error loading index.html");
    }
    res.writeHead(200, { "Content-Type": "text/html" });
    res.end(data);
  });
});

server.listen(3000, () => {
  console.log("Server running on port 3000");
});

// index.html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Favorite Composers</title>

  <!-- inline styles -->
  <style>
    body {
      font-family: sans-serif;
      background-color: #333;
      color: #fff;
    }
    h1 {
      color: yellow;
    }
    ul {
      list-style-type: none;
```

```

padding: 0;
}
li {
padding: 0.5rem;
border-bottom: 1px solid #ccc;
}
li:last-child {
border-bottom: none;
}

#container {
width: 50%;
margin: 0 auto;
}
</style>
</head>
<body>
<div id="container">
<h1>Favorite Composers</h1>
<ul>
<li>Ludwig van Beethoven</li>
<li>Johann Sebastian Bach</li>
<li>Wolfgang Amadeus Mozart</li>
<li>Johannes Brahms</li>
<li>Richard Wagner</li>
<li>Claude Debussy</li>
<li>Pyotr Ilyich Tchaikovsky</li>
</ul>
</div>
</body>
</html>

```

This code example, creates a new HTTP server that sends the index.html file as a string to the client (browser). It uses Node.js's built in fs module (covered in a later chapter) to read the file and sent its content as a string. The Content-Type is set to "text/html" to instruct the browser to handle this string as HTML content. I won't go into details on how the fs module works, because there is an entire chapter dedicated to it. For now, the goal of this example is to illustrate how you can use the http module to send HTML pages from an HTTP server. To run this program, start the script (node app.js) and open a new browser and enter http://localhost:3000. You should see a listing of favorite composers.

In the next section, we will take a look at how routing works in a Node.js HTTP server.

Routing in Node.js HTTP Server

Routing is loosely defined as the navigation between endpoints in an HTTP server. It is how the HTTP server determines how to respond to different types of client requests. Endpoints are URLs that point to functions or resources on an HTTP server or application. Consider the following, a Greenhouse (our HTTP server) has many sections (endpoints) and each section is for a different type of plant (a specific function or resource on our server). For example, there could be a Roses section, Orchids section, and a Cacti section. When you want to visit a plant, you go to its section. This is just like when you want to access something on an HTTP server, you go to a location (endpoint) where the resource exists.

Routing, is like the map or the directions that guide you to each section of the Greenhouse. If you want to visit the Cacti section, you would use the map for directions on how to get to the section. Routing is used in the same context; it is a set of directions to the resource you are requesting. Routing rules are defined to determine which function (think Greenhouse section) or resource is executed based on the endpoint (plant section) you requested.

In a web application, there are many endpoints (Greenhouse sections) for different functions and resources (Greenhouse plants), like `/roses`, `/orchids`, and `/cacti`. And, routing rules are used (the map) to instruct the application how to respond to someone's requests for an endpoint (visiting a section in the green house).

Let's take a look at a simple HTTP server example that demonstrates how routing and API endpoints work in a Node.js application.

```
// greenhouse-server.js
"use strict";

const http = require("http");

const server = http.createServer((req, res) => {
  if (req.url === "/roses") {
    res.write("Welcome to the Roses section!");
    res.end();
  } else if (req.url === "/orchids") {
    res.write("Welcome to the Orchids section!");
    res.end();
  } else if (req.url === "/cacti") {
    res.write("Welcome to the Cacti section!");
    res.end();
  } else {
    res.write("Welcome to the Greenhouse!");
    res.end();
  }
});
```

```
});
```

```
server.listen(3000, () => console.log("Greenhouse Server started  
(http://localhost:3000/) !"));
```

In this code example, we have three endpoints: /roses, /orchids, and /cacti. Each endpoint represents a different section in the Greenhouse. When a client or user requests one of these endpoints, the server will respond with a message welcoming them to the section they selected. However, if the client requests an endpoint that does not exist, for example /sunflowers, they are redirected to the main lobby and a default message of “Welcome to the Greenhouse.” To test this script, start the server (node greenhouse-server.js), open a new browser window, and enter the following URLs:

1. **http://localhost:3000:** “Welcome to the Greenhouse!” should be displayed in the browser window.
2. **http://localhost:3000/roses:** “Welcome to the Roses section!” should be displayed in the browser window.
3. **http://localhost:3000/orchids:** “Welcome to the Orchids section!” should be displayed in the browser window.
4. **http://localhost:3000/cacti:** “Welcome to the Cacti section!” should be displayed in the browser window.

Now that you have an idea of how routing and endpoints work in a Node.js application, let’s take a look at a program that passes data to endpoints using the query parameter object:

```
// taco-order-server.js  
"use strict";  
  
const http = require("http");  
const url = require("url");  
  
const server = http.createServer((req, res) => {  
  const queryObject = url.parse(req.url, true).query;  
  
  if (req.url.startsWith("/beef")) {  
    const topping = queryObject.topping;  
    if (topping === "cheese") {  
      res.write("You've ordered a beef taco with cheese!");  
    } else if (topping === "salsa") {  
      res.write("You've ordered a beef taco with salsa!");  
    } else {
```

```

    res.write("You've ordered a plain beef taco.");
  }
  res.end();
} else if (req.url.startsWith("/chicken")) {
  const topping = queryObject.topping;
  if (topping === "guacamole") {
    res.write("You've ordered a chicken taco with guacamole!");
  } else if (topping === "sourcream") {
    res.write("You've ordered a chicken taco with sour cream!");
  } else {
    res.write("You've ordered a plain chicken taco.");
  }
  res.end();
} else if (req.url.startsWith("/veggie")) {
  const topping = queryObject.topping;
  if (topping === "beans") {
    res.write("You've ordered a veggie taco with beans!");
  } else if (topping === "rice") {
    res.write("You've ordered a veggie taco with rice!");
  } else {
    res.write("You've ordered a plain veggie taco.");
  }
  res.end();
} else {
  res.write("Welcome to the Taco Stand! Visit /beef, /chicken, or /veggie with appropriate query parameters to place your order.");
  res.end();
}
});

server.listen(3000, () => console.log("Taco Order Server started (http://localhost:3000/)!"));

```

In this code example, endpoints have been configured for /beef, /chicken, and /veggie. Next, the query parameter “topping” is used to customize the selected taco. That is, code in each endpoint accepts a query parameter value that determines the taco’s toppings. If statements are needed to conditionally check which query parameter values are being used as actual parameters. Query parameters are defined using the format /endpoint?queryParameter=value. To test this script, start the server (node taco-order-server.js), open a new browser window, and enter the following URLs:

1. **http://localhost:3000:** “Welcome to the Taco Stand! Visit /beef, /chicken, or /veggie with appropriate query parameters to place your order” should be displayed in the browser window.

2. **http://localhost:3000/beef:** “You’ve ordered a plain beef taco” should be displayed in the browser window.
3. **http://localhost:3000/chicken:** “You’ve ordered a plain chicken taco” should be displayed in the browser window.
4. **http://localhost:3000/veggie:** “You’ve ordered a plain veggie taco” should be displayed in the browser window.
5. **http://localhost:3000/beef?topping=cheese:** “You’ve ordered a beef taco with cheese!” should be displayed in the browser window.
6. **http://localhost:3000/beef?topping=salsa:** “You’ve ordered a beef taco with salsa!” should be displayed in the browser window.
7. **http://localhost:3000/chicken?topping=guacamole:** “You’ve ordered a chicken taco with guacamole!” should be displayed in the browser window.
8. **http://localhost:3000/chicken?topping=sourcream:** “You’ve ordered a chicken taco with sour cream!” should be displayed in the browser window.
9. **http://localhost:3000/veggie?topping=beans:** “You’ve ordered a veggie taco with beans!” should be displayed in the browser window.
10. **http://localhost:3000/veggie?topping=rice:** “You’ve ordered a veggie taco with rice!” should be displayed in the browser window.

In the next section, we will look at how to use TDD principles with an HTTP server.

TDD with an HTTP Server

In this section we will take a look at how to use TDD principles to build and test an HTTP server. But, before we do, let’s take a look at some standard file naming conventions for Node.js projects.

- **server.js:** this is often used when a file’s primary responsibility is to start up a server. For simple applications with only a few routes, this naming convention is preferred.
- **app.js:** this is often used when the file contains the main applications logic (configurations, routes, endpoints, etc.). For complex applications, where you are configuring a view engine, logger, encryption, etc., this naming convention is preferred.

- **index.js**: this is often used as the entry point for building Node.js modules.

Example 1: Testing a JSON response of grocery items:

In this example, we are going to create an HTTP server that responds with a JSON payload of grocery items when a GET request is made to the `/grocery` route. We will use TDD principles to guide the direction of our application.

Project Structure:

```
grocery-items-api
src
  server.js
test
  server.spec.js
package.json
```

1. Write the failing tests (Red):

```
"use strict";

const http = require("http");
const server = require("../src/server");

describe("server", () => {
  afterAll(() => {
    server.close();
  });

  test("responds with grocery items in JSON format", done => {
    http.get("http://localhost:3000/grocery", res => {
      let data = "";
      res.on("data", chunk => {
        data += chunk;
      });
      res.on("end", () => {
        expect(JSON.parse(data)).toEqual({items: ["apple", "banana", "carrot"]});
        done();
      });
    });
  });
});
```

2. Make the tests pass (Green):

```
"use strict";
```

```

// server.js
const http = require('http');

const server = http.createServer((req, res) => {
  if (req.url === '/grocery' && req.method === 'GET') {
    res.writeHead(200, { 'Content-Type': 'application/json' });
    res.end(JSON.stringify({ items: ['apple', 'banana', 'carrot'] }));
  } else {
    res.writeHead(404);
    res.end();
  }
});

server.listen(3000);

module.exports = server;

```

3. Refactor:

Example 2: Testing three routes with query parameters for ordering a sandwich in a food app.

In this example, we are going to create an HTTP server with three routes: /order, /checkout, and /confirm. Each route will accept query parameters. We will use TDD principles to guide our development efforts.

Project Structure:

Sandwich-ordering-api

src

server.js

test

server.spec.js

package.json

1. Write the failing tests (Red):

```

// server.spec.js
const http = require('http');
const server = require('../src/server');

describe('Server', () => {
  afterAll(() => {

```

```

    server.close();
  });

  test('responds to /order POST request with query parameter', done => {
    const options = {
      hostname: 'localhost',
      port: 3000,
      path: '/order?item=pizza',
      method: 'POST'
    };
    const req = http.request(options, res => {
      let data = '';
      res.on('data', chunk => {
        data += chunk;
      });
      res.on('end', () => {
        expect(res.statusCode).toBe(201);
        expect(data).toBe('Order for pizza has been placed.');
```

done();

```

      });
    });
    req.end();
  });

  test('responds to /checkout GET request', done => {
    http.get('http://localhost:3000/checkout', res => {
      let data = '';
      res.on('data', chunk => {
        data += chunk;
      });
      res.on('end', () => {
        expect(res.statusCode).toBe(200);
        expect(data).toBe('Checkout page.');
```

done();

```

      });
    });
  });

  test('responds to /confirm POST request', done => {
    const options = {
      hostname: 'localhost',
      port: 3000,
      path: '/confirm',
      method: 'POST'
    };

```

```

};
const req = http.request(options, res => {
  let data = "";
  res.on('data', chunk => {
    data += chunk;
  });
  res.on('end', () => {
    expect(res.statusCode).toBe(200);
    expect(data).toBe('Order has been confirmed.');
```

2. Write the code so the tests pass (Green):

```
"use strict";
```

```
// server.js
```

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

```
const server = http.createServer((req, res) => {
  const parsedUrl = url.parse(req.url, true); // Parse the URL and the query
  parameters
  const pathname = parsedUrl.pathname; // Get the path
  const query = parsedUrl.query; // Get the query parameters as an object
```

```

  if (pathname === '/order' && req.method === 'POST') {
    // You can now access query parameters with query.parameterName
    const item = query.item;
    res.writeHead(201);
    res.end(` Order for ${item} has been placed.`);
  } else if (pathname === '/checkout' && req.method === 'GET') {
    res.writeHead(200);
    res.end('Checkout page.');
```

```

  } else if (pathname === '/confirm' && req.method === 'POST') {
    res.writeHead(200);
    res.end('Order has been confirmed.');
```

```

  } else {
    res.writeHead(404);
    res.end();
  }
}
```

```
    }  
  });  
  
  server.listen(3000);  
  
  module.exports = server;
```

3. Refactor:

Programming Exercises

In this assignment, you will create a character creation screen for a fantasy video game. You will use Node.js's built in http module and apply TDD principles using Jest.

Instructions:

1. Create a new folder for your project and name it `fantasy-game-character-creation`.
2. In this folder create the following folder structure:

```
src  
  server.js  
test  
  server.spect.js  
package.json
```

3. In your `package.json` file, add a test script and a start script. The start script should start your server, listening on port 3000.
4. Your server should have at least 3 routes:
 - a. A POST route for creating a character. This route should accept query parameters for the character's class (Warrior, Mage, Rogue), gender (Male, Female, Other), and a fun fact about your character.
 - b. A POST route for confirming the character creation.
 - c. A GET route for viewing the character. This must be the same character that was created in step 4.a.
5. Write unit tests for each of these routes using Jest. You should write the tests before you implement the routes, following TDD principles.

Grading

You will earn 20 points for passing each unit test, for a total of 60 points. If two tests pass, you will earn 40 points.

Hints:

- Use the `http.createServer` method to create your server.
- Use the `url.parse` method to parse the requested URL and extract the query parameters.
- Remember to listen for the “data” and “end” events on the request object when handling POST requests.

Chapter 7. Buffers and Streams

Chapter Overview

In this chapter we will dive into two fundamental concepts in Node.js: Buffers and Streams. These concepts are central to the Node.js ecosystem and provide a foundation for understanding more advanced computing concepts. We start by exploring what Buffers are in Node.js and how to create them. Next, we will define what Streams are in Node.js and experiment with various techniques for creating and managing streams. This includes reading, writing, and transforming data with streams. Lastly, we will look at how to pipe and chain streams together and how to use TDD principles to drive the development of a Node.js program that uses Streams.

Learning Objectives

By the end of this chapter, you should be able to:

- Define buffers.
- Explain how streams are used for handling data.
- Identify the differences between readable, writable, and transform streams.
- Build a Node.js program using streams.

Introduction to Buffers in Node.js

Buffers, in the context of computer programming, is a storage mechanism for storing data temporarily. Data is stored in a Buffer and retrieved by some type of input device (keyboard) before it is delivered to some type of output device (display screen).

Imagine you are at a party and there is a bowl of pretzels. And, people are eating the pretzels and someone is constantly refilling the bowl when it gets low. The bowl is like a buffer. It temporarily holds the pretzels (or data) until people (or the computer) are ready to consume them.

If the bowl gets empty, everyone has to wait until the bowl is refilled to eat more pretzels. If the bowl gets too full, the pretzels will spill out the sides. So, to avoid a mess, it is important to keep the bowl (buffer) filled to a good level (not too full, but not too empty either). This is similar to how online streaming platforms work. You click on a movie to watch, but it does not play immediately, instead, there is a loading period where you are stuck looking at some type of spinner. Well, in the background, the system is loading a little bit of the movie into a buffer before it starts playing. When you watch the beginning of the movie, the next part of the movie is added to the buffer. This process is repeated throughout the entire movie (similar to refilling the pretzel bowl). The goal is to keep the movie from being interrupted and to ensure it is running smoothly.

In Node.js, Buffer is a global object that is used to work with binary data. Binary data is data stored in a binary format (1's and 0's). This is the format that a computer understands. And, there are several advantages to binary data, which includes:

1. **Efficiency:** Computers use binary data because it is efficient. This is because computers are made up of electronic circuits that only have two states: on and off. One is considered on and 0 is considered off. Working with data in the format that computers understand is very efficient.
2. **Versatility:** Binary data can represent any type of data. This includes: text, images, audio files, executable programs, and video files. This flexibility means you can easily work with all types of data formats in a single program.
3. **Data Transmission:** Binary format is used for data transmission.

Let's take a look at how to create a Buffer in Node.js

```
// buffer-gardening.js
"use strict";

const wateringDaysBuff = Buffer.alloc(10, 7);

console.log(wateringDaysBuff);

const plantBuffer = Buffer.from("RoseBush");

console.log(plantBuffer);
```

In this example, the first line creates a buffer of 10 bytes and fills it with the number 7, representing the 7 days of the week for watering plants. The next buffer is created with a string value of "RoseBush." Running this script will produce the following:

```
<Buffer 07 07 07 07 07 07 07 07 07 07>
```


<Buffer 52 6f 73 65 42 75 73 68>

As you can see from this output, the data is unreadable because it was printed in buffer format. The Buffer constructor is a global object in Node.js. This means, you do not need to import a module or create a new instance of the Buffer object. Instead, you call the Buffer constructor directly in your Node.js program. You can view the Buffer's properties by running the following command:

node -p "Buffer"

```
[Function: Buffer] {
  poolSize: 8192,
  from: [Function: from],
  copyBytesFrom: [Function: copyBytesFrom],
  of: [Function: of],
  alloc: [Function: alloc],
  allocUnsafe: [Function: allocUnsafe],
  allocUnsafeSlow: [Function: allocUnsafeSlow],
  isBuffer: [Function: isBuffer],
  compare: [Function: compare],
  isEncoding: [Function: isEncoding],
  concat: [Function: concat],
  byteLength: [Function: byteLength],
  [Symbol(kIsEncodingSymbol)]: [Function: isEncoding]
}
```

And, as shown in the previous example, you can create a new buffer by using the alloc method.

```
const buffer = Buffer.alloc(10);
console.log(buffer);
buffer instanceof Buffer
```

The above commands were executed using Node.js's built-in REPEL tool. You can access this tool by entering node from the CLI. To exit the tool, enter Ctrl + C.

```
> const buffer = Buffer.alloc(10);
undefined
> console.log(buffer);
<Buffer 00 00 00 00 00 00 00 00 00 00>
undefined
> buffer instanceof Buffer
true
```

You can also, optionally, specify a value to include in the buffer by passing a second argument to the `alloc` method (this was demonstrated in the above program). Another option is to use the method `allocUnsafe(size)`. This method creates a new buffer of the specified size, without initializing it. This is considered the “unsafe” way to allocate a buffer. And, `alloc` is considered the “safe” way. The reason why `allocUnsafe` is considered “unsafe” is because the buffer instance might contain older data that will need to be overwritten.

```
> const unsafeBuffer = Buffer.allocUnsafe(10);
undefined
> console.log(unsafeBuffer);
<Buffer 00 00 00 00 00 00 00 00 00 00>
undefined
```

Here is a list of the most commonly used Buffer methods:

1. **Buffer.from(array) or Buffer.from(string, [encoding]):** This creates a new buffer containing a copy of the specified data.
2. **Buffer.alloc(size, [fill], [encoding]):** This creates a buffer of the provide size. In addition, you can specify the value to be filled and the encoding type.
3. **Buffer.allocUnsafe(size):** this creates a buffer of the provided size, without initializing it. This is faster than `alloc`, but is considered “unsafe.”
4. **Buffer.concat(list, [totalLength]):** This method concatenates a list of Buffer instances into a single Buffer. Similar to how you would concatenate strings together to form a single string. This method works in a similar fashion.
5. **Buffer.length:** This method returns the length of the buffer in bytes.

```
> console.log(unsafeBuffer.length);
10
```

6. **Buffer.fill(value, [offset], [end], [encoding]):** This method fills the buffer with the value.
7. **Buffer.slice([start], [end]):** this method returns a new Buffer that references the same memory location as the original, but with cropping by the start and end indices.
8. **Buffer.copy(targetBuffer, [targetStart], [sourceStart], [sourceEnd]):** this method copies data from a region in a buffer to the targeted region of another buffer.

9. **Buffer.equals(otherBuffer)**: this method returns true if the buffer is an exact match of the argument buffer and false if not.

```
[> console.log(unsafeBuffer.equals(buffer));  
false  
undefined  
> █
```

Let's take a look at how we can use some of the most commonly used Buffer methods in a simple Node.js program. Consider the following:

```
// gardening-buffer-methods.js  
"use strict";  
  
// create a Buffer with the text "RoseBush"  
const roseBushBuffer = Buffer.from("RoseBush");  
  
// print the length of the buffer  
console.log(` Length of RoseBush buffer: ${roseBushBuffer.length}`);  
  
// create a Buffer with the text " and Tulips"  
const andTulipsBuffer = Buffer.from(" and Tulips");  
  
// concatenate the buffers  
const plantsBuffer = Buffer.concat([roseBushBuffer, andTulipsBuffer]);  
  
// print the concatenated buffer  
console.log(` Concatenated buffer: ${plantsBuffer.toString()}`);  
  
// fill the plantsBuffer with "Sunflowers"  
plantsBuffer.fill("Sunflowers");  
  
// print the filled buffer  
console.log(` Filled buffer: ${plantsBuffer.toString()}`);  
  
// create a slice of the plantsBuffer  
const sliceBuffer = plantsBuffer.slice(0, 5);  
  
// print the sliced buffer  
console.log(` Sliced buffer: ${sliceBuffer.toString()}`);
```

In this program, we first create a buffer with the text “RoseBush” and print its length to the console window. Then we create another buffer with the text “and Tulips” and concatenate

it with the first buffer. After that, we fill the concatenated buffer with the text “Sunflowers” and print it. Finally, we create a slice of the filled buffer and print it.

Depending on the text editor you are using, you may notice that the word “slice” has a line through it with a deprecation warning. The deprecation warning is coming from TypeScript, not JavaScript. TypeScript (outside the scope of this book) has much stricter rules than JavaScript. In the context of this chapter’s explanation on buffers, you can safely ignore this warning message. To test this script, execute it using the Node.js command (node gardening-buffer-methods.js). You should see the following messages printed to the console window:

Length of RoseBush buffer: 8

Concatenated buffer: RoseBush and Tulips

Filled buffer: SunflowersSunflower

Sliced buffer: Sunfl

Now, let’s take a look at two examples on how we can convert a buffer back into its original value using our gardening theme:

1. **Using `buffer.toString()` method:** This method converts the buffer’s data to a string with whatever encoding type you specify. If no encoding type is provided, it defaults to “utf-8”

```
[> const roseBushBuffer = Buffer.from("RoseBush");
undefined
[> console.log(roseBushBuffer);
<Buffer 52 6f 73 65 42 75 73 68>
undefined
[> const str = roseBushBuffer.toString();
undefined
[> console.log(str);
RoseBush
undefined
> █
```

In this example, a buffer is defined using the `from` method and assigned to the variable `roseBushBuffer`. Then we print the buffer to the console using JavaScript’s built-in `console.log()` function. After that, we use JavaScript’s built-in `toString()` method to convert the buffer back into a readable text value, which is stored in the `str` variable. Finally, the converted buffer is printed to the console window using JavaScript’s built-in `console.log()` method.

2. **Using `buffer.toJSON()` method:** This method returns a JSON representation of the buffer. The JSON object has two properties: `type` and `data`. The `type` property is a

string that indicates what the JSON object's type is (in our case 'Buffer'). Coincidentally, for a buffer, the type will always be 'Buffer.' The data property is an array that contains the buffers content as a series of numbers, which represent the bytes of that buffer.

```
[> const roseBushBuffer = Buffer.from("RoseBush");
undefined
[> const json = roseBushBuffer.toJSON();
undefined
[> console.log(json);
{
  type: 'Buffer',
  data: [
    82, 111, 115, 101,
    66, 117, 115, 104
  ]
}
undefined
[> const originalBuffer = Buffer.from(json.data);
undefined
[> const originalString = originalBuffer.toString();
undefined
[> console.log(originalString);
RoseBush
```

Understanding how to work with buffers is important for several reasons. Below outlines some of those advantages as it relates to software development:

1. **Performance:** Buffers allow you to directly interact with binary data, which as previously mentioned, can be more efficient than working with the data types directly.
2. **Handling Binary Data:** Buffers are essential for dealing with binary data. Some common tasks that work with binary data includes: reading and writing to a file system, network connections, and databases. Buffers are also used to work with images, audio files, video files, and other types of streaming data.
3. **Consistent API:** Buffers provide a consistent way for dealing with data at a lower programming level. This is very useful for when you are working with data from different data sources, but need a single consistent way for transmitting the data.
4. **Control over Bytes:** Buffers give you control over bytes and byte order.

5. **Data Streaming (covered in the next section):** Buffers are used when data is streamed from one process to another. Buffers allow you to work with “chunks” of data as it arrives. This is a much better approach to the alternative of waiting for all the data to be received before processing it.

In the next section we will take a look at how to work with streams in Node.js.

Understanding Streams in Node.js

In the context of computer science, a stream is a sequence of data elements made available over a period of time. The ability to handle “chunks” of data is particularly useful when you are working with large amounts of data or when the overall size of the data is not known until during the time of its execution. Streams are used in many areas of computing:

1. **File I/O (covered in another chapter):** Streams are often used whenever you are reading or writing data to a file. This is because with streams, you can handle the data in “chunks” rather than loading the entire data file into memory before processing it (reading/writing).
2. **Network I/O:** Streams are used heavily in network operations. When data is sent or received over a network it is often transmitted as a stream. Similar to the explanation provided for File I/O, streams allow network transmissions to be handled as the data arrives or as it is ready to be sent. This flexibility ensures data is fluent.
3. **Pipelines (covered later):** Streams can be piped together to create complex data pipelines. This is especially useful when you are working with large files or when you are needing to read data from multiple files and then write the contents to a single file.

Streams in Node.js are objects that let you read data from one location and write it to a separate location. As previously mentioned, this is especially useful when you are working with large amounts of data. There are four types of streams in Node.js

1. **Readable:** These are data streams that can be read, such as, reading data from a file or reading data from an HTTP request.
2. **Writable:** These are data streams that can be written, such as, writing data to a file or sending an HTTP response.
3. **Duplex:** These are data streams that are both readable and writable.
4. **Transform:** These are duplex data streams that can transform data as it is written or read. That is, you can write, read, and transform data using a transform stream.

And all streams provide events that can be used to track the lifecycle of the stream:

1. **data**: This event is emitted when data is ready to be read.
2. **end**: This event is emitted when there is no more data to be read.
3. **error**: This event is emitted when there is an error.
4. **finish**: This event is emitted when all data has been outputted. This occurs, once the `end()` method has been called. In other words, all data has been flushed from the underlying system.

Let's first take a look at how to use readable streams, then we will explore writable streams, and finally we will finish the section by learning how to use transform streams.

Readable Streams:

1. Creating a Readable Stream from a String:

```
// taco-stream.js
"use strict";

const { Readable } = require("stream");

const tacos = "Beef, Chicken, Veggie, Fish, Carnitas, Barbacoa";

const readableStream = Readable.from(tacos);

readableStream.on("data", (chunk) => {
  console.log(chunk.toString());
})

readableStream.on("end", () => {
  console.log("No more data.")
});
```

In this example, we define a variable named `tacos` with a string value representing the tacos we offer in our taco stand. Next, we define a `readableStream` using `Readable.from()`, passing in the `tacos` string as an actual parameter. Then we set up event listeners for the “data” and “end” events. When the data is available, the “data” event is triggered and we write the content to the console window. When there is no more data to write, the “end” event is triggered and we write a message to the console window. To test this script, execute the script using the Node.js execute

command (node taco-stream.js). The following messages should be displayed in the console window:

Beef, Chicken, Veggie, Fish, Carnitas, Barbacoa
No more data.

2. Creating a Custom Readable Stream:

```
// taco-readable-stream.js
"use strict";

const { Readable } = require("stream");

const tacos = ["Beef", "Chicken", "Veggie", "Fish", "Carnitas", "Barbacoa"];
let index = 0;

const readableStream = new Readable({
  read() {
    if (index === tacos.length) {
      this.push(null);
    } else {
      this.push(tacos[index]);
      index++;
    }
  }
});

readableStream.on("data", (chunk) => {
  console.log(chunk.toString());
});
```

In this example, we define an array of the various types of tacos we offer in our taco stand. Next, we use the Readable constructor to create a new readable stream, passing in an anonymous object literal as the actual parameter. The object literal has one property called “read.” The read property is used to read the contents of the tacos array and push the values to the stream. Then we register an event listener for the “data” event to write our menu to the console window.

The constructor for the Readable stream is an instance of the EventEmitter class. This means that instances of the Readable stream can emit events, and you can register event listeners for these events in the same way you learned how to work with the EventEmitter class in the previous sections. This event-driven nature of streams is how we can handle “chunks” of data as they become available in the stream. To test this script, execute the script using the Node.js execute command

(node taco-readable-stream.js). You should see the following messages printed to the console window:

```
Beef  
Chicken  
Veggie  
Fish  
Carnitas  
Barbacoa
```

Writable Streams:

1. Creating a Custom Writable Stream:

```
// taco-writable-stream.js  
"use strict";  
  
const { Writable } = require("stream");  
  
const writableStream = new Writable({  
  write(chunk, encoding, callback) {  
    console.log(chunk.toString());  
    callback();  
  }  
});  
  
writableStream.write("Beef");  
writableStream.write("Chicken");  
writableStream.end("Veggie");
```

In this example, we use the Writable constructor to create a new writable stream, passing in an anonymous object literal as the actual parameter. The object literal has one property called “write.” The write property is used to log the data it receives to the console window. A writable stream takes three formal parameters: chunk, encoding, and callback. Chunk is the data the writable stream receives, encoding is the encoding type you want to use for the writable stream, and callback is a callback function. Next, we write “Beef”, “Chicken”, and “Veggie” to the stream using the write() method. To test this script, execute the script using the Node.js execute command (node taco-writable-stream.js). You should see the following messages printed to the console window:

```
Beef  
Chicken  
Veggie
```

we create a writable stream that logs the data it receives to the console window.

In this example, we'll create a writable stream that writes data to the console.

```
"use strict";

const { Writable } = require("stream");

const writableStream = new Writable({
  write(chunk, encoding, callback) {
    console.log(chunk.toString());
    callback();
  }
});

writableStream.write("Superman");
writableStream.write("Batman");
writableStream.end("Wonder Woman");
```

2. Creating a Custom Writable Stream with Finish Event Handling:

```
// taco-writable-stream-with-finish.js
"use strict";

const { Writable } = require("stream");

let tacos = [];

const writableStream = new Writable({
  write(chunk, encoding, callback) {
    tacos.push(chunk.toString());
    callback();
  }
});

writableStream.on("finish", () => {
  console.log(tacos);
});

writableStream.write("Beef");
writableStream.write("Chicken");
writableStream.end("Veggie");
```

This example is very similar to the previous one. We create a writable stream and write data to it using the `write()` method. However, the distinction is in how we process the data. In the first example, each chunk of data is written to the stream immediately and logged to the console window. In this example, we register an event listener for the “finish” event and wait for the event to trigger before logging the data to the console window. An array is used to store the writable data (tacos) during each `write()` call.

3. **Error Handling:** The callback function is a function that you can call when you are done processing the data. If an error occurs, you can pass the Error object as the first actual parameter in the callback function. Here is an example of how to incorporate error handling, using the callback function.

```
// taco-writable-stream-with-error-handling.js
"use strict";
```

```
const { Writable } = require("stream");
```

```
let tacos = [];
```

```
const writableStream = new Writable({
  write(chunk, encoding, callback) {
    try {
      tacos.push(chunk.toString());
      callback();
    } catch (err) {
      callback(err);
    }
  }
});
```

```
writableStream.on("finish", () => {
  console.log(tacos);
});
```

```
writableStream.on("error", (err) => {
  console.error("Stream write failed: ", err);
});
```

```
try {
  writableStream.write("Beef");
  writableStream.write("Chicken");
  writableStream.end("Veggie");
} catch (err) {
```

```
    console.error("Failed to write data: ", err);  
  }
```

In this example, we add two try-catch blocks to the program: one in the write method and one when calling the write method. If an error occurs while pushing the chunk of data into the array, we catch the error and pass it to the callback function. This will cause an error event to be emitted. An error event listener is registered to log the emitted error to the console. Finally, a second try-catch block is used to catch any errors that are encountered from calling the write method. The second try-catch block is needed to catch any errors that occur synchronously when calling the write() and end() methods. The errors encountered in this second try-catch block could be different than what's being passed in the write() methods callback function.

In the next section, we will take a look at the Duplex and Transform streams

Reading, Writing, and Transforming Data with Streams

Duplex and Transform are two types of streams in Node.js that combine the features of writable and readable streams into a single stream. That is, they are both readable and writable. Here is a brief explanation of each stream:

1. **Duplex Streams:** A Duplex stream is both readable and writable stream. Imagine Batman and Superman are in a battle and they need to communicate with each other. Batman sends a message (writes data) to Superman through his Bat communicator. Superman receives the message (reads) the data and responds to it (writes data), and Batman receives the response (reads the data). Two-way communication is like a Duplex stream, where both parties (Batman and Superman) can send and receive messages.
2. **Transform Stream:** A Transform stream is a type of Duplex stream, but with the ability to transform the input. That is, chunks of data are transmitted, processed, and transformed. Imagine you have a friend who only speaks Spanish, and you only speak English. You have a mutual friend who speaks both languages and can translate English to Spanish. This is like a Transform stream. You speak English to the translator (write data), the translator translates it to Spanish (transforms data), and your friend hears the Spanish translation (reads data).

Let's take at a Node.js programs that uses the Duplex stream for two-way communication.

```
// duplex.js  
"use strict";  
  
const { Duplex } = require("stream");
```

```

class SuperheroCommunicator extends Duplex {
  constructor() {
    super();
  }

  _write(chunk, encoding, callback) {
    console.log(` Batman sends: ${chunk.toString()} `);
    callback();
  }

  _read(size) {
    this.push(` Superman responds: I received your message, Batman. `);
    this.push(null);
  }
}

const communicator = new SuperheroCommunicator();

communicator.on("data", (chunk) => {
  console.log((chunk.toString()));
});

communicator.write("Superman, do you copy?");

```

In this example, we create a SuperheroCommunicator class that extends the Duplex stream class. The `_write` method logs the message that Batman sends, and the `_read` method pushes Superman's response. When we write to the communicator stream, it logs Batman's message and when we read from the communicator stream, it logs Superman's response. Next, we will take a look at how to use the Transform stream in a Node.js program.

```

// transform.js
"use strict";

const { Transform } = require("stream");

const simpleTranslation = {
  "hello": "hola",
  "world": "mundo",
  "programming": "programar",
  "is": "es",
  "fun": "divertido",
};

```

```

class EnglishToSpanishTranslator extends Transform {
  constructor() {
    super();
  }

  _transform(chunk, encoding, callback) {
    const words = chunk.toString().split(" ");
    const translatedWords = words.map(word =>
simpleTranslation[word.toLowerCase()] || word);
    this.push(translatedWords.join(" "));
    callback();
  }
}

const translator = new EnglishToSpanishTranslator();

translator.on("data", (chunk) => {
  console.log(` Your translated text: ${chunk.toString()} `);
});

translator.write("hello world programming is fun");

```

In this example, we create a `EnglishToSpanishTranslator` class that extends the `Transform` stream class. The `_transform` method translates English words to Spanish words and pushes the translated sentence. When we write to the translator, it translates the English sentence to Spanish. And, when we read from the translator, it logs the Spanish translation.

Let's say we have a superhero named "EchoMan" who has the power to repeat what you say but in reverse order. We could use the `Transform` stream to simulate this super power.

```

"use strict";

const { Transform } = require("stream");

class EchoManPower extends Transform {
  constructor() {
    super();
  }

  _transform(chunk, encoding, callback) {
    const sentence = chunk.toString();
    const reversedSentence = sentence.split("").reverse().join(" ");

```

```

    this.push(reversedSentence);
    callback();
  }
}

const echoManPower = new EchoManPower();

echoManPower.on("data", (chunk) => {
  console.log(`EchoMan repeats: ${chunk.toString()}`);
});

echoManPower.write("I am EchoMan, I am here to help you with your problems and to save the world!");

```

In this code, we create an `EchoManPower` class that extends the `Transform` stream class. The `_transform` method takes a sentence and reverses the order of the words, and then pushes the reversed sentence. Next, an instance of the `EchoManPower` is created. When we write to the `echoManPower` stream it reverses the order of the sentence. And, when we read from the `echoManPower` stream it prints the reversed sentence.

In the next section we will take a look at how to pipe and chain streams together.

Piping and Chaining Streams

Piping and chaining streams together in Node.js are an efficient way to handle large amounts of data or resources that are difficult or slow to load. To explain how piping works, imagine a group of superheroes are working to stop a meteor from hitting Earth. Each superhero has a unique power and they need to use these powers in a specific order to stop the meteor from crashing into Earth.

1. **Superman:** uses his super strength to fly into space and push the meteor towards Earth's atmosphere.
2. **Flash:** uses his super speed to quickly build a giant net to catch the meteor.
3. **Green Lantern:** uses his ring to create a giant cushion in the net to soften the meteor's landing.

In this scenario, the meteor is the data, and the superheroes are the streams. Superman is a readable stream where data comes in. Flash and Green Lantern are transform streams that modify the data. The process of passing the meteor from one superhero to another is similar to how piping works in programming. That is, you chain multiple streams together to produce some type of result. Much like how each superhero takes the output of the previous stream, piping takes the output of one stream, does something with it, and passes

it to the next stream. This allows us to take an extremely complex process and break it down into smaller and more manageable operations. Thus, making your code easier to understand, read, and scale.

Let's take a look at a simple program that uses pipes.

```
// pipe.js  
"use strict";  
  
const { Transform } = require("stream");  
  
class ToUpperCase extends Transform {  
  constructor() {  
    super();  
  }  
  
  _transform(chunk, encoding, callback) {  
    this.push(chunk.toString().toUpperCase());  
    callback();  
  }  
}  
  
const upperCaser = new ToUpperCase();  
  
process.stdin.pipe(upperCaser).pipe(process.stdout);
```

In this code example, we create a `ToUpperCase` class that extends the `Transform` stream class. The `_transform` method converts the chunk of data to a string and then uppercases it. We then create an instance of the `ToUpperCase` and pipe `process.stdin` (the standard input stream, which reads input from the command line), into uppercase, and then pipe `upperCaser` into `process.stdout` (the standard output stream, which writes output to the command line). If you run this program and enter a command, you will notice that the output has been uppercased. This is possible because both `process.stdin` and `process.stdout` are streams (readable and writable).

In Node.js, any stream is pipeable. This is because the `pipe()` method is part of the `Readable` stream prototype, which makes it available on any stream object. Calling `process.stdin.pipe` is possible because of this feature. Just remember, you can only pipe from a `Readable` stream to a `Writable` stream.

Another option is to use Node.js's built-in `pipeline()` utility function. It is similar to chaining the `pipe()` method, but with increased error handling support. Consider the following example,


```
// pipe2.js
"use strict";

const { pipeline, Transform } = require("stream");

class ToUpperCase extends Transform {
  constructor() {
    super();
  }

  _transform(chunk, encoding, callback) {
    this.push(chunk.toString().toUpperCase());
    callback();
  }
}

const upperCaser = new ToUpperCase();

pipeline(process.stdin, upperCaser, process.stdout, (err) => {
  if (err) {
    console.error("Pipeline failed.", err);
  } else {
    console.log("Pipeline succeeded.");
  }
});
```

This time, instead of using the `pipe()` method to chain the transform stream with the `process.stdin` and `process.stdout`, we use the `pipeline` utility function to achieve the same results. The final argument in the `pipeline` function is the callback. In our case we are using what is referred to as the `errBack`.

`errBack` or `callback` is a common pattern in Node.js for asynchronous operations. It is a function that you provide an argument to and it gets called when the operation completes. In our code example, the `errBack` function is called when the pipeline finishes. If there was an error at any point in the pipeline, the `err` argument would contain that error. Otherwise, the pipeline is successful. With the `errBack` approach, you can handle both successes and failures in a single chain of streams.

Let's take a look at another example of how `pipeline` can be used in a Node.js program.

1. Reverse Word Game:

In this example, we will create a program that reverses the input string using `pipeline`.

```

"use strict";

const { pipeline, Transform } = require("stream");

class ReverseString extends Transform {
  constructor() {
    super();
  }

  _transform(chunk, encoding, callback) {
    this.push(chunk.toString().split("").reverse().join(""));
    callback();
  }
}

const reverser = new ReverseString();

pipeline(process.stdin, reverser, process.stdout, (err) => {
  if (err) {
    console.error("Pipeline failed.", err);
  } else {
    console.log("Pipeline succeeded.");
  }
});

```

This code is similar to the EchoMan program we wrote earlier, but this time we are using the pipeline utility function to chain the streams together.

2. Superhero to Villain:

In this program, we will use the Transform stream class to replace the word superhero with the word villain.

```

const { pipeline, Transform } = require("stream");

class ReplaceWord extends Transform {
  constructor() {
    super();
  }

  _transform(chunk, encoding, callback) {
    const inputString = chunk.toString();
    const outputString = inputString.replace(/superhero/g, 'villain');
  }
}

```

```

    this.push(outputString);
    callback();
  }
}

const replacer = new ReplaceWord();

pipeline(process.stdin, replacer, process.stdout, (err) => {
  if (err) {
    console.error("Pipeline failed.", err);
  } else {
    console.log("Pipeline succeeded.");
  }
});

```

In the final section of this chapter, we will take a look at how we can use TDD principles to drive the development of a Node.js program that uses streams.

TDD and Streams

In this section we will take a look at how to use TDD principles to drive the development of two Node.js programs that use streams.

1. Jokes Stream Program:

In this example, you will be creating a Node.js program that uses readable and writable streams to tell jokes. The program will read jokes from a Readable stream and write them to a Writable stream. You will be using Jest to create a test suite for this program, following the principles of Test-Driven Development.

Project Structure:

```

jokes-streamer
  src
    jokes-streamer.js
  test
    jokes-streamer.spec.js
  package.json

```

Failing Test (Red):

```

"use strict";

const { JokesStream } = require("../src/jokes-streamer");

describe("JokesStream", () => {

```

```

test("should output jokes correctly", (done) => {
  const jokes = [
    "What do you call a fake noodle? An impasta!",
    "What do you call a bear with no teeth? A gummy bear!",
    "Why don't scientists trust atoms? Because they make up everything!"
  ];

  const jokesStream = new JokesStream(jokes.slice().reverse());

  let output = "";
  jokesStream.on("data", (chunk) => {
    output += chunk;
  });

  jokesStream.on("end", () => {
    expect(output).toBe(jokes.join("\n") + "\n");
    done();
  })
});

```

Solution (Green):

```

"use strict";

const { Readable } = require("stream");

class JokesStream extends Readable {
  constructor(jokes, options) {
    super(options);
    this.jokes = jokes;
  }

  _read() {
    if (this.jokes.length === 0) {
      this.push(null);
    } else {
      this.push(this.jokes.pop() + "\n");
    }
  }
}

module.exports = { JokesStream };

```

2. Movie Duplex Stream Program:

In this example, you will be creating a Node.js program that uses Duplex streams to process movie titles. The program will read movie titles from the input side of the Duplex and add “The move is: “ before each title, and then output the processed titles from the output side of the Duplex stream. Test-Driven Development will be used to drive the development of this program.

Project Structure:

```
movie-processor  
  src  
    movie-processor.js  
  test  
    movie-processor.spec.js  
  package.json
```

Failing Tests (Red):

```
// movie-processor.spec.js  
"use strict";  
  
const { MovieProcessor } = require("../src/movie-processor");  
  
describe("MovieProcessor", () => {  
  let movieProcessor;  
  
  beforeEach(() => {  
    movieProcessor = new MovieProcessor();  
  });  
  
  test("should process movie data correctly", (done) => {  
    const movies = ["The Matrix", "The Matrix Reloaded", "The Matrix  
Revolutions"];  
  
    movies.forEach(movie => movieProcessor.write(movie));  
  
    movieProcessor.on('data', (data) => {  
      const movie = movies.shift();  
      expect(data.toString().trim()).toBe(` The movie is titled "${movie}"`);  
      if (movies.length === 0) done();  
    });  
  });  
  
  test("should emit 'error' when invalid data is written", (done) => {  
    movieProcessor.on("error", (err) => {  
      expect(err.message).toBe("Invalid data");  
    });  
  });
```

```

    done();
  });

  movieProcessor.write("");
});

test("should transform data correctly when written to", (done) => {
  const movie = "The Matrix";
  const expectedOutput = `The movie is titled "${movie}".`;

  movieProcessor.write(movie, 'utf8', () => {
    movieProcessor.on('data', (data) => {
      expect(data.toString().trim()).toBe(expectedOutput);
      done();
    });
  });
});
});

```

Solution (Green):

```

// movie-processor.js
"use strict";

const { Duplex } = require("stream");

class MovieProcessor extends Duplex {
  constructor(options) {
    super(options);
    this.queue = [];
  }

  _write(chunk, encoding, callback) {
    const movie = chunk.toString().trim();
    if (!movie) {
      callback(new Error("Invalid data"));
      return;
    }

    this.queue.push(`The movie is titled "${movie}".`);
    callback();
  }

  _read(size) {

```

```

    if (this.queue.length) {
      this.push(this.queue.shift() + "\n");
    } else {
      this.push(null);
    }
  }
}
}

```

```

module.exports = { MovieProcessor };

```

In this example, for the sake of simplicity, all three tests were included in the initial Red step of the TDD cycle. However, in a real-world assignment or project, you would typically start with a single failing unit test. After making that test pass (the Green step), you would then refactor the code if necessary. This Red-Green-Refactor cycle would be repeated for each of the three-unit tests.

Programming Exercises

In this assignment, you will create a character creation system for a fantasy video game using Node.js's built-in stream module. You will use a Duplex stream to process character creation data and output a formatted character description. You will also write unit tests for your code using Jest, following TDD principles:

Instructions:

1. Create a new project folder named `fantasy-character-creation-stream`.
2. Inside the project folder, create the following structure:

```

fantasy-character-creation-stream
  src
    character-creator.js
  test
    character-creator.spec.js
  package.json

```

3. In your `package.json` file, add a test script. Remember to install Jest as a dev dependency.
4. In the `character-creator.js` file create a `CharacterCreator` class that extends the Duplex stream class from Node.js's built-in stream module. The class should process character creation data and output a formatted description.
5. The `CharacterCreator` class should take data about the character's class (Warrior, Mage, Route), gender (Male, Female, Other), and a fun fact about the character.

6. Write three-unit tests for your CharacterCreator class in the character-creator.spec.js file. Each test should cover a different aspect about the class's functionality:
 - a. Test 1: The CharacterCreator processes data correctly when written to.
 - b. Test 2: The CharacterCreator emits an "error" event when an empty string is written to it.
 - c. Test 3: The CharacterCreator transforms the data correctly when written to.
7. Follow TDD principles: write a failing test, make it pass, then refactor. Repeat this process for each unit test.

Grading

You will be awarded 20 points for each passing unit test, for a total of 60 points. If two tests pass, you will earn 40 points. If all three tests pass, you will earn the full 60 points.

Hints:

- Use the push method in your Duplex stream to output data.
- Remember to call callback in your _write method.
- Use the done function in your Jest tests to handle asynchronous code.
- Refer to the MoviesStream provided in this chapter for guidance.

Time Estimate

This assignment should take approximately 4-5 hours to complete.

Chapter 8. File System

Chapter Overview

In this chapter, we delve into the powerful File System (fs) module provided by Node.js. This module offers a rich set of methods for interacting with the file system, enabling you to read and write files, manage directories, and more. We will explore how to use these methods to perform common file operations, and how to work with directories. We'll also discuss how to use streams, a core concept in Node.js, in conjunction with the fs module for efficient file handling. Lastly, we will introduce how to apply TDD principles when working with the fs module. This chapter provides a comprehensive guide to mastering file system operations in Node.js, a crucial skill for any Node.js developer.

Learning Objectives

By the end of this chapter, you should be able to:

- List the differences between synchronous and asynchronous operations in the fs module.
- Demonstrate how to read and write data to a text file.
- Identify the benefits of using streams with large files.
- Build a Node.js program using the fs module.

Introduction to the File System (fs) Module

The fs module is a module that comes pre-installed with Node.js. It provides a way to interact with the file system, which is a method of storing and organizing data in a storage medium such as a hard drive, USB drive, or virtual storage device. Without a file system, data stored in these mediums would be a large “blob” of data, making it challenging to differentiate between them. The file system on a computer manages files, folders, and data, and everything is often stored in a hierarchical structure. The file system also includes and tracks attributes like file/folder permissions, and data modification dates.

In the context of Node.js, the fs module’s API allows users to interact with the file system and perform various operations like reading and writing files, creating directories, setting permissions, and watching for changes for files or folders. The file system provides three sets of methods for interacting with your file system. Each of these sets of methods provide similar behaviors, but in a different way. Here is an explanation of each set of methods.

1. **Synchronous API:** These methods block the Node.js event loop until the operations have completed. Results are returned directly and throw exceptions on errors. In most cases, they are suffixed with Sync. For example, `readFileSync`, `writeFileSync`, and `readdirSync`. Each of these methods are part of the fs module.
2. **Callback API:** These methods are non-blocking and take a callback function as the last parameter in the function. The callback is called when the operation has completed, either an error as the first argument and the data as the second. Examples of these are: `readFile`, `writeFile`, and `readdir`. Each of these methods are part of the fs module.
3. **Promise API:** These methods are non-blocking Promise-based variants. Similar to the callback API, they are a way for interacting with the file system through asynchronous execution. Examples of these are: `readFile`, `writeFile`, and `readdir`. The distinction is, to access these methods, you use `fs/promises`. That is, `fs.promises.readFile` instead of `fs.readFile`.

The most commonly used methods in each of these categories includes:

1. **Synchronous methods:**

- a. `readFileSync`: Reads the entire contents of a file synchronously.
- b. `writeFileSync`: Writes data to a file synchronously. If the file already exists, it is replaced.
- c. `existsSync`: Synchronously tests if the provided path actually exists.
- d. `readdirSync`: Reads the contents of a directory synchronously.
- e. `mkdirSync`: Creates a new directory synchronously.
- f. `statSync`: This method gets the status of a file or directory synchronously.

2. **Callback methods:**

- a. `createReadStream`: Creates a readable stream for file reading.
- b. `createWriteStream`: Creates a writable stream for file writing.
- c. `mkdir`: Creates a new directory asynchronously.
- d. `readFile`: Reads the contents of a file asynchronously.
- e. `writeFile`: Writes data to a file asynchronously.
- f. `exists`: Asynchronously tests if the provided path actually exists.
- g. `readdir`: Reads the contents of a directory asynchronously.
- h. `stat`: This method gets the status of a file or directory asynchronously.
- i. `watch`: This method watches for changes to either a file or directory.

3. **Promise methods:**

- a. `fs.promises.readFile`: This method reads the entire contents of a file asynchronously and returns a Promise.
- b. `fs.promises.writeFile`: This method writes data to a file asynchronously and returns a Promise.
- c. `fs.promises.exists`: This method tests if the provided path actually exists and returns a Promise.

- d. `fs.promises.readdir`: Reads the contents of a directory asynchronously and returns a Promise.
- e. `fs.promises.stat`: This method gets the status of a file or directory asynchronously and returns a Promise.

The key point here is, synchronous methods will block the rest of your code from executing until the process has finished. While, callback and promise-based methods will not. In general, you should follow these rules when deciding which set of methods to use.

1. **Synchronous:** These methods are simple to use because they do not require the use of callbacks or promises (resolve or reject). However, they do block the process. This means all other JavaScript code cannot be executed until the synchronous operation has completed. They are best used for one-time setup tasks before your application starts or for scripts that do not need to respond to user supplied data.
2. **Callbacks:** These methods are more complex because they require you to deal with the callback function. However, they do not block the process and they are great for I/O operations like reading from the file system, making network requests, or writing data to a file. Generally speaking, in a running application you do not want to block the event loop. Thus, using callbacks is a great approach to building non-blocking programs that interact with the file system.
3. **Promises:** These methods are also non-blocking and they return a Promise. A promise resolves once the operation has completed. They are a lot easier to work with when compared to callback functions. Especially when you are dealing with several asynchronous operations at once. For these methods you must use `async/await`. They are best used in the same scenarios as the callback methods, but when you also prefer to use the promise-based syntax of `async/await`.

A key point to remember here is, Node.js was designed to handle asynchronous operations and it excels at it. Blocking the event loop could be considered an anti-pattern and could eventually cause performance issues. Now, that is not to say, synchronous methods should not be used. In fact, there are many use cases where they are appropriate. The above rules are intended to be used as a guide to deciding when to use what methods. It is not all encompassing and you should always rely on performance and business requirements to drive the direction of your development.

In the next section, we will take a look at how to use these methods to read and write data to files.

Reading and Writing Files

In this section we will explore examples of how to read and write data to files using all three of the methods mentioned in the previous section. We start by taking a look at the `readFileSync`, `writeFileSync`, `readdirSync`, and `mkdirSync`.

1. `readFileSync`:

```
"use strict";

const { readFileSync } = require("fs");
const { join } = require("path");

const file = join(__dirname, "superheroes.txt");

const superheroes = readFileSync(file, "utf-8");

console.log(superheroes);
```

In this example, we import the `readFileSync` method from the `fs` module. Next, we import the `join` method from the `path` module. Then we use the `join` method to join the current directory name (`__dirname`) with the name of our text file. Then we call the `readFileSync` method, passing in the file and encoding type as arguments. Had we left the encoding type off, the `readFileSync` method would have returned a buffer that we would need to cast to a string using the `toString()` method. Finally, we use `console.log` to output the contents of the `superheroes` text file.

superheroes.txt:

```
Superman
Batman
Wonder Woman
Flash
Green Lantern
Aquaman
```

2. `writeFileSync`:

```
"use strict";

const { writeFileSync } = require("fs");
const { join } = require("path");

const file = "superheroes.json";
```

```

const superheroes = [
  "Superman",
  "Batman",
  "Wonder Woman",
  "Flash",
  "Green Lantern",
  "Aquaman"
]

const data = JSON.stringify(superheroes, null, 2);

writeFileSync(file, data, "utf-8");

console.log("Superheroes written to file.")

```

In this example, `writeFileSync` is used to write the JSON string data to “superheroes.json.” If the file already exists, it will be replaced. If it does not exist, it will be created. The method does not return any value. After the superheroes data is written to the file, a message is printed to the console. Encoding type “utf-8” is used to tell Node.js to interpret the data as a UTF-8 encoding string. Without this, Node.js will write the data as a raw buffer (more on this below). The second and third arguments in the `JSON.stringify` function are optional and control the formatting of the generated JSON string. The second argument (`null`) is used to ensure all properties of the object are included in the resulting JSON string. The third argument (`2` in this case) is a string or number that is used to insert white space into the outputted JSON string for readability. If this is a number, it indicates the number of space characters to use as white space. Technically, we do not need to include UTF-8 in the `writeFileSync` method, because `JSON.stringify` does this by default when you convert JSON to a string. However, by including this, we are clear in our intentions. This is a best practice and should be used whenever you are wanting to work with strings or utf-8 encoding.

Now, let’s take a look at how we can perform similar operations, but with the callback methods.

1. `readFile`:

```

"use strict";

const { readFile } = require("fs");
const { join } = require("path");

const file = join(__dirname, "superheroes.txt"); // __dirname is the current
directory

```

```

readFile(file, "utf-8", (err, data) => {
  if (err) {
    console.error("Error reading file:", err.message);
    return;
  }
  console.log(data);
})

```

In this example, `readFile` is used to read the contents of “superheroes.txt”. The “utf-8” argument specifies the character encoding type for this file. Without this, the data would be returned as a raw buffer. The function provides a callback that will be called when the operation finishes. If there was an error reading the file, it logs the error message. Otherwise, it logs the contents of the file. Optionally, you can replace “superheroes.txt” with the actual path to a file of your choosing.

2. `writeFile`:

```

"use strict";

const { writeFile } = require("fs");
const { join } = require("path");

const file = join(__dirname, "ingredients.json");

const ingredients = [
  "1 cup flour",
  "1 cup sugar",
  "4 eggs",
  "2 cups chocolate chips",
  "1 cup butter",
  "1 teaspoon baking soda",
  "1 teaspoon vanilla extract",
  "1 teaspoon salt",
  "1 cup brown sugar",
  "1/2 cup cocoa powder",
  "1 cup milk",
  "1 cup water"
];

const data = JSON.stringify(ingredients, null, 2);

writeFile(file, data, "utf-8", (err) => {
  if (err) {

```

```

    console.error("Error writing file:", err);
    return;
}

    console.log("Ingredients written to file.")
})

```

In this example, `writeFile` is used to write the JSON string to “`ingredients.json`”. If the file already exists, it will be replaced. If it does not exist, it will be created. The function provides a callback that will be called when the operation finishes. If there was an error while writing the file, it logs the error message. Otherwise, it logs a success message.

Now, let’s take a look at the promise version of these methods.

1. `readFile`:

```

"use strict";

const { readFile } = require("fs").promises;
const { join } = require("path");

const file = join(__dirname, "superheroes.txt");

async function readSuperheroes() {
  try {
    const data = await readFile(file, "utf8");
    console.log(data); // print file contents
  } catch (err) {
    console.error("Error reading file:", err);
  }
}

readSuperheroes();

```

In this example, the code reads the contents of the file named `superheroes.txt` using the `readFile` method from the `fs` module’s promise API. The `readFile` method is used within an asynchronous function named `readSuperheroes`. This function tries to read the file and logs its content to the console. If an error occurs during the operation, it is caught and an error message is logged to the console. The `readSuperheroes` function is then called to execute the read operation.

2. `writeFile`:

```

"use strict";

const { writeFile } = require("fs").promises;
const { join } = require("path");

const file = join(__dirname, "villains.txt")

const villains = [
  "Joker",
  "Lex Luthor",
  "Black Adam",
  "Deathstroke",
  "Darkseid"
];

async function writeVillains() {
  try {
    await writeFile(file, villains.join("\n"));
    console.log("Villains written successfully!");
  } catch (err) {
    console.error("Error writing villains!", err);
  }
}

writeVillains(); // call the function

```

In this example, `writeFile` is used to write the array of villains to “villains.txt”. The villains are joined into a string with each villain on a new line using `join("\n")`. If the file already exists, it will be replaced. If it does not exist it will be created. If there was an error during the write operation, the error message is logged to the console. Otherwise, it logs a success message.

In the next section we will take a look at how to work with directories using Node.js built-in `fs` module.

Working with Directories

In this section we will look at how to work with directories using Node.js’s built-in `fs` module. Similar to how we explored the previous section, we will look at all three method types. To start, let’s take a look at the synchronous methods: `mkdirSync`, `readdirSync`, and `existsSync`.

1. **mkdirSync:**

```
"use strict";

const { mkdirSync } = require("fs");

const dirName = "superhero-movies";

mkdirSync(dirName);

console.log("New directory created: " + dirName);
```

In this example, `mkdirSync` is used to create a new directory. If the directory already exists, an error will be thrown. After the directory is created, a message is logged to the console. Whenever you use `mkdirSync`, it is a good practice to wrap the code in a try/catch block. This will allow you to handle thrown errors (directory already exists). Optionally, you can replace the value for the `dirName` variable with the name of the directory you want to create.

2. **readdirSync:**

```
"use strict";

const { readdirSync } = require("fs");

const files = readdirSync(__dirname); // __dirname is the current directory

console.log("The files in this directory are:", files);
```

In this example, `readdirSync` is used to read the documents of the current directory (represented by `__dirname`). The method returns an array of filenames, which is then logged to the console. Optionally, you can replace `__dirname` with the actual path of the directory you want to read the files from.

3. **existsSync:**

```
"use strict";

const { existsSync } = require("fs");
const { join } = require("path");

const file = join(__dirname, "superheroes.txt");

if (existsSync(file)) {
```

```
    console.log("The file exists!", file);
  } else {
    console.log("The file does not exist!", file);
  }
}
```

In this example, `existsSync` is used to synchronously check if a file named `superheroes.txt` exists in the same directory as the script. If the file exists, it logs, “The file exists!” to the console. If the file does not exist, it logs “The file does not exist!” to the console.

Now, let’s take a look at the callback versions of `readdir`, `watch`, and `stat`.

1. `readdir`:

```
"use strict";

const { readdir } = require("fs");

const dir = __dirname;

readdir(dir, "utf-8", (err, files) => {
  if (err) {
    console.error("Error reading directory:", err);
    return;
  }

  console.log("Files in directory:", files)
})
```

In this example, `readdir` is used to read the contents of the current directory (represented by `__dirname`). The function provides a callback that will be called when the operation finishes. If there was an error reading the directory, it logs the error message. Otherwise, it logs the filenames in the directory. Optionally, you can replace `__dirname` with the actual path to the directory you want to read.

2. `watch`:

```
"use strict";

const { watch } = require("fs");
const dir = __dirname; // current directory

const watcher = watch(dir, (evt, filename) => {
  if (filename) {
```

```

    console.log(` File ${filename} changed.` );
  } else {
    console.log("File changed.");
  }
})

setTimeout(() => {
  watcher.close();
  console.log("Stopped watching.");
}, 60000); // stop watching after 60 seconds

```

In this example, watch is used to monitor the current directory (represented by `__dirname`). The function provides a callback that will be called when a file in the directory is changed. The callback receives two arguments: the type of event (renamed or changed) and the name of the file that changed. After 60 seconds, it stops watching the directory. To test this program, start it and begin adding or modifying files in the program's current directory. The program should output the changes.

3. **stat:**

```

"use strict";

const { join } = require("path");
const { readdir, stat } = require("fs");

const dir = __dirname; // current directory

readdir(dir, (err, items) => {
  if (err) {
    console.error("Error reading directory:", err);
    return;
  }

  items.forEach((item) => {
    const itemPath = join(dir, item);
    stat(itemPath, (err, stats) => {
      if (err) {
        console.error("Error getting stats for item:", err);
        return;
      }

      if (stats.isFile()) {
        console.log("File:", item);
      }
    });
  });
});

```

```

    }
  });
}
});

```

In this example, `readdir` is used to read the contents of the current directory (represented by `__dirname`). For each item in the directory, `stat` is used to get the item's stats. If the item is a file (not a directory), its name is logged to the console.

Lastly, let's take a look at the Promise versions of `readdir` and `stat`.

1. `readdir`:

```

"use strict";

const { readdir } = require("fs").promises;

const dir = __dirname;

async function readDir() {
  try {
    const files = await readdir(dir);
    console.log("Files in directory:", files)
  } catch (err) {
    console.error("Error reading directory!", err);
  }
}

readDir(); // call the function

```

In this example, `readdir` is used to read the contents of the current directory (represented by `__dirname`). The function `readDir` is asynchronous and waits for the Promise from `readdir` to resolve with the filenames in the directory. If there was an error reading the directory, it logs the error message. Otherwise, it logs the file name. Optionally, you can replace `__dirname` with a path to the directory you want to read.

2. `stat`:

```

"use strict";

const { join } = require("path");
const { readdir, stat } = require("fs").promises;

```

```

const dir = __dirname;

async function main() {
  try {
    const items = await readdir(dir);

    for (const item of items) {
      const itemPath = join(dir, item);
      const stats = await stat(itemPath);

      if (stats.isFile()) {
        console.log("File:", item);
      }
    }
  } catch (err) {
    console.log("An error occurred!", err);
  }
}

main();

```

In this version of the code, `readdir` and `stat` are used with `await` to pause the execution of the main function until the Promise resolves. If an error occurs at any point, it is caught and logged to the console.

In the next section, we will take a look at how we can use streams with a file system.

Streams and the File System

In this section we will explore two built-in methods of the callback methods in the `fs` module. Specifically, `createReadStream` and `createWriteStream` and how we can use them with stream classes, like the Transform stream.

1. `createReadStream`:

```

const { createReadStream } = require('fs');
const { join } = require('path');

const file = join(__dirname, "superheroes.txt"); // superheroes.txt is in the same
directory as this file

const readStream = createReadStream(file, { encoding: 'utf8' });

readStream.on("data", (chunk) => {

```

```

    console.log(chunk);
  });

  readStream.on("error", (err) => {
    console.error("An error occurred while reading the file: ", err);
  });

```

In this example, `createReadStream` is used to create a readable stream from the `superheroes.txt` file. The `on("data")` event listener logs each chunk of data as it is read from the file. The `on("error")` event listener logs any errors that occur while reading the file.

2. `createWriteStream`:

```

const { createWriteStream } = require('fs');
const { join } = require('path');

const file = join(__dirname, "marvel_superheroes.txt"); //
// marvel_superheroes.txt is in the same directory as this file

const stream = createWriteStream(file, { encoding: 'utf8' });

stream.write("Iron Man\n");
stream.write("Spider-Man\n");
stream.write("Captain America\n");
stream.write("Thor\n");
stream.write("Black Widow\n");
stream.write("Hulk\n");

stream.end(); // close the stream

stream.on("finish", () => {
  console.log("Finished writing Marvel superheroes to file."); // this will be
  // printed last
});

stream.on("error", (err) => {
  console.error("An error occurred while writing to the file: ", err);
});

```

In this example, `createWriteStream` is used to create a writable stream to the `"marvel_superheroes.txt"` file. The `write` method is used to write data to the file. The `end` method is called when no more data will be written to the stream. The `on("finish")` event listener logs a message when all data has been flushed to the

underlying system after the “end” event. The on(“error”) event listener logs any errors that occur while writing to the file.

3. Read – Transform – Write:

```
"use strict";

const { createReadStream, createWriteStream } = require("fs");
const { Transform } = require("stream");
const { pipeline } = require('stream/promises');
const { join } = require("path");

const translations = {
  "hello": "hola",
  "world": "mundo",
  "programming": "programar",
  "is": "es",
  "fun": "divertido"
};

const transformStream = new Transform({
  transform(chunk, encoding, callback) {
    const transformedChunk = chunk
      .toString()
      .split(' ')
      .map(word => translations[word] || word)
      .join(' ');

    callback(null, transformedChunk);
  }
});

async function translateFile() {
  const readStream = createReadStream(join(__dirname, "english.txt"));
  const writeStream = createWriteStream(join(__dirname, "spanish.txt"));

  try {
    await pipeline(readStream, transformStream, writeStream);
    console.log('File has been translated.');
```

```
  } catch (err) {
    console.error('An error occurred:', err);
  }
}
```

translateFile();

In this example, the Transform stream splits each chunk into words, translates each word to Spanish if a translation exists, and joins the words back together. The pipeline function is used to pipe the data from the read stream, through the transform stream, to the write stream. If an error occurs at any point, it is caught and logged to the console.

In the final section of this chapter, we will take a look at how we can use TDD principles to build Node.js programs that use Node.js's built-in fs module.

TDD with Reading and Writing Files

In this section, we will take a look at how we can use TDD principles to build and test Node.js programs that use the fs module. We start this section by building a program that uses the callback method approach.

1. Trivia Game (using callbacks)

You are going to build a trivia game for 10th grade students. The game will read questions from a file, and then write the student's answers to another file. We will use the readFile and writeFile functions from Node.js's fs module, which use callbacks. For the development of this project, we will use TDD principles (Red – Green – Refactor).

Project Structure

```
trivia-game
├── src
│   └── trivia-game.js
├── test
│   └── trivia-game.spec.js
└── package.json
```

Test (Red):

```
// trivia-game.spec.js
"use strict";

const fs = require('fs');

describe("Trivia Game", () => {
  let readTriviaQuestions;
  let writeStudentAnswers;

  beforeEach(() => {
```



```

    jest.resetModules();
    jest.spyOn(fs, 'readFile').mockImplementation((file, options, callback) =>
    callback(null, "Question 1\nQuestion 2\nQuestion 3\n"));
    jest.spyOn(fs, 'writeFile').mockImplementation((file, data, callback) =>
    callback(null));
    ({ readTriviaQuestions, writeStudentAnswers } = require('./src/trivia-game'));
  });

  test("reads trivia questions from a file", (done) => {
    readTriviaQuestions((err, questions) => {
      expect(err).toBeNull();
      expect(questions).toEqual(["Question 1", "Question 2", "Question 3"]);
      done();
    })
  });

  test("writes student answers to a file", (done) => {
    writeStudentAnswers(["Answer 1", "Answer 2", "Answer 3"], (err) => {
      expect(err).toBeNull();
      done();
    });
  });

  test("handles errors when reading trivia questions", (done) => {
    fs.readFile.mockImplementationOnce((file, options, callback) =>
    callback(new Error("File not found")));

    readTriviaQuestions((err) => {
      expect(err).not.toBeNull();
      expect(err.message).toBe("File not found");
      done();
    });
  });

```

Write Implementation (Green):
"use strict";

```

const { readFile, writeFile } = require('fs');
const { join } = require('path');

```

```

const TRIVIA_QUESTIONS_FILE = join(__dirname, "questions.txt"); // __dirname
is the directory of the current file
const STUDENT_ANSWERS_FILE = join(__dirname, "answers.txt");

```

```

function readTriviaQuestions(callback) {
  readFile(TRIVIA_QUESTIONS_FILE, { encoding: "utf8" }, (err, data) => {
    if (err) {
      callback(err);
    } else {
      const questions = data.split("\n").filter(question => question);
      callback(null, questions); // null is the first argument because there is no
error
    }
  });
}

```

```

function writeStudentAnswers(answers, callback) {
  const data = answers.join("\n");

  writeFile(STUDENT_ANSWERS_FILE, data, (err) => {
    callback(err);
  });
}

```

```

module.exports = { readTriviaQuestions, writeStudentAnswers };

```

Note: for sake of simplicity, all tests and solution were included in a single (Red – Green – Refactor) cycle. If you were developing this project on your own, you would write the first unit test to fail, write the code to pass it, and then repeat the process for the remaining tests.

2. Video Game (using Promises)

You are going to build a program that reads a list of video games from a file, and then writes a list of your favorite games to another file. We will use the `readFile` and `writeFile` functions from Node.js's `fs` module, which returns promises.

Project Structure

```

video-game-favorites
  src
    video-game-favorites.js
  test
    video-game-favorites.spec.js
  package.json

```

Failing Tests (Red):

```

"use strict";

```

```

const fs = require('fs').promises;

describe("Video Game Favorites", () => {
  let readFavoriteGames;
  let writePlayerRatings;

  beforeEach(() => {
    jest.resetModules();
    jest.spyOn(fs, 'readFile').mockImplementation(() => Promise.resolve("Game
1\nGame 2\nGame 3\n"));
    jest.spyOn(fs, 'writeFile').mockImplementation(() => Promise.resolve());
    ({ readFavoriteGames, writePlayerRatings } = require('../src/video-game-
favorites'));
  });

  test("reads favorite games from a file", async () => {
    const games = await readFavoriteGames();
    expect(games).toEqual(["Game 1", "Game 2", "Game 3"]);
  });

  test("writes player ratings to a file", async () => {
    await expect(writePlayerRatings(["Rating 1", "Rating 2", "Rating
3"])).resolves.toBeUndefined();
  });

  test("handles errors when reading favorite games", async () => {
    fs.readFile.mockImplementationOnce(() => Promise.reject(new Error("File not
found")));

    await expect(readFavoriteGames()).rejects.toThrow("File not found");
  });
});

```

Implementation (Green):
"use strict";

```

const { readFile, writeFile } = require("fs").promises;
const { join } = require("path");
const GAMES_FILE = join(__dirname, "games.txt");
const RATINGS_FILE = join(__dirname, "ratings.txt");

async function readFavoriteGames() {
  try {

```

```

    const data = await readFile(GAMES_FILE, "utf8");
    const games = data.split("\n").filter(game => game);
    return games;
  } catch (err) {
    throw err;
  }
}

```

```

async function writePlayerRatings(ratings) {
  try {
    const data = ratings.join("\n");
    await writeFile(RATINGS_FILE, data);
  } catch (err) {
    throw err;
  }
}

```

```

module.exports = { readFavoriteGames, writePlayerRatings };

```

Note: for sake of simplicity, all tests and solution were included in a single (Red – Green – Refactor) cycle. If you were developing this project on your own, you would write the first unit test to fail, write the code to pass it, and then repeat the process for the remaining tests.

Programming Exercises

In this assignment you will create a character creation system for a fantasy video game. You will use TDD principles with Jest and Node.js's built-in fs module. You can choose to use either the callback methods or promise methods (async/await).

Requirements:

1. **Project Structure:** Your project should have the following structure:

```

fantasy-character-creation
  src
    character-creation.js
  test
    character-creation.spec.js
  package.json

```

2. **TDD:** You must follow TDD principles. Write your tests first, then write the code to make the tests pass.
3. **Unit Tests:** You must write at least three-unit tests:

- a. Test that data can be written to a file.
 - b. Test that data can be read from a file
 - c. Test that it handles errors when reading from the file
4. **File Operations:** You must use `__dirname`, `readFile`, and `writeFile`. And, you must use `const` variables for each file name.
5. **Character Creation:** Your program should allow the creation of a character with the following properties:
 - a. Class (Warrior, Mage, Rogue)
 - b. Gender (Male, Female, Other)
 - c. An additional property for something neat and fun about your character.
6. **File Data:** Your program should write character data to a file and read character data from the file.
7. **Package.json:** Add a test script to the `package.json` file to run your tests with Jest

Instructions:

1. Set up your project structure as describe in the Requirements section.
2. Write your first test in `character-creation.spec.js`. This test should check that data can be written to a file.
3. Run your test script to confirm that the test fails.
4. Write code in the `character-creation.js` to make the test pass.
5. Run your test script again to confirm that the test now passes.
6. Repeat steps 2-5 for the remaining two tests.
7. Make sure to handle errors properly in your code and tests.

Grading:

You will be awarded 20 points for each passing unit test, for a total of 60 points. If two tests pass, you will earn 40 points. If all three tests pass, you will earn the full 60 points.

Hints:

- Use the `fs` module's `readFile` and `writeFile` functions to read from and write to files.
- Use `__dirname` to get the directory name of the current module.
- Use `const` variables for each file name.

- Remember to follow TDD principles: write your test first, then write code to make the test pass.
- Refer to the examples in this chapter to complete this assignment.

Chapter 9. Child Processes

Chapter Overview

In this chapter, we delve into the world of child processes in Node.js, a powerful feature that allows Node.js to run system commands within the context of your application. This capability opens up a new dimension of possibilities, enabling your Node.js application to interact more deeply with the underlying system. We will explore how to spawn child processes, execute shell commands, handle errors, and even test code that uses child processes. By the end of this chapter, you will have a solid understanding of how to leverage child processes in Node.js to extend the functionality of your application beyond the confines of the JavaScript runtime.

Learning Objectives

By the end of this chapter, you should be able to:

- Define what child processes are and why they are used.
- Explain how errors are handled in a child process.
- Identify the differences between `spawn()` and `exec()`.
- Build a Node.js program using child processes.

Introduction to Child Processes in Node.js

When we talk about software development, a child process refers to a process that has been initiated by another process. The process that starts this other process is known as the parent process, while the process that has been launched is referred to as the child process. This approach helps facilitate the concurrent execution of processes, which is a fundamental aspect of any operating system.

In the case of Node.js, the single-threaded nature of this platform makes it necessary to leverage multicores on a system. One way to achieve this is by using child processes. By using child processes, Node.js can handle memory and CPU-intensive operations that would otherwise block the single thread nature of Node.js. This way, applications can be more responsive.

Understanding how to utilize child processes in a Node.js application is important for the following reasons:

1. **Performance:** Node.js is a single-threaded environment. This means it cannot natively work with a multicore system. However, through child processes, you can

build processes that run multiple instances of your application in parallel. Thus, you are effectively utilizing the multicore nature of a system and significantly improving the performance of your application.

2. **Non-blocking operations:** CPU-intensive tasks will block Node.js's single-threaded environment, which causes your application to become unresponsive. By leveraging child processes, we can off load some of these labor-intensive operations to a child process. This will keep your main application responsive while the child process handles the task.
3. **Running system commands:** By using child processes in your Node.js program you can establish a deeper interaction with the underlying system. This allows you to run system commands, execute other scripts or applications, and communicate with other processes.
4. **Complex workflows:** You can chain child processes together. This will allow you to create complex workflows in your application.
5. **Testing and isolation:** Child processes are often used to isolate parts of your application for testing or to protect the main process from crashing.

To grasp the nature of how child processes work in a Node.js program, consider the following analogy. Imagine your computer is like the city of Gotham and your Node.js application is Batman. I think we all can agree that Batman (Node.js) is pretty amazing. He can do a lot of things on his own. But there are times when he needs help from one of his allies to keep the city safe. In these situations, he calls upon his allies like Robin, Batgirl, and Commissioner Gordon. His allies are like child processes.

When Batman (Node.js) needs to do something that is too big for him to handle on his own, he can ask his allies to take on the task (spawn a child process). For example, he might ask Robin (a child process) to patrol the city (perform a task) while he investigates a crime scene (continues executing other tasks).

Just like how Batman can communicate with Commissioner Gordon using the Batphone, Node.js can communicate with these child processes. I can send instructions to a child process and a child process can respond back with updates or the results of some operation.

In the Node.js world, these allies (child processes) can be other Node.js processes or even other programs on a computer. By using child processes, Node.js (Batman) can do more things at once, effectively using all of the available resources on the computer. However, just like Batman has many ways to contact his allies (Batphone, Bat-Signal, radio, etc.), Node.js has different functions to create child processes. Similar to what we explored in Node.js's `fs` module, the `child_process` module provides both synchronous and

asynchronous functions for creating child processes. The most commonly used functions are:

Asynchronous Functions:

1. **spawn()**: This function launches a new process with a given command and returns a stream. Typically, you will use the spawn() function when the process you are creating returns a large amount of data.
2. **exec()**: This function is similar to the spawn() function, but it buffers the output and it uses a callback function.
3. **fork()**: This is a special version of the spawn() function, which has the ability to pass messages between parent and child processes.

Synchronous Functions:

1. **spawnSync()**: This is a synchronous version of the spawn() function. However, it does block the event loop until the spawned process has completed.
2. **execSync()**: This is the synchronous version of the exec() function, it buffers the output and returns the whole output value when the child process is complete.

In the next section we will take a look at how we can use these functions to spawn child processes.

Spawning Child Processes

In this section we explore how to spawn child processes using both synchronous and asynchronous functions from the `child_process` module. We will start by looking at the synchronous functions: `execSync()` and `spawnSync()`.

Synchronous:

1. **execSync()**:

In this example, we will use the `execSync()` function to execute a Node.js script that prints a list of US presidents. First, create a script named `presidents.js`:

```
"use strict";

const presidents = ["George Washington", "John Adams", "Thomas Jefferson"];

presidents.forEach((president) => {
  console.log("President:", president);
});
```


This script defines an array of US presidents and calls JavaScript's built-in `forEach()` function to print each president to the console. Try running this script to verify that it is running correctly.

Next, create another script and name it `execsync-example.js`:

```
"use strict";  
  
const { execSync } = require("child_process");  
  
const output = execSync("node presidents.js");  
  
console.log(output.toString());
```

Now, try running this script to verify that it calls and prints the contents of the `presidents.js` script.

2. `spawnSync()`:

In this example, we will use the `spawnSync()` function to execute a Node.js script that prints a list of DC superheroes. First, create a new script named `superheroes.js`:

```
"use strict";  
  
const superheroes = ["Batman", "Superman", "Wonder Woman", "Cyborg",  
"Aquaman", "Flash"];  
  
superheroes.forEach((superhero) => {  
  console.log("Superhero:", superhero);  
});
```

Try running this script to verify that it prints a list of DC superheroes. Next, create another script and name it `spawnsync-example.js`:

```
"use strict";  
  
const { spawnSync } = require("child_process");  
  
const child = spawnSync("node", ["superheroes.js"]);  
  
console.log(child.stdout.toString());
```

Now, try running this script to verify that it calls and prints the contents of the `superheroes.js` script.

There are a couple things to point out here. First, the `spawnSync` function takes two arguments:

- a. The first argument is the command to run. In the case of the provided example, it's `"node"`.
- b. The second argument is an array of strings that represent the command line arguments to pass to the new process. In the case of our example, we are using `"superheroes"`, which means the entire script will be `node superheroes.js`.

Second, the `stdout` property of the child object contains the output of the child process and `toString()` is used to convert the returned buffer into a string.

Finally, here are a few examples of some command-line arguments that can be used with the `spawnSync` function in `Node.js`

- a. Running a Node.js script: **`spawnSync("node", ["script.js"]);`**
- b. List files in a directory using the `ls` command (Unix-based systems):
`spawnSync("ls", ["-l", "/path/to/directory"]);`
- c. Use the `git` command to clone a repository: **`spawnSync("git", ["clone", "https://github.com/user/repo.git"]);`**
- d. Using `npm` command to install a package: **`spawnSync("npm", ["install", "express"]);`**

Next, let's take a look at the asynchronous functions: `spawn()`, `exec()`, and `fork()`.

1. `spawn()`:

In this example, we will use the `spawn()` function to execute a Node.js script that prints a list of cake ingredients. First, create a script named `ingredients.js`:

```
"use strict";
```

```
const ingredients = ["flour", "sugar", "eggs", "butter", "chocolate"];
```

```
ingredients.forEach((ingredient) => {  
  console.log("Ingredient:", ingredient);  
});
```

```
});
```

Next, create a file named spawn-example.js:

```
"use strict";

const { spawn } = require("child_process");

const child = spawn("node", ["ingredients.js"]);

child.stdout.on("data", (data) => {
  console.log(data.toString());
});
```

Now, try running this script to see the results. A list of ingredients should be printed to the console.

2. `exec()`:

In this example, we will use the `exec()` function to execute a Node.js script that prints a programmer joke. First, create a new script named joke.js:

```
"use strict";

console.log("Why don't programmers like nature? It has too many bugs.");
```

Next, create a script named exec-example.js:

```
"use strict";

const { exec } = require("child_process");

exec("node joke.js", (err, stdout, stderr) => {
  if (err) {
    console.error("exec error:", err);
    return;
  }

  console.log("stdout:", stdout);
})
```

Now, try running this script to see the results. A joke with the punchline should be printed to the console.

3. **fork():**

In this example, we will create a script that forks a new process for each superhero, and each child process will print its superhero's name. First create a script named **superhero.js**:

```
"use strict";

process.on("message", (hero) => {
  console.log("This process will handle:", hero);
});

process.send("done");
```

Next, create a script named **fork-example.js**:

```
"use strict";

const { fork } = require("child_process");

const superheroes = ["Batman", "Superman", "Wonder Woman"];

superheroes.forEach((hero) => {
  const child = fork("superhero.js"); // fork a new process
  child.send(hero); // send the hero to the child process

  child.on("message", (message) => {
    console.log("Child process for " + hero + " is " + message);
  });
});
```

The main script creates a new process for each superhero. It sends the superhero's name to the child process, and the child process prints the superhero's name. When the child process is done, it sends a message back to the parent process, which prints a message indicating that the child process has completed.

Try running this script to see the outputted results.

In the next section we will take a look at how we can use child processes to execute shell commands.

Executing Shell Commands with Child Processes

In this section we will take a look at how we can execute shell commands using child processes. Shell commands are instructions that you can type in the terminal window to perform OS level tasks. These tasks include navigating files system, installing software, starting or stopping a service, and managing system settings. The shell is a program that reads these commands (similar to you building a CLI command) and executes them.

As a Node.js developer, there are several reasons why you should care about shell commands:

1. **Automation:** Shell commands can be used to automate repetitive tasks, such as deploying a Node.js app. This is done by creating a script containing shell commands that can be executed automatically.
2. **Interacting with the System:** The `child_process` module in Node.js allows the execution of shell commands from within the Node.js code. This is useful for tasks that are handled more efficiently by the system's shell than by Node.js. For instance, the `wkhtmltopdf` command line tool can be used to convert an HTML template to a PDF report. By executing this tool as a child process, we avoid the complicated task of writing the code to implement this functionality in pure Node.js, which would involve parsing HTML, applying CSS styles, laying out the page, and generating the PDF. Instead, we can simply call “**wkhtmltopdf filename.html filename.pdf**” and accomplish the task with ease.
3. **Package Management:** Node.js uses npm, which as you learned is a command-line tool that allows you to install and manage Node.js packages. Understanding shell commands will make working with npm a lot easier.
4. **Development Environment Setup:** Setting up an environment for development typically requires using shell commands to install software, configure environment variables, and manage files and directory permissions.
5. **Debugging and Troubleshooting:** When your application breaks (things go wrong), being able to use shell commands to figure out the state of your system can be very helpful.

Let's take a look at some Node.js programs that use `spawnSync` and `spawn` to execute shell commands. The programs will leverage Node.js's built-in `process` object to execute shell commands that are universally accepted on Windows, macOS, and Linux.

spawnSync:

1. Using `echo` to print a message:

```
"use strict";
```

```
const { spawnSync } = require("child_process");
```

```
const child = spawnSync("echo", ["hola", "mundo", "programar", "es",  
"divertido"], { encoding: "utf8" });
```

```
console.log("stdout:", child.stdout.toString());
```

2. Using dir or ls to list directory contents:

```
"use strict";
```

```
const { spawnSync } = require("child_process");
```

```
const command = process.platform === "win32" ? "dir" : "ls";  
const child = spawnSync(command); // no arguments  
console.log("stdout:", child.stdout.toString());
```

spawn:

1. Using ping to check network connectivity:

```
"use strict";
```

```
const { spawn } = require("child_process");
```

```
const child = spawn("ping", ["-c", "4", "www.google.com"]);
```

```
child.stdout.on("data", (data) => {  
  console.log("stdout:", data.toString());  
});
```

2. Using date or time to print the current date/time:

```
"use strict";
```

```
const { spawn } = require("child_process");
```

```
const command = process.platform === "win32" ? "time /t" : "date";  
const child = spawn(command); // no arguments
```

```
child.stdout.on("data", (data) => {  
  console.log("stdout:", data.toString());  
});
```

```
}}
```

In the next section, we will take a look how to use error handling in child processes.

Error Handling in Child Processes

In this section we will take a look at how to work with error handling when spawning child processes. To do this, we will take a look at error handling using `execSync`, `spawnSync`, `exec`, and `spawn`.

Synchronous:

1. **`execSync`:**

```
"use strict";

const { execSync } = require("child_process");

try {
  const stdout = execSync("echo 'Flour, Sugar, Eggs, Butter'"); // stdout is a
  Buffer
  console.log("stdout:", stdout.toString());
} catch (err) {
  console.error("stderr:", err.stderr.toString());
}
```

2. **`spawnSync`:**

```
"use strict";

const { spawnSync } = require("child_process");

const child = spawnSync("echo", ["Superman, Batman, Wonder Woman"]);

if (child.error) {
  console.error("Error:", child.error);
} else {
  console.log("stdout:", child.stdout.toString());
}
```

Asynchronous:

1. **`exec`:**

```
"use strict";
```

```

const { exec } = require("child_process");

exec("echo 'Flour, Sugar, Eggs, Butter'", (err, stdout, stderr) => {
  if (err) {
    console.error("Error:", err);
    return;
  }

  if (stderr) {
    console.error("stderr:", stderr);
    return;
  }
  console.log("stdout:", stdout);
});

```

2. spawn:

```

"use strict";

const { spawn } = require("child_process");

const child = spawn("echo", ["Superman, Batman, Wonder Woman"]);

child.on("error", (err) => {
  console.error("Error:", err);
})

child.stdout.on("data", (data) => {
  console.log("stdout:", data.toString());
});

child.stderr.on("data", (data) => {
  console.error("stderr:", data)
});

```

In the final section of this chapter, we will take a look at how we can use TDD principles to build a Node.js program that works with child_processes.

TDD with Child Processes

In this section, we will take a look at how to use TDD principles to build Node.js programs that uses child processes. We will start this section by looking at a program that uses spawnSync with error handling to execute a Node.js script.

Trivia Game

In this example, you will be creating a fun trivia game that tests your knowledge of Node.js topics. The game will ask a question and expect an answer in return. You will use the `child_process` module's `spawnSync` function to call a separate Node.js script that checks if the provided answer is correct. The main program should handle any errors that occur during the process. You will follow the TDD principles: write failing tests first (Red), make them pass (Green), and then refactor your code.

Project Structure

```
trivia-game
├── src
│   ├── trivia-game.js
│   └── answer-checker.js
├── test
│   └── trivia-game.spec.js
└── package.json
```

1. Child Script (`answer-checker.js`):

```
"use strict";
```

```
const { question, answer } = message;  
let correctAnswer;
```

```
switch(question) {  
  case "What is Node.js?":  
    correctAnswer = "JavaScript runtime";  
    break;  
  default:  
    correctAnswer = "";  
}
```

```
if (answer === correctAnswer) {  
  process.send({result: "Correct!"});  
} else {  
  process.send({result: "Incorrect"})  
}
```

2. Write failing tests (Red):

```
// trivia-game.spec.js  
const { TriviaGame } = require('../src/trivia-game');  
const child_process = require('child_process');  
const { spawnSync } = child_process;
```

```

jest.mock('child_process');

describe("Trivia Game", () => {
  let triviaGame;

  beforeEach(() => {
    triviaGame = new TriviaGame();
    spawnSync.mockClear();
  });

  test("should correctly answer a trivia question", () => {
    const mockChild = { stdout: Buffer.from("Correct"), error: null };
    spawnSync.mockReturnValueOnce(mockChild);
    const result = triviaGame.answerQuestion("What is Node.js?", "JavaScript
runtime")
    expect(result).toBe(true);
  });

  test("should incorrectly answer a trivia question", () => {
    const mockChild = { stdout: Buffer.from("Incorrect"), error: null };
    spawnSync.mockReturnValueOnce(mockChild);
    const result = triviaGame.answerQuestion("What is Node.js?", "A
programming language")
    expect(result).toBe(false);
  });

  test("should handle an error when the trivia game is not available", () => {
    spawnSync.mockImplementationOnce(() => {
      throw new Error("Trivia game not found");
    });

    expect(() => triviaGame.answerQuestion("What is Node.js", "JavaScript
runtime")).toThrow("Trivia game not found");
  });
});

```

3. Write passing tests (Green):

```

// trivia-game.js
"use strict";

const { spawnSync } = require('child_process');

class TriviaGame {
  answerQuestion(question, answer) {

```

```

    const child = spawnSync("node", ["answer-checker.js"], { input:
JSON.stringify({ question, answer }) });

    if (child.error) {
      throw child.error;
    }

    const result = child.stdout.toString();

    return result === "Correct";
  }
}

module.exports = { TriviaGame };

```

4. Refactor

Video Game:

In this project, you will be creating a program that simulates playing a video game. The program will send a command to a separate Node.js script using the `child_process` module's `spawn` function, and the script will send back the game's state. The main program should handle any errors that occur during this process. You will follow TDD principles: write failing tests first (Red), make them pass (Green), and then refactor your code.

Project Structure

```

video-game
  src
    game.js
    video-game.js
  test
    video-game.spec.js
  package.json

```

1. Child Script (game.js):

```

"use strict";

process.on("message", (command) => {
  let state;

  switch(command) {
    case "start":
      state = "Game started";

```

```

    break;
  case "play":
    state = "Playing game";
    break;
  case "stop":
    state = "Game stopped";
    break;
  default:
    state = "Unknown command";
  }

  process.send({ state });
});

```

2. Write failing tests (Red):

```

"use strict";

const { VideoGame } = require("../src/video-game");
const child_process = require("child_process");
const { spawn } = child_process;

jest.mock("child_process");

describe("Video Game", () => {
  let videoGame;

  beforeEach(() => {
    videoGame = new VideoGame();
    spawn.mockClear();
  });

  test("should correctly send a command to the video game", (done) => {
    const mockChild = {
      stdout: { on: jest.fn((event, callback) => callback(JSON.stringify({ state:
"Game started" }))) },
      on: jest.fn(),
      send: jest.fn(),
    };
    spawn.mockReturnValueOnce(mockChild);

    videoGame.sendCommand("start", (state) => {
      expect(state).toBe("Game started");
      done();
    });
  });
});

```

```

    });
  });

  test("should handle an error when the video game is not available", () => {
    spawn.mockImplementationOnce(() => {
      throw new Error("Video game not found");
    });

    expect(() => videoGame.sendCommand("start")).toThrow("Video game not found");
  });

  test("should handle an unknown command", (done) => {
    const mockChild = {
      stdout: { on: jest.fn((event, callback) => callback(JSON.stringify({ state: "Unknown command" }))) },
      on: jest.fn(),
      send: jest.fn(),
    };
    spawn.mockReturnValueOnce(mockChild);

    videoGame.sendCommand("unknown", (state) => {
      expect(state).toBe("Unknown command");
      done();
    });
  });
});

```

3. Make the tests pass (Green):

```

"use strict";

const { spawn } = require("child_process");

class VideoGame {
  sendCommand(command, callback) {
    const child = spawn("node", ["game.js"]);

    child.on("error", (error) => {
      throw error;
    });

    child.stdout.on("data", (data) => {
      const { state } = JSON.parse(data.toString());
    });
  }
}

```

```

        callback(state);
    });

    child.send({ command })
  }
}

module.exports = { VideoGame };

```

4. Refactor

Comic Book Library

In this project, you will be building a ComicBooks class that retrieves and processes data from a separate JavaScript file using Node.js child processes. The ComicBooks class will have a getComicBooks method that spawns a child process to run a script, captures the data it sends back, and passes the data to a callback function.

You will also need to handle any errors that might occur during this process. If there is an error while spawning the child process, it should be logged to the console. If the child process sends any data to stderr, this should also be logged to the console and passed to the callback function. We will use TDD principles to drive the development of this project.

Project Structure

```

comic-book-library
  src
    comic-books-data.js
    comic-books.js
    failing-script.js
  test
    comic-books.spec.js
  package.json

```

1. comic-books-data.js

```

"use strict";

const comicBooks = [
  { Name: "The Dark Knight Returns", Publisher: "DC Comics", Superhero:
    "Batman", Villain: "The Joker" },
  { Name: "The Death of Superman", Publisher: "DC Comics", Superhero:
    "Superman", Villain: "Doomsday" },
];

console.log(JSON.stringify(comicBooks));

```

You should execute this file independently so you can familiarize yourself with how it works and what it outputs.

2. failing-script.js

```
// failing-script.js  
throw new Error("This script is supposed to fail");
```

We are going to use this script as part of our unit tests. For now, run/execute this script so you can familiarize yourself with how it works and what it outputs.

3. Write the failing tests (Red):

```
// comic-books.spec.js  
const { ComicBooks } = require("../src/comic-books");  
  
describe("ComicBooks", () => {  
  let comicBooks;  
  
  beforeEach(() => {  
    comicBooks = new ComicBooks();  
  });  
  
  test("should return comic books data", (done) => {  
    comicBooks.getComicBooks((data, error) => {  
      expect(error).toBeNull();  
      expect(data).toEqual([  
        { Name: "The Dark Knight Returns", Publisher: "DC Comics", Superhero:  
"Batman", Villain: "The Joker" },  
        { Name: "The Death of Superman", Publisher: "DC Comics", Superhero:  
"Superman", Villain: "Doomsday" },  
      ]);  
      done();  
    });  
  });  
  
  test("should handle an error when the comic books data script is not found",  
(done) => {  
    const comicBooks = new ComicBooks("nonexistent-script.js");  
    comicBooks.getComicBooks((data, error) => {  
      expect(data).toBeNull();  
      expect(error).not.toBeNull();  
      done();  
    });  
  });
```

```

    });
  });

  test("should handle an error when the comic books data script fails", (done) =>
  {
    const comicBooks = new ComicBooks("failing-script.js");
    comicBooks.getComicBooks((data, error) => {
      expect(data).toBeNull();
      expect(error).not.toBeNull();
      done();
    });
  });
});

```

4. Write the code to pass the failing tests (Green):

```

// comic-books.js
"use strict";

const { spawn } = require("child_process");
const { join } = require("path");
const dataFile = join(__dirname, "comic-books-data.js");

class ComicBooks {
  constructor(scriptPath = dataFile) {
    this.scriptPath = scriptPath;
  }

  getComicBooks(callback) {
    const child = spawn("node", [this.scriptPath]);

    child.stdout.on("data", (data) => {
      const comicData = JSON.parse(data.toString());
      callback(comicData, null);
    });

    child.stderr.on("data", (data) => {
      console.error(` stderr: ${data}`);
      callback(null, new Error(data.toString()));
    });

    child.on("error", (error) => {
      console.error(` spawn error: ${error}`);
      callback(null, error);
    });
  }
}

```



```
});  
}  
}
```

```
module.exports = { ComicBooks };
```

Programming Exercises

In this assignment, you will be building a `GameCharacters` class that retrieves and processes data from a separate JavaScript file using Node.js child processes. This class will have a `getCharacters` method that spawns a child process to run a script, captures the data it sends back, and passes this data to a callback function.

You will also need to handle any errors that might occur during the process. If there's an error while spawning the child process, it should be logged to the console. If the child process sends any data to `stderr`, this should also be logged to the console and passed to the callback function.

You will also create a failing script to simulate and test error handling scenarios.

Additionally, you will create a data script (`game-characters-data.js`) that the `GameCharacters` class will use to retrieve the characters. This script should log a JSON stringified array of game characters to the console.

Requirements:

1. **Project Structure:** Your project should have the following structure:

```
fantasy-game-characters  
  src  
    game-characters.js  
    game-characters-data.js  
    failing-script.js  
  test  
    game-characters.spec.js  
  package.json
```

2. **TDD:** You must follow TDD principles. Write your tests first, then write the code to make them pass.
3. **Unit Tests:** You must write at least three-unit tests
 - a. Test that data is being returned from the `game-characters-data` script.
 - b. Test that it handles an error when the `game-characters-data` script is not found.

- c. Test that it handles an error when the game-characters-data script fails.
4. **Character data (game-characters-data.js):** Characters should have the following properties:
 - a. Class (Warrior, Mage, Rogue)
 - b. Gender (Male, Female, Other)
 - c. An additional property for something neat and fun about your character.
5. **GameCharacters module (game-characters.js):** Define the GameCharacters class with a constructor that accepts a script file name. The constructor should use the join function from the path module to create the path to the script file and store the path in an instance variable.
6. **Package.json:** Add a test script to the package.json file to run your tests with Jest.

Instructions:

1. Set up your project structure as described in the Requirements section.
2. Create the game-characters-data.js file. This script should log a JSON stringified array of game characters objects to the console.
3. Create the failing-script.js file. This script should log an error message to the stderr. This will be used to test the error handling in the GameCharacters class.
4. Write your first test in game-characters.spec.js. This test should test that data is being returned from the game-characters-data script.
5. Write code in the game-characters.js to make the test pass.
6. Run your test again to confirm that the test now passes.
7. Repeat steps 4-6 for the remaining two tests.
8. Make sure to handle errors properly in your code and tests.

Grading:

You will be awarded 20 points for each passing unit test, for a total of 60 points. If two tests pass, you will earn 40 points. If all three tests pass, you will earn the full 60 points.

Hints:

- Use the `child_process.spawn` function to run your script in a child process.
- Use the `path.join` function to create the path to your script file.

- Remember to call `done()` in your Jest test to signal that the asynchronous test is complete.
- Use the `beforeEach` function in Jest to create a new `GameCharacters` instance before each test.
- Use the `toEqual` function in Jest to compare the data returned by the `getCharacters` method to the expected data.
- Use the `toBeNull` function in Jest to check that an error is null when there's no error and that data is null when there's an error.
- Remember to follow TDD principles: write your test first, then write code to make the test pass.
- Refer to the examples in this chapter to complete this assignment.

References

Copilot. (n.d.). OpenAI. *Microsoft Copilot*. computer software. Retrieved December 19, 2023, from <https://www.microsoft.com/en-us/microsoft-copilot>.

Node.js. "Index | Node.js V20.10.0 Documentation." *Nodejs.org*, nodejs.org/docs/latest-v20.x/api/index.html. Accessed 9 Jan. 2024.