CHAPTER 4

JavaScript Primer

In this chapter, I provide a quick tour of the most important features of the JavaScript language as they apply to React development. I don't have the space to describe JavaScript completely, so I have focused on the essentials that you'll need to get up to speed and follow the examples in this book.

JavaScript has been modernized in recent years with the addition of convenient language features and a substantial expansion of the utility functions available for common tasks such as array handling. Not all browsers support the latest features, and so the React development tools include the Babel package, which is responsible for transforming JavaScript written using the latest features into code that can be relied on to work in most mainstream browsers. This is means you are able to enjoy a modern development experience without needing to pay attention to dealing with the differences between browsers and keeping track of the features each supports. Table 4-1 summarizes the chapter.

Table 4-1. *Chapter Summary*

Problem	Solution	Listing
Provide instructions that will be executed by the browser	Use JavaScript statement	4
Delay execution of statements until they are required	Use JavaScript functions	5–7, 10–12
Define functions with variable numbers of parameters	Use default and rest parameters	8, 9
Express functions concisely	Use fat arrow functions	13
Define variables and constants	Use the `let` and `const` keywords	14, 15
Use the JavaScript primitive types	Use the `string`, `number`, or `boolean` keywords	16, 17, 19
Define strings that include other values	Use template strings	18
Execute statements conditionally	Use the `if` and `else` and `switch` keywords	20
Compare values and identities	Use the equality and identity operators	21, 22
Convert types	Use the type conversion keywords	23–25
Group related items	Define an array	26, 27
Read or change a value in an array	Use the index accessor notation	28, 29
Enumerate the contents of an array	Use a `for` loop or the `forEach` method	30

(*continued*)

A. Freeman, *Pro React 16*, https://doi.org/10.1007/978-1-4842-4451-7_4

Table 4-1. (*continued*)

Problem	Solution	Listing
Expand the contents of an array	Use the spread operator	31, 32
Process the contents of an array	Use the built-in array method	33
Gather related values into a single unit	Define an object using a literal or a class	34–36, 40
Define an operation that can be performed on the values of an object	Define a method	37, 39, 43, 44
Copy properties and value from one object to another	Use the `Object.assign` method or use the spread operator	41, 42
Group related features	Define a JavaScript module	45–54
Observe an asynchronous operation	Define a `Promise` and use the `async` and `await` keywords	55–58

Preparing for This Chapter

In this chapter, I continue working with the `primer` project created in Chapter 3. To prepare for this chapter, I added a file called `example.js` to the `src` folder and added the code shown in Listing 4-1.

■ **Tip** You can download the example project for this chapter—and for all the other chapters in this book—from https://github.com/Apress/pro-react-16.

Listing 4-1. The Contents of the example.js File in the src Folder

```
console.log("Hello");
```

To incorporate the `example.js` file into the application, I added the statement shown in Listing 4-2 to the `index.js` file in the `src` folder.

Listing 4-2. Importing a File in the index.js File in the src Folder

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import * as serviceWorker from './serviceWorker';
import 'bootstrap/dist/css/bootstrap.css';

import "./example";

ReactDOM.render(<App />, document.getElementById('root'));

// If you want your app to work offline and load faster, you can change
// unregister() to register() below. Note this comes with some pitfalls.
// Learn more about service workers: http://bit.ly/CRA-PWA
serviceWorker.unregister();
```

Open a command prompt, navigate to the `primer` folder, and run the command shown in Listing 4-3 to start the React development tools.

Listing 4-3. Starting the Development Tools

```
npm start
```

The initial preparation of the project will take a moment, after which a new browser window or tab will open and navigate to `http://localhost:3000`, displaying the content shown in Figure 4-1.

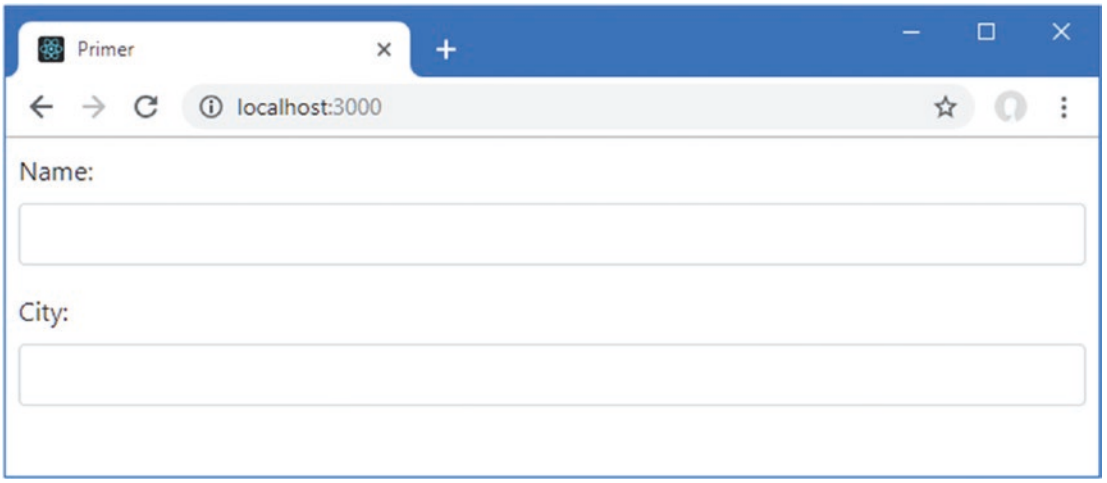

Figure 4-1. *Running the example application*

Open the browser's F12 development tools, which can usually be done by pressing F12 on the keyboard or right-clicking in the browser window and selecting Inspect from the pop-up menu. Inspect the Console tab, and you will see that the statement in the `example.js` file from Listing 4-1 has produced a simple result, as shown in Figure 4-2.

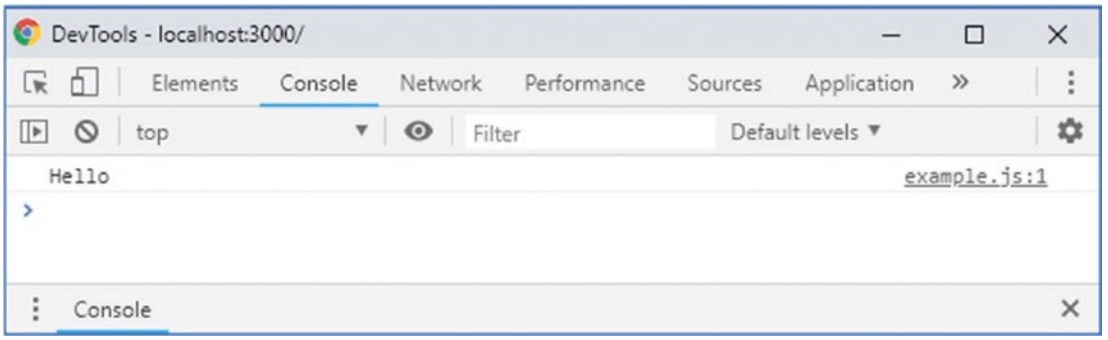

Figure 4-2. *A result in the browser's console*

All of the examples in this chapter produce text output and so rather than show screenshots of the Console tab, I will use just the text, like this:

```
Hello
```

Using Statements

The basic JavaScript building block is the *statement*. Each statement represents a single command, and statements are usually terminated by a semicolon (;). The semicolon is optional, but using them makes your code easier to read and allows for multiple statements on a single line. In Listing 4-4, I have added statements to the JavaScript file.

Listing 4-4. Adding JavaScript Statements in the example.js File in the src Folder

```
console.log("Hello");
console.log("Apples");
console.log("This is a statement");
console.log("This is also a statement");
```

The browser executes each statement in turn. In this example, all the statements simply write messages to the console. The results are as follows:

```
Hello
Apples
This is a statement
This is also a statement
```

Defining and Using Functions

When the browser receives JavaScript code, it executes the statements it contains in the order in which they are defined. This is what happened in the previous example. The statements in the `example.js` file were executed one by one, all of which wrote a message to the console, all in the order in which they were defined in `example.js`. You can also package statements into a *function*, which won't be executed until the browser encounters a statement that *invokes* that function, as shown in Listing 4-5.

Listing 4-5. Defining a JavaScript Function in the example.js File in the src Folder

```
const myFunc = function () {
    console.log("This statement is inside the function");
};

console.log("This statement is outside the function");

myFunc();
```

Defining a function simple: use the `const` keyword followed by the name you want to give the function, followed by the equal sign (=) and the `function` keyword, followed by parentheses (the (and) characters). The statements you want the function to contain are enclosed between braces (the { and } characters).

In the listing, I used the name myFunc, and the function contains a single statement that writes a message to the JavaScript console. The statement in the function won't be executed until the browser reaches another statement that calls the myFunc function, like this:

```
...
myFunc();
...
```

When you save the changes to the example.js file, the updated JavaScript code will be sent to the browser, where it is executed and produces the following output:

```
This statement is outside the function
This statement is inside the function
```

You can see that the statement inside the function isn't executed immediately, but other than demonstrating how functions are defined, this example isn't especially useful because the function is invoked immediately after it has been defined. Functions are much more useful when they are invoked in response to some kind of change or event, such as user interaction.

You can also define functions so you don't have to explicitly create and assign a variable, as shown in Listing 4-6.

Listing 4-6. Defining a Function in the example.js File in the src Folder

```
function myFunc() {
    console.log("This statement is inside the function");
}

console.log("This statement is outside the function");

myFunc();
```

The code works in the same way as Listing 4-5 but is more familiar for most developers. This example produces the same result as Listing 4-5.

Defining Functions with Parameters

JavaScript allows you to define parameters for functions, as shown in Listing 4-7.

Listing 4-7. Defining Functions with Parameters in the example.js File in the src Folder

```
function myFunc(name, weather) {
    console.log("Hello " + name + ".");
    console.log("It is " + weather + " today.");
}

myFunc("Adam", "sunny");
```

I added two parameters to the myFunc function, called name and weather. JavaScript is a dynamically typed language, which means you don't have to declare the data type of the parameters when you define the function. I'll come back to dynamic typing later in the chapter when I cover JavaScript variables. To invoke a function with parameters, you provide values as arguments when you invoke the function, like this:

```
...
myFunc("Adam", "sunny");
...
```

The results from this listing are as follows:

```
Hello Adam.
It is sunny today.
```

Using Default and Rest Parameters

The number of arguments you provide when you invoke a function doesn't need to match the number of parameters in the function. If you call the function with fewer arguments than it has parameters, then the value of any parameters you have not supplied values for is undefined, which is a special JavaScript value. If you call the function with more arguments than there are parameters, then the additional arguments are ignored.

The consequence of this is that you can't create two functions with the same name and different parameters and expect JavaScript to differentiate between them based on the arguments you provide when invoking the function. This is called *polymorphism*, and although it is supported in languages such as Java and C#, it isn't available in JavaScript. Instead, if you define two functions with the same name, then the second definition replaces the first.

There are two ways that you can modify a function to respond to a mismatch between the number of parameters it defines and the number of arguments used to invoke it. *Default parameters* deal with the situation where there are fewer arguments than parameters and allow you to provide a default value for the parameters for which there are no arguments, as shown in Listing 4-8.

Listing 4-8. Using a Default Parameter in the example.js File in the src Folder

```
function myFunc(name, weather = "raining") {
    console.log("Hello " + name + ".");
    console.log("It is " + weather + " today.");
}

myFunc("Adam");
```

The weather parameter in the function has been assigned a default value of raining, which will be used if the function is invoked with only one argument, producing the following results:

```
Hello Adam.
It is raining today.
```

Rest parameters are used to capture any additional arguments when a function is invoked with additional arguments, as shown in Listing 4-9.

Listing 4-9. Using a Rest Parameter in the example.js File in the src Folder

```
function myFunc(name, weather, ...extraArgs) {
    console.log("Hello " + name + ".");
    console.log("It is " + weather + " today.");
    for (let i = 0; i < extraArgs.length; i++) {
        console.log("Extra Arg: " + extraArgs[i]);
    }
}

myFunc("Adam", "sunny", "one", "two", "three");
```

The rest parameter must be the last parameter defined by the function, and its name is prefixed with an ellipsis (three periods, `...`). The rest parameter is an array to which any extra arguments will be assigned. In the listing, the function prints out each extra argument to the console, producing the following results:

```
Hello Adam.
It is sunny today.
Extra Arg: one
Extra Arg: two
Extra Arg: three
```

Defining Functions That Return Results

You can return results from functions using the `return` keyword. Listing 4-10 shows a function that returns a result.

Listing 4-10. Returning a Result from a Function in the example.js File in the src Folder

```
function myFunc(name) {
    return ("Hello " + name + ".");
}

console.log(myFunc("Adam"));
```

This function defines one parameter and uses it to produce a result. I invoke the function and pass the result as the argument to the `console.log` function, like this:

```
...
console.log(myFunc("Adam"));
...
```

Notice that you don't have to declare that the function will return a result or denote the data type of the result. The result from this listing is as follows:

```
Hello Adam.
```

Using Functions as Arguments to Other Functions

JavaScript functions can be treated as objects, which means you can use one function as the argument to another, as demonstrated in Listing 4-11.

Listing 4-11. Using a Function as an Arguments in the example.js File in the src Folder

```
function myFunc(nameFunction) {
    return ("Hello " + nameFunction() + ".");
}

console.log(myFunc(function () {
    return "Adam";
}));
```

The `myFunc` function defines a parameter called `nameFunction` that it invokes to get the value to insert into the string it returns. I pass a function that returns `Adam` as the argument to `myFunc`, which produces the following output:

```
Hello Adam.
```

Functions can be chained together, building up more complex functionality from small and easily tested pieces of code, as shown in Listing 4-12.

Listing 4-12. Chaining Functions Calls in the example.js File in the src Folder

```
function myFunc(nameFunction) {
    return ("Hello " + nameFunction() + ".");
}

function printName(nameFunction, printFunction) {
    printFunction(myFunc(nameFunction));
}

printName(function () { return "Adam" }, console.log);
```

This example produces the following output:

```
Hello Adam.
```

Using Arrow Functions

Arrow functions—also known as *fat arrow functions* or *lambda expressions*—are an alternative way of defining functions and are often used to define functions that are used only as arguments to other functions. Listing 4-13 replaces the functions from the previous example with arrow functions.

Listing 4-13. Using Arrow Functions in the example.js File in the src Folder

```
const myFunc = (nameFunction) => ("Hello " + nameFunction() + ".");

const printName = (nameFunction, printFunction) =>
    printFunction(myFunc(nameFunction));

printName(function () { return "Adam" }, console.log);
```

These functions perform the same work as the ones in Listing 4-12. There are three parts to an arrow function: the input parameters, then an equal sign and a greater-than sign (the "arrow"), and finally the function result. The `return` keyword and curly braces are required only if the arrow function needs to execute more than one statement. There are more examples of arrow functions later in this chapter, and you will see them used throughout the book.

■ **Note** In React development, you can decide which style of function you prefer to use, and you will see that I use both in the examples in this book. Care must be taken when defining functions that respond to events, however, as explained in Chapter 12.

Using Variables and Types

The `let` keyword is used to declare variables and, optionally, assign a value to the variable in a single statement—as opposed to the `const` keyword I used in earlier examples, which creates a constant value that cannot be modified.

When you use `let` or `const`, the variable or constant that you create can be accessed only in the region of code in which they are defined, which is known as the variable or constant's scope and which is demonstrated in Listing 4-14.

Listing 4-14. Using let to Declare Variables in the example.js File in the src Folder

```
function messageFunction(name, weather) {
    let message = "Hello, Adam";
    if (weather === "sunny") {
        let message = "It is a nice day";
        console.log(message);
    } else {
        let message = "It is " + weather + " today";
        console.log(message);
    }
    console.log(message);
}

messageFunction("Adam", "raining");
```

In this example, there are three statements that use the `let` keyword to define a variable called `message`. The scope of each variable is limited to the region of code that it is defined in, producing the following results:

```
It is raining today
Hello, Adam
```

This may seem like an odd example, but there is another keyword that can be used to declare variables: `var`. The `let` and `const` keywords are relatively new additions to the JavaScript specification that is intended to address some oddities in the way `var` behaves. Listing 4-15 takes the example from Listing 4-14 and replaces `let` with `var`.

USING LET AND CONST

It is good practice to use the `const` keyword for any value that you don't expect to change so that you receive an error if any modifications are attempted. This is a practice that I rarely follow, however—in part because I am still struggling to adapt to not using the `var` keyword and in part because I write code in a range of languages and there are some features that I avoid because they trip me up when I switch from one to another. If you are new to JavaScript, then I recommend trying to use `const` and `let` correctly and avoiding following my poor behavior.

Listing 4-15. Using var to Declare Variables in the example.js File in the src Folder

```
function messageFunction(name, weather) {
    var message = "Hello, Adam";
    if (weather === "sunny") {
        var message = "It is a nice day";
        console.log(message);
    } else {
        var message = "It is " + weather + " today";
        console.log(message);
    }
    console.log(message);
}

messageFunction("Adam", "raining");
```

When you save the changes in the listing, you will see the following results:

```
It is raining today
It is raining today
```

Some browsers will show repeated statements as a single line with a number next to them indicating how many times that output has occurred. This means you may see one statement with the number 2 next to it, indicating that it occurred twice.

The problem is that the `var` keyword creates variables whose scope is the containing function, which means that all the references to `message` are referring to the same variable. This can cause unexpected results for even experienced JavaScript developers and is the reason that the more conventional `let` keyword

was introduced. The React development tools include warnings for common problems, which is why you will also see the following messages in the JavaScript console:

```
Line 4:  'message' is already defined  no-redeclare
Line 7:  'message' is already defined  no-redeclare
```

These messages can be cryptic until you get used to them, and the easiest way to learn more about them is to consult the documentation for the ESLint package, which applies a set of rules to JavaScript code and is used by the React development tools to create the warnings. The name of the rule is included in the warning, and the name of the rule that produced the warnings for Listing 4-15 is no-redeclare, which is described at https://eslint.org/docs/rules/no-redeclare.

USING VARIABLE CLOSURE

If you define a function inside another function—creating *inner* and *outer* functions—then the inner function is able to access the outer function's variables, using a feature called *closure*, like this:

```
function myFunc(name) {
    let myLocalVar = "sunny";
    let innerFunction = function () {
        return ("Hello " + name + ". Today is " + myLocalVar + ".");
    }
    return innerFunction();

}

console.log(myFunc("Adam"));
```

The inner function in this example is able to access the local variables of the outer function, including its parameter. This is a powerful feature that means you don't have to define parameters on inner functions to pass around data values, but caution is required because it is easy to get unexpected results when using common variable names like `counter` or `index`, where you may not realize that you are reusing a variable name from the outer function.

Using the Primitive Types

JavaScript defines a basic set of primitive types: `string`, `number`, `boolean`. This may seem like a short list, but JavaScript manages to fit a lot of flexibility into these types.

■ **Tip** I am simplifying here. There are three other primitives that you may encounter. Variables that have been declared but not assigned a value are `undefined`, while the `null` value is used to indicate that a variable has no value, just as in other languages. The final primitive type is `Symbol`, which is an immutable value that represents a unique ID but which is not widely used at the time of writing.

Working with Booleans

The boolean type has two values: true and false. Listing 4-16 shows both values being used, but this type is most useful when used in conditional statements, such as an if statement. There is no console output from this listing, although you will see warnings because the variables have been defined and not used.

Listing 4-16. Defining boolean Values in the example.js File in the src Folder

```
let firstBool = true;
let secondBool = false;
```

Working with Strings

You define string values using either the double quote or single quote characters, as shown in Listing 4-17.

Listing 4-17. Defining string Variables in the example.js File in the src Folder

```
let firstString = "This is a string";
let secondString = 'And so is this';
```

The quote characters you use must match. You can't start a string with a single quote and finish with a double quote, for example. There is no console output for this listing. JavaScript provides string objects with a basic set of properties and methods, the most useful of which are described in Table 4-2.

Table 4-2. *Useful string Properties and Methods*

Name	Description
length	This property returns the number of characters in the string.
charAt(index)	This method returns a string containing the character at the specified index.
concat(string)	This method returns a new string that concatenates the string on which the method is called and the string provided as an argument.
indexOf(term, start)	This method returns the first index at which term appears in the string or -1 if there is no match. The optional start argument specifies the start index for the search.
replace(term, newTerm)	This method returns a new string in which all instances of term are replaced with newTerm.
slice(start, end)	This method returns a substring containing the characters between the start and end indices.
split(term)	This method splits up a string into an array of values that were separated by term.
toUpperCase() toLowerCase()	These methods return new strings in which all the characters are uppercase or lowercase.
trim()	This method returns a new string from which all the leading and trailing whitespace characters have been removed.

Using Template Strings

A common programming task is to combine static content with data values to produce a string that can be presented to the user. The traditional way to do this is through string concatenation, which is the approach I have been using in the examples so far in this chapter, as follows:

```
...
let message = "It is " + weather + " today";
...
```

JavaScript also supports *template strings*, which allow data values to be specified inline, which can help reduce errors and result in a more natural development experience. Listing 4-18 shows the use of a template string.

Listing 4-18. Using a Template String in the example.js File in the src Folder

```
function messageFunction(weather) {
    let message = `It is ${weather} today`;
    console.log(message);
}

messageFunction("raining");
```

Template strings begin and end with backticks (the ` character), and data values are denoted by curly braces preceded by a dollar sign. This string, for example, incorporates the value of the `weather` variable into the template string:

```
...
let message = `It is ${weather} today`;
...
```

This example produces the following output:

```
It is raining today
```

Working with Numbers

The `number` type is used to represent both *integer* and *floating-point* numbers (also known as *real numbers*). Listing 4-19 provides a demonstration.

Listing 4-19. Defining number Values in the example.js File in the src Folder

```
let daysInWeek = 7;
let pi = 3.14;
let hexValue = 0xFFFF;
```

You don't have to specify which kind of number you are using. You just express the value you require, and JavaScript will act accordingly. In the listing, I have defined an integer value, defined a floating-point value, and prefixed a value with `0x` to denote a hexadecimal value.

Using JavaScript Operators

JavaScript defines a largely standard set of operators. I've summarized the most useful in Table 4-3.

Table 4-3. *Useful JavaScript Operators*

Operator	Description
`++, --`	Pre- or post-increment and decrement
`+, -, *, /, %`	Addition, subtraction, multiplication, division, remainder
`<, <=, >, >=`	Less than, less than or equal to, more than, more than or equal to
`==, !=`	Equality and inequality tests
`===, !==`	Identity and nonidentity tests
`&&, ||`	Logical AND and OR (|| is used to coalesce null values)
`=`	Assignment
`+`	String concatenation
`?:`	Three-operand conditional statement

Using Conditional Statements

Many of the JavaScript operators are used in conjunction with conditional statements. In this book, I tend to use the `if/else` and `switch` statements. Listing 4-20 shows the use of both, which will be familiar to most developers.

Listing 4-20. Using Conditional Statements in the example.js File in the src Folder

```
let name = "Adam";

if (name === "Adam") {
    console.log("Name is Adam");
} else if (name === "Jacqui") {
    console.log("Name is Jacqui");
} else {
    console.log("Name is neither Adam or Jacqui");
}

switch (name) {
    case "Adam":
        console.log("Name is Adam");
        break;
    case "Jacqui":
        console.log("Name is Jacqui");
        break;
    default:
        console.log("Name is neither Adam or Jacqui");
        break;
}
```

This example produces the following results:

```
Name is Adam
Name is Adam
```

The Equality Operator vs. the Identity Operator

The equality and identity operators are of particular note. The equality operator will attempt to coerce (convert) operands to the same type to assess equality. This is a handy feature, as long as you are aware it is happening. Listing 4-21 shows the equality operator in action.

Listing 4-21. Using the Equality Operator in the example.js File in the src Folder

```
let firstVal = 5;
let secondVal = "5";

if (firstVal == secondVal) {
    console.log("They are the same");
} else {
    console.log("They are NOT the same");
}
```

The output from this example is as follows:

```
They are the same
```

JavaScript is converting the two operands into the same type and comparing them. In essence, the equality operator tests that values are the same irrespective of their type. This causes sufficient confusion that you will also see a warning in the JavaScript console:

```
Line 4:  Expected '===' and instead saw '=='  eqeqeq
```

A more predictable way of making comparisons is to use the identity operator (===, three equal signs, rather than the two of the equality operator), as shown in Listing 4-22.

Listing 4-22. Using the Identity Operator in the example.js File in the src Folder

```
let firstVal = 5;
let secondVal = "5";

if (firstVal === secondVal) {
    console.log("They are the same");
} else {
    console.log("They are NOT the same");
}
```

In this example, the identity operator will consider the two variables to be different. This operator doesn't coerce types. The result is as follows:

```
They are NOT the same
```

Explicitly Converting Types

The string concatenation operator (+) has precedence over the addition operator (also +), which means that JavaScript will concatenate variables in preference to adding. This can cause confusion because JavaScript will also convert types freely to produce a result—and not always the result that is expected, as shown in Listing 4-23.

Listing 4-23. String Concatenation Operator Precedence in the example.js File in the src Folder

```
let myData1 = 5 + 5;
let myData2 = 5 + "5";

console.log("Result 1: " + myData1);
console.log("Result 2: " + myData2);
```

These statements produce the following result:

```
Result 1: 10
Result 2: 55
```

The second result is the kind that causes confusion. What might be intended to be an addition operation is interpreted as string concatenation through a combination of operator precedence and over-eager type conversion. To avoid this, you can explicitly convert the types of values to ensure you perform the right kind of operation, as described in the following sections.

Converting Numbers to Strings

If you are working with multiple number variables and want to concatenate them as strings, then you can convert the numbers to strings with the `toString` method, as shown in Listing 4-24.

Listing 4-24. Using the number.toString Method in the example.js File in the src Folder

```
let myData1 = (5).toString() + String(5);

console.log("Result: " + myData1);
```

Notice that I placed the numeric value in parentheses, and then I called the `toString` method. This is because you have to allow JavaScript to convert the literal value into a `number` before you can call the methods that the `number` type defines. I have also shown an alternative approach to achieve the same effect, which is to call the `String` function and pass in the numeric value as an argument. Both of these techniques have the same effect, which is to convert a `number` to a `string`, meaning that the + operator is used for string concatenation and not addition. The output from this script is as follows:

```
Result: 55
```

There are some other methods that allow you to exert more control over how a number is represented as a string. I briefly describe these methods in Table 4-4. All of the methods shown in the table are defined by the `number` type.

Table 4-4. *Useful Number-to-String Methods*

Method	Description
`toString()`	This method returns a string that represents a number in base 10.

`toString(2)`
`toString(8)`
`toString(16)`
	This method returns a string that represents a number in binary, octal, or hexadecimal notation.

`toFixed(n)`	This method returns a string representing a real number with `n` digits after the decimal point.
`toExponential(n)`	This method returns a string that represents a number using exponential notation with one digit before the decimal point and `n` digits after.
`toPrecision(n)`	This method returns a string that represents a number with `n` significant digits, using exponential notation if required.

Converting Strings to Numbers

The complementary technique is to convert strings to numbers so that you can perform addition rather than concatenation. You can do this with the `Number` function, as shown in Listing 4-25.

Listing 4-25. Converting Strings to Numbers in the example.js File in the src Folder

```
let firstVal = "5";
let secondVal = "5";

let result = Number(firstVal) + Number(secondVal);
console.log("Result: " + result);
```

The output from this script is as follows:

```
Result: 10
```

The `Number` function is strict in the way that is parses string values, but there are two other functions you can use that are more flexible and will ignore trailing non-number characters. These functions are `parseInt` and `parseFloat`. I have described all three methods in Table 4-5.

Table 4-5. *Useful String to Number Methods*

Method	Description
`Number(str)`	This method parses the specified string to create an integer or real value.
`parseInt(str)`	This method parses the specified string to create an integer value.
`parseFloat(str)`	This method parses the specified string to create an integer or real value.

Working with Arrays

JavaScript arrays work like arrays in most other programming languages. Listing 4-26 shows how you can create and populate an array.

Listing 4-26. Creating and Populating an Array in the example.js File in the src Folder

```
let myArray = new Array();
myArray[0] = 100;
myArray[1] = "Adam";
myArray[2] = true;
```

I have created a new array by calling `new Array()`. This creates an empty array, which I assign to the variable `myArray`. In the subsequent statements, I assign values to various index positions in the array. (There is no output from this listing.)

There are a couple of things to note in this example. First, I didn't need to declare the number of items in the array when I created it. JavaScript arrays will resize themselves to hold any number of items. The second point is that I didn't have to declare the data types that the array will hold. Any JavaScript array can hold any mix of data types. In the example, I have assigned three items to the array: a `number`, a `string`, and a `boolean`.

Using an Array Literal

The example in Listing 4-26 produces a warning because using `new Array()` isn't the standard way to create an array. Instead, the array literal style lets you create and populate an array in a single statement, as shown in Listing 4-27.

Listing 4-27. Using the Array Literal Style in the example.js File in the src Folder

```
let myArray = [100, "Adam", true];
```

In this example, I specified that the `myArray` variable should be assigned a new array by specifying the items I wanted in the array between square brackets ([and]). (There is no console output from this listing, although there will be a warning because the array is defined but not used.)

Reading and Modifying the Contents of an Array

You read the value at a given index using square braces ([and]), placing the index you require between the braces, as shown in Listing 4-28.

Listing 4-28. Reading the Data from an Array Index in the example.js File in the src Folder

```
let myArray = [100, "Adam", true];

console.log(`Index 0: ${myArray[0]}`);
```

You can modify the data held in any position in a JavaScript array simply by assigning a new value to the index. Just as with regular variables, you can switch the data type at an index without any problems. The output from the listing is as follows:

```
Index 0: 100
```

Listing 4-29 demonstrates how to modify the contents of an array.

Listing 4-29. Modifying the Contents of an Array in the example.js File in the src Folder

```
let myArray = [100, "Adam", true];
myArray[0] = "Tuesday";

console.log(`Index 0: ${myArray[0]}`);
```

In this example, I have assigned a `string` to position `0` in the array, a position that was previously held by a `number` and produces this output:

```
Index 0: Tuesday
```

Enumerating the Contents of an Array

You enumerate the content of an array using a `for` loop or using the `forEach` method, which receives a function that is called to process each element in the array. Both approaches are shown in Listing 4-30.

Listing 4-30. Enumerating the Contents of an Array in the example.js File in the src Folder

```
let myArray = [100, "Adam", true];

for (let i = 0; i < myArray.length; i++) {
    console.log(`Index ${i}: ${myArray[i]}`);
}

console.log("---");

myArray.forEach((value, index) => console.log(`Index ${index}: ${value}`));
```

The JavaScript `for` loop works just the same way as loops in many other languages. You determine how many elements there are in the array by using the `length` property.

The function passed to the `forEach` method is given two arguments: the value of the current item to be processed and the position of that item in the array. In this listing, I have used an arrow function as the argument to the `forEach` method, which is the kind of use for which they excel (and you will see used throughout this book). The output from the listing is as follows:

```
Index 0: 100
Index 1: Adam
Index 2: true
---
Index 0: 100
Index 1: Adam
Index 2: true
```

Using the Spread Operator

The spread operator is used to expand an array so that its contents can be used as function arguments. Listing 4-31 defines a function that accepts multiple arguments and invokes it using the values in an array with and without the spread operator.

Listing 4-31. Using the Spread Operator in the example.js File in the src Folder

```
function printItems(numValue, stringValue, boolValue) {
    console.log(`Number: ${numValue}`);
    console.log(`String: ${stringValue}`);
    console.log(`Boolean: ${boolValue}`);
}

let myArray = [100, "Adam", true];

printItems(myArray[0], myArray[1], myArray[2]);

printItems(...myArray);
```

The spread operator is an ellipsis (a sequence of three periods), and it causes the array to be unpacked and passed to the `printItems` function as individual arguments.

```
...
printItems(...myArray);
...
```

The spread operator also makes it easy to concatenate arrays, as shown in Listing 4-32.

Listing 4-32. Concatenating Arrays in the example.js File in the src Folder

```
let myArray = [100, "Adam", true];
let myOtherArray = [200, "Bob", false, ...myArray];

myOtherArray.forEach((value, index) => console.log(`Index ${index}: ${value}`));
```

Using the spread operator, I am able to specify `myArray` as an item when I define `myOtherArray`, with the result that the contents of the first array will be unpacked and added as items to the second array. This example produces the following results:

```
Index 0: 200
Index 1: Bob
Index 2: false
Index 3: 100
Index 4: Adam
Index 5: true
```

■ **Note** Arrays can also be de-structured, whereby the individual elements of an array are assigned to different variables, so that `[var1, var2] = [3, 4]` assigns a value of `3` to `var1` and `4` to `var2`. Array de-structuring is used by the hooks feature, which is described in Chapter 11.

Using the Built-in Array Methods

The JavaScript `Array` object defines a number of methods that you can use to work with arrays, the most useful of which are described in Table 4-6.

Table 4-6. *Useful Array Methods*

Method	Description
`concat(otherArray)`	This method returns a new array that concatenates the array on which it has been called with the array specified as the argument. Multiple arrays can be specified.
`join(separator)`	This method joins all the elements in the array to form a string. The argument specifies the character used to delimit the items.
`pop()`	This method removes and returns the last item in the array.
`shift()`	This method removes and returns the first element in the array.
`push(item)`	This method appends the specified item to the end of the array.
`unshift(item)`	This method inserts a new item at the start of the array.
`reverse()`	This method returns a new array that contains the items in reverse order.
`slice(start,end)`	This method returns a section of the array.
`sort()`	This method sorts the array. An optional comparison function can be used to perform custom comparisons.
`splice(index, count)`	This method removes `count` items from the array, starting at the specified `index`. The removed items are returned as the result of the method.
`unshift(item)`	This method inserts a new item at the start of the array.
`every(test)`	This method calls the `test` function for each item in the array and returns `true` if the function returns `true` for all of them and `false` otherwise.
`some(test)`	This method returns `true` if calling the `test` function for each item in the array returns `true` at least once.
`filter(test)`	This method returns a new array containing the items for which the `test` function returns `true`.
`find(test)`	This method returns the first item in the array for which the `test` function returns `true`.
`findIndex(test)`	This method returns the index of the first item in the array for which the `test` function returns `true`.
`forEach(callback)`	This method invokes the `callback` function for each item in the array, as described in the previous section.
`includes(value)`	This method returns `true` if the array contains the specified value.
`map(callback)`	This method returns a new array containing the result of invoking the `callback` function for every item in the array.
`reduce(callback)`	This method returns the accumulated value produced by invoking the callback function for every item in the array.

Since many of the methods in Table 4-6 return a new array, these methods can be chained together to process data, as shown in Listing 4-33.

Listing 4-33. Processing an Array in the example.js File in the src Folder

```
let products = [
    { name: "Hat", price: 24.5, stock: 10 },
    { name: "Kayak", price: 289.99, stock: 1 },
    { name: "Soccer Ball", price: 10, stock: 0 },
    { name: "Running Shoes", price: 116.50, stock: 20 }
];

let totalValue = products
    .filter(item => item.stock > 0)
    .reduce((prev, item) => prev + (item.price * item.stock), 0);

console.log(`Total value: $${totalValue.toFixed(2)}`);
```

I use the `filter` method to select the items in the array whose `stock` value is greater than zero and use the `reduce` method to determine the total value of those items, producing the following output:

```
Total value: $2864.99
```

Working with Objects

There are several ways to create objects in JavaScript. Listing 4-34 gives a simple example to get started.

Listing 4-34. Creating an Object in the example.js File in the src Folder

```
let myData = new Object();
myData.name = "Adam";
myData.weather = "sunny";

console.log(`Hello ${myData.name}.`);
console.log(`Today is ${myData.weather}.`);
```

I create an object by calling `new Object()`, and I assign the result (the newly created object) to a variable called `myData`. Once the object is created, I can define properties on the object just by assigning values, like this:

```
...
myData.name = "Adam";
...
```

Prior to this statement, my object doesn't have a property called `name`. When the statement has executed, the property does exist, and it has been assigned the value `Adam`. You can read the value of a property by combining the variable name and the property name with a period, like this:

```
...
console.log(`Hello ${myData.name}.`);
...
```

The result from the listing is as follows:

```
Hello Adam.
Today is sunny.
```

Using Object Literals

The previous example produces a warning because the standard way to define objects is to do so using the object literal format, which also allows properties to be defined in a single step, as shown in Listing 4-35.

Listing 4-35. Using the Object Literal Format in the example.js File in the src Folder

```
let myData = {
    name: "Adam",
    weather: "sunny"
};

console.log(`Hello ${myData.name}.`);
console.log(`Today is ${myData.weather}.`);
```

Each property that you want to define is separated from its value using a colon (`:`), and properties are separated using a comma (`,`). The effect is the same as in the previous example, and the result from the listing is as follows:

```
Hello Adam.
Today is sunny.
```

Using Variables as Object Properties

If you use a variable as an object property, JavaScript will use the variable name as the property name and the variable value as the property value, as shown in Listing 4-36.

Listing 4-36. Using a Variable in an Object Literal in the example.js File in the src Folder

```
let name = "Adam"

let myData = {
  name,
  weather: "sunny"
};

console.log(`Hello ${myData.name}.`);
console.log(`Today is ${myData.weather}.`);
```

The name variable is used to add a property to the myData object, such that the property is taken from the variable, name in this case, as its value, Adam. This is a useful technique when you want to combine a set of data values into an object, and you will see it used in examples in later chapters. The code in Listing 4-37 produces the following output:

```
Hello Adam.
Today is sunny.
```

Using Functions as Methods

One of the features that I like most about JavaScript is the way you can add functions to objects. A function defined on an object is called a *method*. Listing 4-37 shows how you can add methods in this manner.

Listing 4-37. Adding Methods to an Object in the example.js File in the src Folder

```
let myData = {
    name: "Adam",
    weather: "sunny",
    printMessages: function () {
        console.log(`Hello ${myData.name}.`);
        console.log(`Today is ${myData.weather}.`);
    }
};

myData.printMessages();
```

In this example, I have used a function to create a method called printMessages. Notice that to refer to the properties defined by the object, I have to use the this keyword. When a function is used as a method, the function is implicitly passed the object on which the method has been called as an argument through the special variable this. The output from the listing is as follows:

```
Hello Adam.
Today is sunny.
```

You can also define methods without using the function keyword, as shown in Listing 4-38.

Listing 4-38. Defining a Method in the example.js File in the src Folder

```
let myData = {
    name: "Adam",
    weather: "sunny",
    printMessages() {
        console.log(`Hello ${myData.name}.`);
        console.log(`Today is ${myData.weather}.`);
    }
};

myData.printMessages();
```

The output from this listing is as follows:

```
Hello Adam.
Today is sunny.
```

The fat arrow syntax can also be used to define methods, as shown in Listing 4-39.

Listing 4-39. Defining a Fat Arrow Method in the example.js File in the src Folder

```
let myData = {
    name: "Adam",
    weather: "sunny",
    printMessages: () => {
        console.log(`Hello ${myData.name}.`);
        console.log(`Today is ${myData.weather}.`);
    }
};

myData.printMessages();
```

■ **Tip** If you are returning an object literal as the result from a fat arrow function, then you must enclose the object in parentheses, e.g., `myFunc = () => ({ data: "hello"})`. You will receive an error if you omit the parentheses because the build tools will assume that the curly braces of the object literal are the start and end of a function body.

Using Classes

Classes are templates for objects, defining the properties and methods that new instances will possess. Classes are a recent addition to the JavaScript language, and they are used in React development to define components that have state data, as explained in Chapter 11. In Listing 4-40, I have replaced the object literal with a class.

Listing 4-40. Using a Class in the example.js File in the src Folder

```
class MyData {

    constructor() {
        this.name = "Adam";
        this.weather = "sunny";
    }

    printMessages = () => {
        console.log(`Hello ${this.name}.`);
        console.log(`Today is ${this.weather}.`);
    }
}

let myData = new MyData();
myData.printMessages();
```

Classes are defined using the `class` keyword. The `constructor` is a special method that is automatically invoked when an object is created from the class, which is known as *instantiating the class*. An object created from a class is said to be an *instance* of that class.

In JavaScript, the constructor is used to define the properties that instances will have, and the current object is referred to using the `this` keyword. The constructor in Listing 4-40 defines `name` and `weather` properties by assigning values to `this.name` and `this.weather`. Classes define methods by assigning functions to names, and in Listing 4-40, the class defines a `printMessages` method that is defined using the fat arrow syntax and that prints out messages to the console. Notice that the `this` keyword is required to access the values of the `name` and `weather` variables.

■ **Tip** There are other ways to use JavaScript classes, but I have focused on the way they are used in React development and in the examples throughout this book. See `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes` for full details.

A new instance of the class is created using the `new` keyword, and a class can be used to create multiple objects, each of which has its own data values that are separate from the other instances. In the listing, the `new` keyword is used to create an object from the `MyData` class, which is then assigned to a variable named `myData`. The object's `printMessages` method is invoked, producing the following output:

```
Hello Adam.
Today is sunny.
```

In other languages and frameworks, classes are used for inheritance, where one class builds on the methods and properties defined by another. React development does not use class inheritance directly and uses an alternative approach, known as *composition*, to create complex features, as described in Chapter 14. The exception is when a React component is defined using a class, where the `extends` keyword must be used to ensure that the class inherits the core features required for a component. If you examine the contents of the `App.js` file, you will see that the component is defined using the `class` and `extends` keywords, like this:

```
...
import React, { Component } from "react";

export default class App extends Component {

    render = () =>
        <div className="m-2">
            <div className="form-group">
                <label>Name:</label>
                <input className="form-control" />
            </div>
            <div className="form-group">
                <label>City:</label>
                <input className="form-control" />
            </div>
        </div>
}
...
```

Copying Properties from One Object to Another

Some important features provided by React and the packages I describe in Part 3 rely on copying the properties from one object to another. JavaScript provides the `Object.assign` method for this purpose, as demonstrated in Listing 4-41.

Listing 4-41. Copying Object Properties in the example.js File in the src Folder

```
class MyData {

    constructor() {
        this.name = "Adam";
        this.weather = "sunny";
    }

    printMessages = () => {
        console.log(`Hello ${this.name}.`);
        console.log(`Today is ${this.weather}.`);
    }
}

let myData = new MyData();

let secondObject = {};

Object.assign(secondObject, myData);

secondObject.printMessages();
```

This example uses the literal form to create a new object that has no properties and uses the `Object.assign` method to copy the properties—and their values—from the `myData` object. This example produces the following output:

```
Hello Adam.
Today is sunny.
```

The destructuring operator—which is the same as the spread operator—can be used to copy properties from one object to another, and a technique I use in later chapters is to copy all of the existing properties using the destructuring operator and then define a new value for some of them, as shown in Listing 4-42.

Listing 4-42. Copying Using a Spread in the example.js File in the src Folder

```
class MyData {

    constructor() {
        this.name = "Adam";
        this.weather = "sunny";
    }

    printMessages = () => {
        console.log(`Hello ${this.name}.`);
```

```
        console.log(`Today is ${this.weather}.`);
    }
}

let myData = new MyData();

let secondObject = { ...myData, weather: "cloudy"};

console.log(`myData: ${ myData.weather}, secondObject: ${secondObject.weather}`);
```

This example copies the properties from the `myData` object and provides a new value for the `weather` property, producing the following output:

```
myData: sunny, secondObject: cloudy
```

Capturing Parameter Names from Objects

When an object is received as a function or method parameter, it can be awkward to navigate through the properties to get the data required. As a simple example, Listing 4-43 defines a structure of objects that are navigated to get data values.

Listing 4-43. Navigating Object Properties in the example.js File in the src Folder

```
const myData = {
    name: "Bob",
    location: {
        city: "Paris",
        country: "France"
    },
    employment: {
        title: "Manager",
        dept: "Sales"
    }
}

function printDetails(data) {
    console.log(`Name: ${data.name}, City: ${data.location.city},
        Role: ${data.employment.title}`);
}

printDetails(myData);
```

The `printDetails` function has to navigate through the object to get the `name`, `city`, and `title` properties it requires. The same outcome can be achieved more elegantly by capturing specific properties as named parameters, as shown in Listing 4-44.

Listing 4-44. Capturing Named Parameters in the example.js File in the src Folder

```
const myData = {
    name: "Bob",
    location: {
        city: "Paris",
        country: "France"
    },
    employment: {
        title: "Manager",
        dept: "Sales"
    }
}

function printDetails({ name, location: { city }, employment: { title }}) {
    console.log(`Name: ${name}, City: ${city}, Role: ${title}`);
}

printDetails(myData);
```

This example applies the technique described in Listing 4-36 to select specific properties from the object. This listing and Listing 4-43 produce the same output.

```
Name: Bob, City: Paris, Role: Manager
```

Understanding JavaScript Modules

React applications are too complex to define in a single JavaScript file. To break up an application into more manageable chunks, JavaScript supports *modules*, which contain JavaScript code that other parts of the application depend on. In the sections that follow, I explain the different ways that modules can be defined and used.

Creating and Using a JavaScript Module

There are already JavaScript modules in the example project, but the best way to understand how they work is to create and use a new module. I added a file called `sum.js` in the `src` folder and added the code shown in Listing 4-45.

Listing 4-45. The Contents of the sum.js File in the src Folder

```
export default function(values) {
    return values.reduce((total, val) => total + val, 0);
}
```

The `sum.js` file contains a function that accepts an array of values and uses the JavaScript array `reduce` method to sum them and return the result. What's important about this example is not what it does but the fact that the function is defined in its own file, which is the basic building block for a module.

There are two keywords used in Listing 4-45 that you will often encounter when defining modules: `export` and `default`. The `export` keyword is used to denote the features that will be available outside the

module. By default, the contents of the JavaScript file are private and must be explicitly shared using the `export` keyword before they can be used in the rest of the application. The `default` keyword is used when the module contains a single feature, such as the function defined in Listing 4-45. Together, the `export` and `default` keywords are used to specify that the only function in the `sum.js` file is available for use in the rest of the application.

Using the JavaScript Module

Another keyword is required to use a module: the `import` keyword. In Listing 4-46, I used the `import` keyword to access the function defined in the previous section so that it can be used in the `example.js` file.

Listing 4-46. Using a JavaScript Module in the example.js File in the src Folder

```
import additionFunction from "./sum";

let values = [10, 20, 30, 40, 50];

let total = additionFunction(values);

console.log(`Total: ${total}`);
```

The `import` keyword is used to declare a dependency on the module. The `import` keyword can be used in a number of different ways, but this is the format you will use most often when working with modules you have created yourself, and the key parts are illustrated in Figure 4-3.

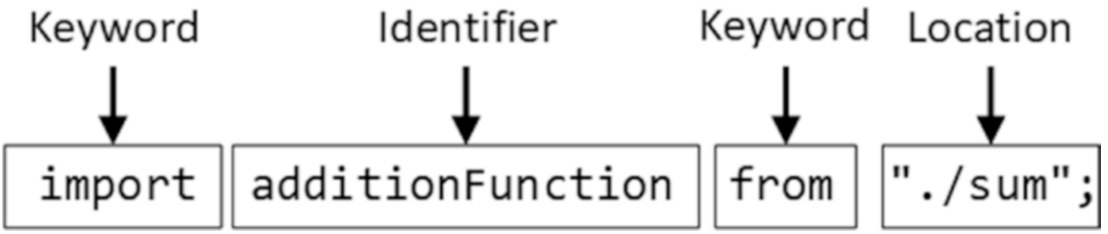

Figure 4-3. *Declaring a dependency on a module*

The `import` keyword is followed by an identifier, which is the name by which the function will be known when it is used, and the identifier in this example is `additionFunction`.

■ **Tip** Notice that it is the `import` statement in which the identifier is applied, which means that the code that consumes the function from the module chooses the name by which it will be known and that multiple `import` statements for the same module in different parts of the application can use different names to refer to the same function. See the next section for details of how the module can specify the names of the features it contains.

The from keyword follows the identifier, which is then followed by the location of the module. It is important to pay close attention to the location because different behaviors are created by different location formats, as described in the sidebar.

During the build process, the React tools will detect the import statement and include the function from the sum.js file in the JavaScript file that is sent to the browser so that it can execute the application. The identifier used in the import statement can be used to access the function in the module, in just the same way that locally defined functions are used.

```
...
let total = additionFunction(values);
...
```

If you examine the browser's JavaScript console, you will see that the code in Listing 4-42 uses the module's function to produce the following result:

```
Total: 150
```

UNDERSTANDING MODULE LOCATIONS

The location of a module changes the way that the build tools will look for the module when creating the JavaScript file that is sent to the browser. For modules you have defined yourself, the location is specified as a relative path; it starts with one or two periods, which indicates that the path is relative to the current file or to the current file's parent directory. In Listing 4-46, the location starts with a period.

```
...
import additionFunction from "./sum";
...
```

This location tells the build tools that there is a dependency on the sum module, which can be found in the same folder as the file that contains the import statement. Notice that the file extension is not included in the location.

If you omit the initial period, then the import statement declares a dependency on a module in the node_modules folder, which is where packages are installed during the project setup. This kind of location is used to access features provided by third-party packages, including the React packages, which is why you will see statements like this in React projects:

```
...
import React, { Component } from "react";
...
```

The location for this import statement doesn't start with a period and will be interpreted as a dependency on the react module in the project's node_modules folder, which is the package that provides the core React application features.

Exporting Named Features from a Module

A module can assign names to the features it exports, which is the approach I have taken for most of the examples in this book. In Listing 4-47, I have given a name to the function that is exported by the sum module.

Listing 4-47. Exporting a Named Feature in the sum.js File in the src Folder

```
export function sumValues (values) {
    return values.reduce((total, val) => total + val, 0);
}
```

The function provides the same feature but is exported using the name sumValues and no longer uses the default keyword. In Listing 4-48, I have imported the feature using its new name in the example.js file.

Listing 4-48. Importing a Named Feature in the example.js File in the src Folder

```
import { sumValues } from "./sum";

let values = [10, 20, 30, 40, 50];

let total = sumValues(values);

console.log(`Total: ${total}`);
```

The name of the feature to be imported is specified in curly braces (the { and } characters) and is used by this name in the code. A module can export default and named features, as shown in Listing 4-49.

Listing 4-49. Exporting Named and Default Features in the sum.js File in the src Folder

```
export function sumValues (values) {
    return values.reduce((total, val) => total + val, 0);
}

export default function sumOdd(values) {
    return sumValues(values.filter((item, index) => index % 2 === 0));
}
```

The new feature is exported using the default keyword. In Listing 4-50, I have imported the new feature as the default export from the module.

Listing 4-50. Importing a Default Feature in the example.js File in the src Folder

```
import oddOnly, { sumValues } from "./sum";

let values = [10, 20, 30, 40, 50];

let total = sumValues(values);
let odds = oddOnly(values);

console.log(`Total: ${total}, Odd Total: ${odds}`);
```

This is the pattern you will see at the start of the React components in the examples throughout this book because the core React features required for JSX are the default export from the `react` module and the `Component` class is a named feature:

```
...
import React, { Component } from "react";
...
```

The example in Listing 4-50 produces the following output:

```
Total: 150, Odd Total: 90
```

Defining Multiple Named Features in a Module

Modules can contain more than one named function or value, which is useful for grouping related features. To demonstrate, I created a file called `operations.js` to the `src` folder and added the code shown in Listing 4-51.

Listing 4-51. The Contents of the operations.js File in the src Folder

```
export function multiply(values) {
    return values.reduce((total, val) => total * val, 1);
}

export function subtract(amount, values) {
    return values.reduce((total, val) => total - val, amount);
}

export function divide(first, second) {
    return first / second;
}
```

This module defines three functions to which the `export` keyword has been applied. Unlike the previous example, the `default` keyword is not used, and each function has its own name. When importing from a module that contains multiple features, the names of the required features are specified as a comma-separated list between the braces, as shown in Listing 4-52.

Listing 4-52. Importing Named Features in the example.js File in the src Folder

```
import oddOnly, { sumValues } from "./sum";
import { multiply, subtract } from "./operations";

let values = [10, 20, 30, 40, 50];

let total = sumValues(values);
let odds = oddOnly(values);

console.log(`Total: ${total}, Odd Total: ${odds}`);
console.log(`Multiply: ${multiply(values)}`);
console.log(`Subtract: ${subtract(1000, values)}`);
```

The braces that follow the `import` keyword surround the list of functions that I want to use, which is the `multiply` and `subtract` functions in this case, separated by commas. I only declare dependencies on the functions that I require, and there is no dependency on the `divide` function, which is defined in the module but not used. This example produces the following output:

```
Total: 150, Odd Total: 90
Multiply: 12000000
Subtract: 850
```

Changing Module Feature Names

When importing named features from modules, you may find that there are two modules that use the same name or that the name used by the module doesn't produce readable code when it is imported. You can select a new name using the `as` keyword, as shown in Listing 4-53.

Listing 4-53. Assigning a Name to a Feature in the example.js File in the src Folder

```
import oddOnly, { sumValues } from "./sum";
import { multiply, subtract as deduct } from "./operations";

let values = [10, 20, 30, 40, 50];

let total = sumValues(values);
let odds = oddOnly(values);

console.log(`Total: ${total}, Odd Total: ${odds}`);
console.log(`Multiply: ${multiply(values)}`);
console.log(`Subtract: ${deduct(1000, values)}`);
```

I used the `as` keyword to specify that the `subtract` function should be given the name `deduct` when imported into the `example.js` file. This listing produces the same output as Listing 4-53.

Importing an Entire Module

Listing the names of all the functions in a module gets out of hand for complex modules. A more elegant approach is to import all the features provided by a module and just use the features you require, as shown in Listing 4-54.

Listing 4-54. Importing an Entire Module in the example.js File in the src Folder

```
import oddOnly, { sumValues } from "./sum";
import * as ops from "./operations";

let values = [10, 20, 30, 40, 50];

let total = sumValues(values);
let odds = oddOnly(values);

console.log(`Total: ${total}, Odd Total: ${odds}`);
console.log(`Multiply: ${ops.multiply(values)}`);
console.log(`Subtract: ${ops.subtract(1000, values)}`);
```

An asterisk is used to import everything in a module, followed by the as keyword and an identifier through which the module functions and values will be accessed. In this case, the identifier is ops, which means that the multiply, subtract, and divide functions can be accessed as ops.multiply, ops.subtract, and ops.divide. This listing produces the same output as Listing 4-53.

Understanding JavaScript Promises

A promise is a background activity that will be completed at some point in the future. The most common use for promises in this book is requesting data using an HTTP request, which is performed asynchronously and produces a result when a response is received from the web server.

Understanding the Asynchronous Operation Problem

The classic asynchronous operation for a web application is an HTTP request, which is typically used to get the data and content that a user requires. I explain how to make HTTP requests in Part 3 of this book, but I need something simpler for this chapter, so I added a file called async.js to the src folder with the code shown in Listing 4-55.

Listing 4-55. The Contents of the async.js File in the src Folder

```
import { sumValues } from "./sum";

export function asyncAdd(values) {
    setTimeout(() => {
        let total = sumValues(values);
        console.log(`Async Total: ${total}`);
        return total;
    }, 500);
}
```

The setTimeout function invokes a function asynchronously after a specified delay. In the listing, the asyncAdd function receives a parameter that is passed to the sumValues function defined in the sum module after a delay of 500 milliseconds, creating a background operation that doesn't complete immediately for the examples in this chapter and acting as a placeholder for more useful operations, such as making an HTTP request. In Listing 4-56, I have updated the example.js file to use the asyncAdd function.

Listing 4-56. Performing Background Work in the example.js File in the src Folder

```
import { asyncAdd } from "./async";

let values = [10, 20, 30, 40, 50];

let total = asyncAdd(values);

console.log(`Main Total: ${total}`);
```

The problem this example demonstrates is that the result from the asyncAdd function isn't produced until after the statements in the example.js file have been executed, which you can see in the output shown in the browser's JavaScript console:

```
Main Total: undefined
Async Total: 150
```

The browser executes the statements in the example.js file and invokes the asyncAdd function as instructed. The browser moves on to the next statement in the example.js file, which writes a message to the console using the result provided by asyncAdd—but this happens before the asynchronous task has been completed, which is why the output is undefined. The asynchronous task subsequently completes, but it is too late for the result to be used by the example.js file.

Using a JavaScript Promise

To solve the problem in the previous section, I need a mechanism that allows me to observe the asynchronous task so that I can wait for it to complete and then write out the result. This is the role of the JavaScript Promise, which I have applied to the asyncAdd function in Listing 4-57.

Listing 4-57. Using a Promise in the async.js File in the src Folder

```
import { sumValues } from "./sum";

export function asyncAdd(values) {
    return new Promise(callback =>
        setTimeout(() => {
            let total = sumValues(values);
            console.log(`Async Total: ${total}`);
            callback(total);
        }, 500));
}
```

It can be difficult to unpack the functions in this example. The new keyword is used to create a Promise, which accepts the function that is to be observed. The observed function is provided with a callback that is invoked when the asynchronous task has completed and that accepts the result of the task as an argument. Invoking the callback function is known as *resolving the promise*.

The Promise object that has become the result of the asyncAdd function allows the asynchronous task to be observed so that follow-up work can be performed when the task completes, as shown in Listing 4-58.

Listing 4-58. Observing a Promise in the example.js File in the src Folder

```
import { asyncAdd } from "./async";

let values = [10, 20, 30, 40, 50];

asyncAdd(values).then(total => console.log(`Main Total: ${total}`));
```

The then method accepts a function that will be invoked when the callback is used. The result passed to the callback is provided to the then function. In this case, that means the total isn't written to the browser's JavaScript console until the asynchronous task has completed and produces the following output:

```
Async Total: 150
Main Total: 150
```

Simplifying the Asynchronous Code

JavaScript provides two keywords—async and await—that support asynchronous operations without having to work directly with promises. In Listing 4-59, I have applied these keywords in the example.js file.

■ **Caution** It is important to understand that using async/await doesn't change the way that an application behaves. The operation is still performed asynchronously, and the result will not be available until the operation completes. These keywords are just a convenience to simplify working with asynchronous code so that you don't have to use the then method.

Listing 4-59. Using async and await in the example.js File in the src Folder

```
import { asyncAdd } from "./async";

let values = [10, 20, 30, 40, 50];

async function doTask() {
    let total = await asyncAdd(values);
    console.log(`Main Total: ${total}`);
}

doTask();
```

These keywords can be applied only to functions, which is why I added the doTask function in this listing. The async keyword tells JavaScript that this function relies on functionality that requires a promise. The await keyword is used when calling a function that returns a Promise and has the effect of assigning the result provided to the Promise object's callback and then executing the statements that follow, producing the following result:

```
Async Total: 150
Main Total: 150
```

Summary

In this chapter, I provided a brief primer on JavaScript, focusing on the core functionality that will get you started for React development. In the next chapter, I start the process of building a more complex and realistic project, called SportsStore.