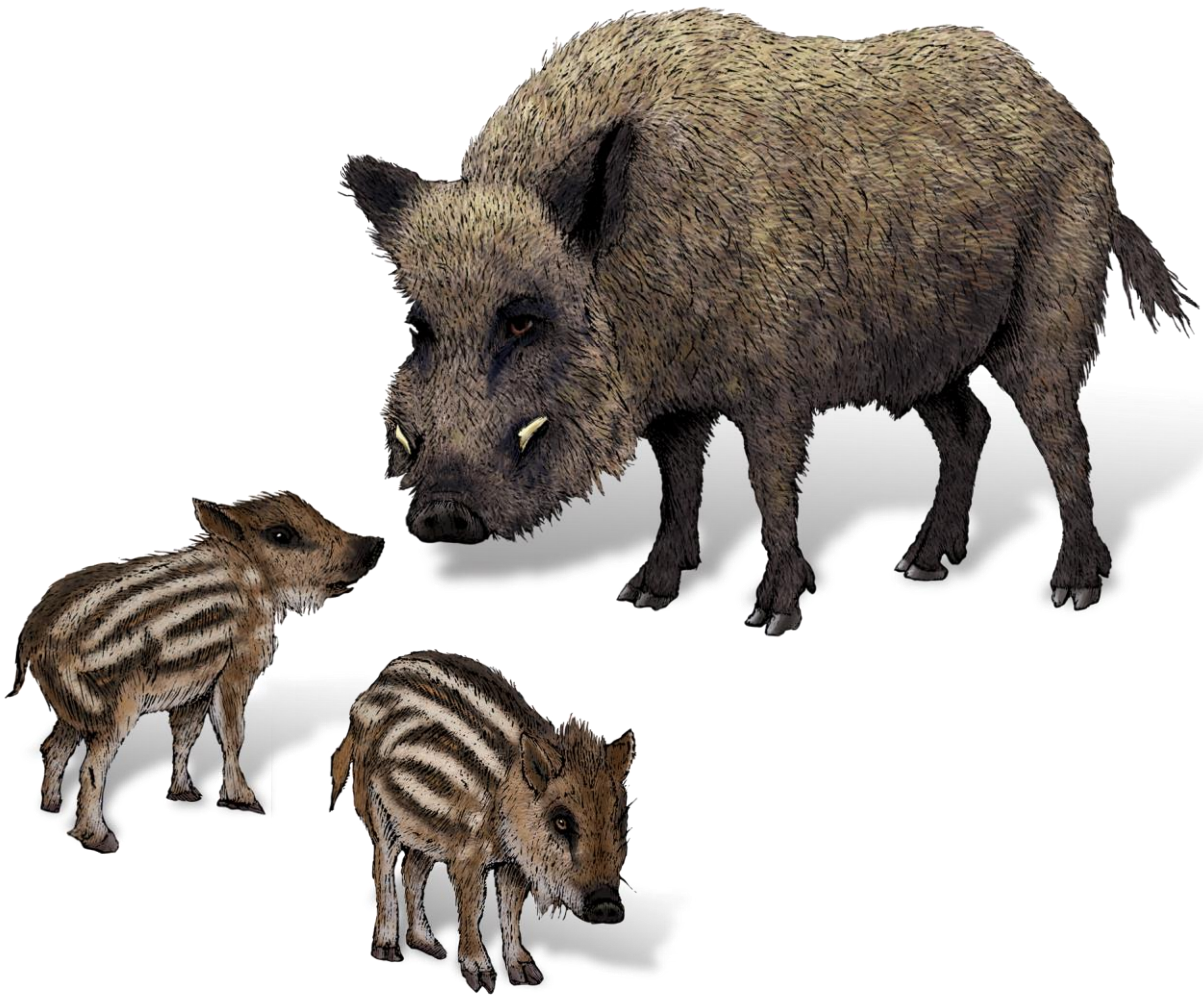
Jordan Jordanov

Learning JS

Modern Development Patterns



CHAPTER 1

Intro

Since its release in 1995, JavaScript has gone through many changes. At first, we used JavaScript to add interactive elements to web pages: button clicks, hover states, form validation, etc.. Later, JavaScript got more robust with DHTML and AJAX. Today, with Node.js, JavaScript has become a real software language that's used to build full- stack applications. JavaScript is everywhere.

JavaScript's evolution has been guided by a group of individuals from companies thatuse JavaScript, browser vendors, and community leaders. The committee in charge ofshepherding the changes to JavaScript over the years is the European Computer Man-ufacturers Association (ECMA). Changes to the language are community-driven,originating from proposals written by community members. Anyone can submit a proposal to the ECMA committee. The responsibility of the ECMA committee is to manage and prioritize these proposals to decide what's included in each spec.

The first release of ECMAScript was in 1997, ECMAScript1. This was followed in 1998 by ECMAScript2. ECMAScript3 came out in 1999, adding regular expressions, string handling, and more. The process of agreeing on an ECMAScript4 became a chaotic, political mess that proved to be impossible. It was never released. In 2009, ECMAScript5(ES5) was released, bringing features like new array methods, object properties, and library support for JSON.

Since then, there has been a lot more momentum in this space. After ES6 or ES2015 was released in, yes, 2015, there have been yearly releases of new JS features. Any- thing that's part of the stage proposals is typically called ESNext, which is a simplified way of saying this is the next stuff that will be part of the JavaScript spec.

Proposals are taken through clearly defined stages, from stage 0, which represents the newest proposals, up through stage 4, which represents the finished proposals. When a proposal gains traction, it's up to the browser vendors like Chrome and Firefox to

implement the features. Consider the `const` keyword. When creating variables, we used to use `var` in all cases. The ECMA committee decided there should be a `const` keyword to declare constants (more on that later in the chapter). When `const` was first introduced, you couldn't just write `const` in JavaScript code and expect it to run in a browser. Now you can because browser vendors have changed the browser to support it.

Many of the features we'll discuss in this chapter are already supported by the newest browsers, but we'll also be covering how to compile your JavaScript code. This is the process of transforming new syntax that the browser doesn't recognize into older syn- tax that the browser understands. The kangax compatibility table is a great place to stay informed about the latest JavaScript features and their varying degrees of supportby browsers.

In this chapter, we'll show you JavaScript syntax

Declaring Variables

Prior to ES2015, the only way to declare a variable was with the `var` keyword. Wenow have a few different options that provide improved functionality.

The const Keyword

A constant is a variable that cannot be overwritten. Once declared, you cannot change its value. A lot of the variables that we create in JavaScript should not be over- written, so we'll be using `const` a lot. Like other languages had done before it, Java-Script introduced constants with ES6.

Before constants, all we had were variables, and variables could be overwritten:

```
var pizza = true;
pizza = false;
console.log(pizza); // false
```

We cannot reset the value of a constant variable, and it will generate a console error (as shown in Figure 2-1) if we try to overwrite the value:

```
const pizza = true;
pizza = false;
```

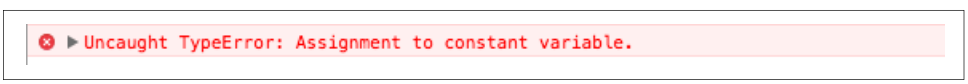


Figure 2-1. An attempt at overwriting a constant

The let Keyword

JavaScript now has *lexical variable scope*. In JavaScript, we create code blocks with curly braces ({}). In functions, these curly braces block off the scope of any variable declared with `var`. On the other hand, consider `if/else` statements. If you're comingfrom other languages, you might assume that these blocks would also block variable scope. This was not the case until `let` came along.

If a variable is created inside of an `if/else` block, that variable is not scoped to the block:

```
var topic = "JavaScript";

if (topic) {
  var topic = "React";
  console.log("block", topic); // block React
}

console.log("global", topic); // global React
```

The `topic` variable inside the `if` block resets the value of `topic` outside of the block.

With the `let` keyword, we can scope a variable to any code block. Using `let` protectsthe value of the global variable:

```
var topic = "JavaScript";

if (topic) {
  let topic = "React";
  console.log("block", topic); // React
}

console.log("global", topic); // JavaScript
```

The value of `topic` is not reset outside of the block.

Another area where curly braces don't block off a variable's scope is in `for` loops:

```
var div,
  container = document.getElementById("container");

for (var i = 0; i < 5; i++) {
  div = document.createElement("div");
  div.onclick = function() {
    alert("This is box #" + i);
  };
  container.appendChild(div);
}
```

In this loop, we create five `div`s to appear within a container. Each `div` is assigned an `onclick` handler that creates an alert box to display the index. Declaring `i` in the `for`loop creates a global variable named `i`, then iterates over it until its value reaches `5`.

When you click on any of these boxes, the alert says that `i` is equal to `5` for all `div`s,because the current value for the global `i` is `5` (see Figure 2-2).

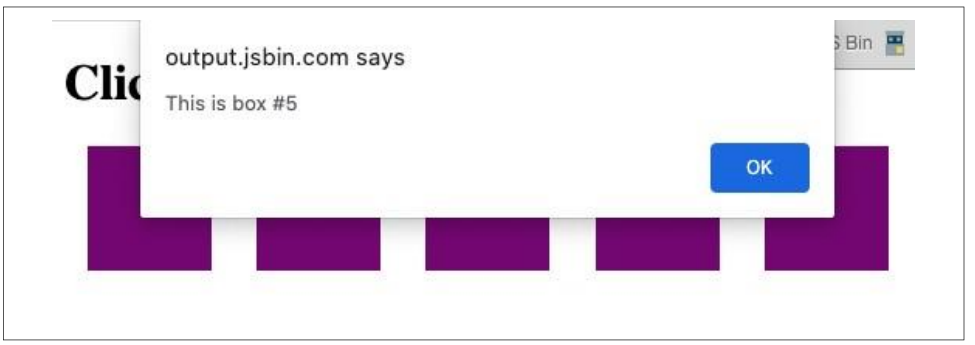


Figure 2-2. i is equal to 5 for each box

Declaring the loop counter `i` with `let` instead of `var` does block off the scope of `i`. Now clicking on any box will display the value for `i` that was scoped to the loop itera-tion (see Figure 2-3):

```
const container = document.getElementById("container");
let div;
for (let i = 0; i < 5; i++) {
  div = document.createElement("div");
  div.onclick = function () {
    alert("This is box #: " + i);
  };
  div.innerHTML = "Hello";
  container.appendChild(div);
}
```

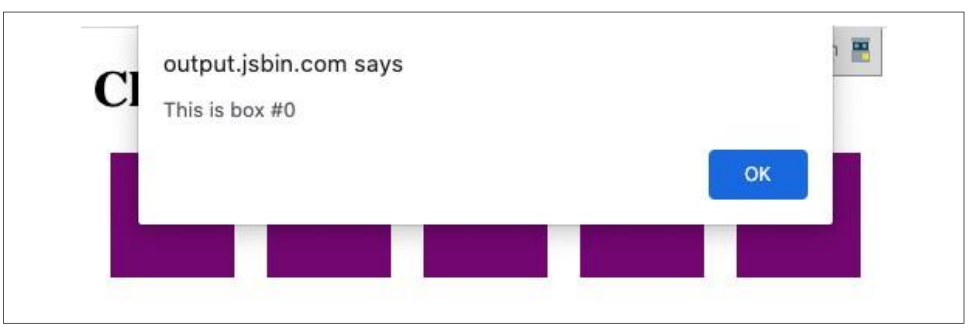


Figure 2-3. The scope of `i` is protected with `let`

The scope of `i` is protected with `let`.

Template Strings

Template strings provide us with an alternative to string concatenation. They also allow us to insert variables into a string. You'll hear these referred to as template strings, template literals, or string templates interchangeably.

Traditional string concatenation uses plus signs to compose a string using variable values and strings:

```
console.log(lastName + ", " + firstName + " " + middleName);
```

With a template, we can create one string and insert the variable values by surround- ing them with ${ }:

```
console.log(`${lastName}, ${firstName} ${middleName}`);
```

Any JavaScript that returns a value can be added to a template string between the ${ } in a template string.

Template strings honor whitespace, making it easier to draft up email templates, code examples, or anything else that contains whitespace. Now you can have a string that spans multiple lines without breaking your code:

```
const email = `
Hello ${firstName},

Thanks for ordering ${qty} tickets to ${event}.

Order Details
${firstName} ${middleName} ${lastName}
     ${qty} x $${price} = $${qty*price} to ${event}

You can pick your tickets up 30 minutes before
the show.

Thanks,

${ticketAgent}
`
```

Previously, using an HTML string directly in our JavaScript code was not so easy to do because we'd need to run it together on one line. Now that the whitespace is recog- nized as text, you can insert formatted HTML that is easy to read and understand:

```
document.body.innerHTML = `
<section>
  <header>
      <h1>The React Blog</h1>
  </header>
  <article>
      <h2>${article.title}</h2>
      ${article.body}
```

```
  </article>
  <footer>
      <p>copyright ${new Date().getYear()} | The React Blog</p>
  </footer>
</section>
`;
```

Notice that we can include variables for the page title and article text as well.

Creating Functions

Any time you want to perform some sort of repeatable task with JavaScript, you can use a function. Let's take a look at some of the different syntax options that can be used to create a function and the anatomy of those functions.

Function Declarations

A function declaration or function definition starts with the `function` keyword, which is followed by the name of the function, `logCompliment`. The JavaScript statements that are part of the function are defined between the curly braces:

```
function logCompliment() {
  console.log("You're doing great!");
}
```

Once you've declared the function, you'll invoke or call it to see it execute:

```
function logCompliment() {
  console.log("You're doing great!");
}

logCompliment();
```

Once invoked, you'll see the compliment logged to the console.

Function Expressions

Another option is to use a function expression. This just involves creating the function as a variable:

```
const logCompliment = function() {
  console.log("You're doing great!");
};

logCompliment();
```

The result is the same, and `You're doing great!` is logged to the console.

You can invoke a function before you write a function declaration. You cannot invoke a function created by a function expression.

For example , this will work.:

```
// Invoking the function before it's declared
hey();
// Function Declaration
function hey() {
  alert("hey!");
}
```

You'll see the alert appear in the browser. It works because the function is hoisted, or moved up, to the top of the file's scope. Trying the same exercise with a function expression will cause an error:

```
// Invoking the function before it's declared
hey();
// Function Expression
const hey = function() {
  alert("hey!");
};
```

```
TypeError: hey is not a function
```

This is obviously a small example, but this TypeError can occasionally arise when importing files and functions in a project. If you see it, you can always refactor as a declaration.

Passing arguments

The `logCompliment` function currently takes in no arguments or parameters. If we want to provide dynamic variables to the function, we can pass named parameters to a function simply by adding them to the parentheses. Let's start by adding a `first Name` variable:

```
const logCompliment = function(firstName) {
  console.log(`You're doing great, ${firstName}`);
};

logCompliment("Molly");
```

Now when we call the `logCompliment` function, the `firstName` value sent will be added to the console message.

We could add to this a bit by creating another argument called `message`. Now, we won't hard-code the message. We'll pass in a dynamic value as a parameter:

```
const logCompliment = function(firstName, message) {
  console.log(`${firstName}: ${message}`);
};

logCompliment("Molly", "You're so cool");
```

Function returns

The `logCompliment` function currently logs the compliment to the console, but moreoften, we'll use a function to return a value. Let's add a `return` statement to this func-tion. A `return` statement specifies the value returned by the function. We'll rename the function `createCompliment`:

```
const createCompliment = function(firstName, message) {
  return `${firstName}: ${message}`;
};

createCompliment("Molly", "You're so cool");
```

If you wanted to check to see if the function is executing as expected, just wrap the function call in a `console.log`:

```
console.log(createCompliment("You're so cool", "Molly"));
```

Default Parameters

Languages including C++ and Python allow developers to declare default values for function arguments. Default parameters are included in the ES6 spec, so in the event that a value is not provided for the argument, the default value will be used.

For example, we can set up default strings for the parameters `name` and `activity`:

```
function logActivity(name = "Shane McConkey", activity = "skiing") {
  console.log(`${name} loves ${activity}`);
}
```

If no arguments are provided to the `logActivity` function, it will run correctly usingthe default values. Default arguments can be any type, not just strings:

```
const defaultPerson = {
  name: {
    first: "Shane",
    last: "McConkey"
  },
  favActivity: "skiing"
};

function logActivity(person = defaultPerson) {
  console.log(`${person.name.first} loves ${person.favActivity}`);
}
```

Arrow Functions

Arrow functions are a useful new feature of ES6. With arrow functions, you can create functions without using the `function` keyword. You also often do not have to use the `return` keyword. Let's consider a function that takes in a `firstName` and returns astring, turning the person into a lord. Anyone can be a lord:

```
const lordify = function(firstName) {
  return `${firstName} of Canterbury`;
};

console.log(lordify("Dale")); // Dale of Canterbury
console.log(lordify("Gail")); // Gail of Canterbury
```

With an arrow function, we can simplify the syntax tremendously:

```
const lordify = firstName => `${firstName} of Canterbury`;
```

With the arrow, we now have an entire function declaration on one line. The `function` keyword is removed. We also remove `return` because the arrow points to what should be returned. Another benefit is that if the function only takes one argument, we can remove the parentheses around the arguments.

More than one argument should be surrounded by parentheses:

```
// Typical function
const lordify = function(firstName, land) {
  return `${firstName} of ${land}`;
};

// Arrow Function
const lordify = (firstName, land) => `${firstName} of ${land}`;

console.log(lordify("Don", "Piscataway")); // Don of Piscataway
console.log(lordify("Todd", "Schenectady")); // Todd of Schenectady
```

We can keep this as a one-line function because there is only one statement that needsto be returned. If there are multiple lines, you'll use curly braces:

```
const lordify = (firstName, land) => {
  if (!firstName) {
    throw new Error("A firstName is required to lordify");
  }

  if (!land) {
    throw new Error("A lord must have a land");
  }

  return `${firstName} of ${land}`;
};

console.log(lordify("Kelly", "Sonoma")); // Kelly of Sonoma
console.log(lordify("Dave")); // ! JAVASCRIPT ERROR
```

These `if/else` statements are surrounded with brackets but still benefit from theshorter syntax of the arrow function.

Returning objects

What happens if you want to return an object? Consider a function called `person` that builds an object based on parameters passed in for `firstName` and `lastName`:

```
const person = (firstName, lastName) =>
    {
        first: firstName,
        last: lastName
    }

console.log(person("Brad", "Janson"));
```

As soon as you run this, you'll see the error: `Uncaught SyntaxError: Unexpectedtoken :`. To fix this, just wrap the object you're returning with parentheses:

```
const person = (firstName, lastName) => ({
  first: firstName,
  last: lastName
});

console.log(person("Flad", "Hanson"));
```

These missing parentheses are the source of countless bugs in JavaScript and Reactapps, so it's important to remember this one!

Arrow functions and scope

Regular functions do not block `this`. For example, `this` becomes something else inthe `setTimeout` callback, not the `tahoe` object:

```
const tahoe = {
  mountains: ["Freel", "Rose", "Tallac", "Rubicon", "Silver"],
  print: function(delay = 1000) {
    setTimeout(function() {
      console.log(this.mountains.join(",  "));
    }, delay);
  }
};

tahoe.print(); // Uncaught TypeError: Cannot read property 'join' of undefined
```

This error is thrown because it's trying to use the `.join` method on what `this` is. If welog `this`, we'll see that it refers to the `Window` object:

```
console.log(this); // Window {}
```

To solve this problem, we can use the arrow function syntax to protect the scope of `this`:

```
const tahoe = {
  mountains: ["Freel", "Rose", "Tallac", "Rubicon", "Silver"],
  print: function(delay = 1000) {
```

```
    setTimeout(() => {
      console.log(this.mountains.join(",  "));
    }, delay);
  }
};

tahoe.print(); // Freel, Rose, Tallac, Rubicon, Silver
```

This works as expected, and we can `.join` the resorts with a comma. Be careful that you're always keeping scope in mind. Arrow functions do not block off the scope of `this`:

```
const tahoe = {
  mountains: ["Freel", "Rose", "Tallac", "Rubicon", "Silver"],
  print: (delay = 1000) => {
    setTimeout(() => {
      console.log(this.mountains.join(",  "));
    }, delay);
  }
};

tahoe.print(); // Uncaught TypeError: Cannot read property 'join' of undefined
```

Changing the `print` function to an arrow function means that `this` is actually the window.

Compiling JavaScript

When a new JavaScript feature is proposed and gains support, the community often wants to use it before it's supported by all browsers. The only way to be sure that yourcode will work is to convert it to more widely compatible code before running it inthe browser. This process is called *compiling*. One of the most popular tools for JavaScript compilation is Babel.

In the past, the only way to use the latest JavaScript features was to wait weeks, months, or even years until browsers supported them. Now, Babel has made it possible to use the latest features of JavaScript right away. The compiling step makes JavaScript similar to other languages. It's not quite traditional compiling: our code isn't compiled to binary. Instead, it's transformed into syntax that can be interpreted by a wider range of browsers. Also, JavaScript now has source code, meaning that there will be some files that belong to your project that don't run in the browser.

As an example, let's look at an arrow function with some default arguments:

```
const add = (x = 5, y = 10) => console.log(x + y);
```

If we run Babel on this code, it will generate the following:

```
"use strict";
```

```
var add = function add() {
  var x =
    arguments.length <= 0 || arguments[0] === undefined ? 5 : arguments[0];
  var y =
    arguments.length <= 1 || arguments[1] === undefined ? 10 : arguments[1];
  return console.log(x + y);
};
```

Babel added a “use strict” declaration to run in strict mode. The variables `x` and `y` aredefaulted using the `arguments` array, a technique you may be familiar with. The resulting JavaScript is more widely supported.

A great way to learn more about how Babel works is to check out the Babel REPL on the documentation website. Type some new syntax on the left side, then see some older syntax created.

The process of JavaScript compilation is typically automated by a build tool like webpack or Parcel.

Objects and Arrays

Since ES2016, JavaScript syntax has supported creative ways of scoping variables within objects and arrays. These creative techniques are widely used among the React community. Let’s take a look at a few of them, including destructuring, object literal enhancement, and the spread operator.

Destructuring Objects

Destructuring assignment allows you to locally scope fields within an object and to declare which values will be used. Consider the `sandwich` object. It has four keys, but we only want to use the values of two. We can scope `bread` and `meat` to be used locally:

```
const sandwich = {
  bread: "dutch crunch",
  meat: "tuna",
  cheese: "swiss",
  toppings: ["lettuce", "tomato", "mustard"]
};

const { bread, meat } = sandwich;

console.log(bread, meat); // dutch crunch tuna
```

The code pulls `bread` and `meat` out of the object and creates local variables for them. Also, since we declared these destructed variables using `let`, the `bread` and `meat` vari-ables can be changed without changing the original sandwich:

```
const sandwich = {
  bread: "dutch crunch",
  meat: "tuna",
  cheese: "swiss",
  toppings: ["lettuce", "tomato", "mustard"]
};

let { bread, meat } = sandwich;

bread = "garlic";
meat = "turkey";

console.log(bread); // garlic
console.log(meat); // turkey

console.log(sandwich.bread, sandwich.meat); // dutch crunch tuna
```

We can also destructure incoming function arguments. Consider this function thatwould log a person's name as a lord:

```
const lordify = regularPerson => {
  console.log(`${regularPerson.firstname} of Canterbury`);
};

const regularPerson = {
  firstname: "Bill",
  lastname: "Wilson"
};

lordify(regularPerson); // Bill of Canterbury
```

Instead of using dot notation syntax to dig into objects, we can destructure the valueswe need out of `regularPerson`:

```
const lordify = ({ firstname }) => {
  console.log(`${firstname} of Canterbury`);
};

const regularPerson = {
  firstname: "Bill",
  lastname: "Wilson"
};

lordify(regularPerson); // Bill of Canterbury
```

Let's take this one level farther to reflect a data change. Now, the `regularPerson` object has a new nested object on the `spouse` key:

```
const regularPerson = {
  firstname: "Bill",
  lastname: "Wilson",
  spouse: {
    firstname: "Phil",
```

```
    lastname: "Wilson"
  }
};
```

If we wanted to lordify the spouse's first name, we'd adjust the function's destructuredarguments slightly:

```
const lordify = ({ spouse: { firstname } }) => {
  console.log(`${firstname} of Canterbury`);
};

lordify(regularPerson); // Phil of Canterbury
```

Using the colon and nested curly braces, we can destructure the `firstname` from the `spouse` object.

Destructuring Arrays

Values can also be destructured from arrays. Imagine that we wanted to assign the first value of an array to a variable name:

```
const [firstAnimal] = ["Horse", "Mouse", "Cat"];

console.log(firstAnimal); // Horse
```

We can also pass over unnecessary values with *list matching* using commas. List matching occurs when commas take the place of elements that should be skipped. With the same array, we can access the last value by replacing the first two values withcommas:

```
const [, , thirdAnimal] = ["Horse", "Mouse", "Cat"];

console.log(thirdAnimal); // Cat
```

Later in this section, we'll take this example a step farther by combining array destructuring and the spread operator.

Object Literal Enhancement

Object literal enhancement is the opposite of destructuring. It's the process of restructuring or putting the object back together. With object literal enhancement, we can grab variables from the global scope and add them to an object:

```
const name = "Tallac";
const elevation = 9738;

const funHike = { name, elevation };

console.log(funHike); // {name: "Tallac", elevation: 9738}
```

`name` and `elevation` are now keys of the `funHike` object.

We can also create object methods with object literal enhancement or restructuring:

```
const name = "Tallac";
const elevation = 9738;
const print = function() {
  console.log(`Mt. ${this.name} is ${this.elevation} feet tall`);
};

const funHike = { name, elevation, print };

funHike.print(); // Mt. Tallac is 9738 feet tall
```

Notice we use `this` to access the object keys.

When defining object methods, it's no longer necessary to use the `function` keyword:

```
// Old
var skier = {
  name: name,
  sound: sound,
  powderYell: function() {
    var yell = this.sound.toUpperCase();
    console.log(`${yell} ${yell} ${yell}!!!`);
  },
  speed: function(mph) {
    this.speed = mph;
    console.log("speed:", mph);
  }
};

// New
const skier = {
  name,
  sound,
  powderYell() {
    let yell = this.sound.toUpperCase();
    console.log(`${yell} ${yell} ${yell}!!!`);
  },
  speed(mph) {
    this.speed = mph;
    console.log("speed:", mph);
  }
};
```

Object literal enhancement allows us to pull global variables into objects and reduces typing by making the `function` keyword unnecessary.

The Spread Operator

The spread operator is three dots (`...`) that perform several different tasks. First, the spread operator allows us to combine the contents of arrays. For example, if we had two arrays, we could make a third array that combines the two arrays into one:

```
const peaks = ["Tallac", "Ralston", "Rose"];
const canyons = ["Ward", "Blackwood"];
const tahoe = [...peaks, ...canyons];

console.log(tahoe.join(", ")); // Tallac, Ralston, Rose, Ward, Blackwood
```

All of the items from `peaks` and `canyons` are pushed into a new array called `tahoe`.

Let's take a look at how the spread operator can help us deal with a problem. Using the `peaks` array from the previous sample, let's imagine that we wanted to grab the last item from the array rather than the first. We could use the `Array.reverse` method to reverse the array in combination with array destructuring:

```
const peaks = ["Tallac", "Ralston", "Rose"];
const [last] = peaks.reverse();

console.log(last); // Rose
console.log(peaks.join(", ")); // Rose, Ralston, Tallac
```

See what happened? The `reverse` function has actually altered or mutated the array. In a world with the spread operator, we don't have to mutate the original array. Instead, we can create a copy of the array and then reverse it:

```
const peaks = ["Tallac", "Ralston", "Rose"];
const [last] = [...peaks].reverse();

console.log(last); // Rose
console.log(peaks.join(", ")); // Tallac, Ralston, Rose
```

Since we used the spread operator to copy the array, the `peaks` array is still intact and can be used later in its original form.

The spread operator can also be used to get the remaining items in the array:

```
const lakes = ["Donner", "Marlette", "Fallen Leaf", "Cascade"];

const [first, ...others] = lakes;

console.log(others.join(", ")); // Marlette, Fallen Leaf, Cascade
```

We can also use the three-dot syntax to collect function arguments as an array. When used in a function, these are called `rest parameters`. Here, we build a function that takes in *n* number of arguments using the spread operator, then uses those argumentsto print some console messages:

```
function directions(...args) {
  let [start, ...remaining] = args;
  let [finish, ...stops] = remaining.reverse();

  console.log(`drive through ${args.length} towns`);
  console.log(`start in ${start}`);
  console.log(`the destination is ${finish}`);
  console.log(`stopping ${stops.length} times in between`);
```

```
}

directions("Truckee", "Tahoe City", "Sunnyside", "Homewood", "Tahoma");
```

The `directions` function takes in the arguments using the spread operator. The first argument is assigned to the `start` variable. The last argument is assigned to a `finish`variable using `Array.reverse`. We then use the length of the `arguments` array to display how many towns we're going through. The number of stops is the length of the `arguments` array minus the `finish` stop. This provides incredible flexibility because we could use the `directions` function to handle any number of stops.

The spread operator can also be used for objects (see the GitHub page for Rest/ Spread Properties). Using the spread operator with objects is similar to using it with arrays. In this example, we'll use it the same way we combined two arrays into a third array, but instead of arrays, we'll use objects:

```
const morning = {
  breakfast: "oatmeal",
  lunch: "peanut butter and jelly"
};

const dinner = "mac and cheese";

const backpackingMeals = {
  ...morning,
  dinner
};

console.log(backpackingMeals);

// {
//   breakfast: "oatmeal",
//   lunch: "peanut butter and jelly",
//   dinner: "mac and cheese"
// }
```

Asynchronous JavaScript

The code samples that have been part of this chapter so far have been synchronous. When we write synchronous JavaScript code, we're providing a list of instructions that execute immediately in order. For example, if we wanted to use JavaScript to han-dle some simple DOM manipulation, we'd write the code to do so like this:

```
const header = document.getElementById("heading");
header.innerHTML = "Hey!";
```

These are instructions. "Yo, go select that element with an id of `heading`. Then when you're done with that, how about you set that inner HTML to *Hey*." It works synchro- nously. While each operation is happening, nothing else is happening.

With the modern web, we need to perform asynchronous tasks. These tasks often have to wait for some work to finish before they can be completed. We might need to access a database. We might need to stream video or audio content. We might need to fetch data from an API. With JavaScript, asynchronous tasks do not block the main thread. JavaScript is free to do something else while we wait for the API to return data. JavaScript has evolved a lot over the past few years to make handling these asyn- chronous actions easier. Let's explore some of the features that make this possible.

Simple Promises with Fetch

Making a request to a REST API used to be pretty cumbersome. We'd have to write 20+ lines of nested code just to load some data into our app. Then the `fetch()` func-tion showed up and simplified our lives. Thanks to the ECMAScript committee for making fetch happen.

Let's get some data from the randomuser.me API. This API has information like email address, name, phone number, location, and so on for fake members and is great to use as dummy data. `fetch` takes in the URL for this resource as its only parameter:

```
console.log(fetch("https://api.randomuser.me/?nat=US&results=1"));
```

When we log this, we see that there is a pending promise. *Promises* give us a way to make sense out of asynchronous behavior in JavaScript. The promise is an object that represents whether the async operation is pending, has been completed, or has failed. Think of this like the browser saying, "Hey, I'm going to try my best to go get this data. Either way, I'll come back and let you know how it went."

So back to the `fetch` result. The pending promise represents a state before the data has been fetched. We need to chain on a function called `.then()`. This function will take in a callback function that will run if the previous operation was successful. In other words, fetch some data, then do something else.

The something else we want to do is turn the response into JSON:

```
fetch("https://api.randomuser.me/?nat=US&results=1").then(res =>
  console.log(res.json())
);
```

The `then` method will invoke the callback function once the promise has resolved. Whatever you return from this function becomes the argument of the next `then` function. So we can chain together `then` functions to handle a promise that has been successfully resolved:

```
fetch("https://api.randomuser.me/?nat=US&results=1")
  .then(res => res.json())
  .then(json => json.results)
```

```
  .then(console.log)
  .catch(console.error);
```

First, we use `fetch` to make a GET request to randomuser.me. If the request is suc- cessful, we'll then convert the response body to JSON. Next, we'll take the JSON data and return the results, then we'll send the results to the `console.log` function, whichwill log them to the console. Finally, there is a `catch` function that invokes a callback if the `fetch` did not resolve successfully. Any error that occurred while fetching data from randomuser.me will be based on that callback. Here, we simply log the error to the console using `console.error`.

Async/Await

Another popular approach for handling promises is to create an async function.Some developers prefer the syntax of async functions because it looks more familiar, like code that's found in a synchronous function. Instead of waiting for the results ofa promise to resolve and handling it with a chain of `then` functions, `async` functions can be told to wait for the promise to resolve before further executing any code foundin the function.

Let's make another API request but wrap the functionality with an async function:

```
const getFakePerson = async () => {
  let res = await fetch("https://api.randomuser.me/?nat=US&results=1");
  let { results } = res.json();
  console.log(results);
};

getFakePerson();
```

Notice that the `getFakePerson` function is declared using the `async` keyword. This makes it an asynchronous function that can wait for promises to resolve before exe- cuting the code any further. The `await` keyword is used before promise calls. This tells the function to wait for the promise to resolve. This code accomplishes the exact same task as the code in the previous section that uses `then` functions. Well, almost the exact same task…

```
const getFakePerson = async () => {
  try {
    let res = await fetch("https://api.randomuser.me/?nat=US&results=1");
    let { results } = res.json();
    console.log(results);
  } catch (error) {
    console.error(error);
  }
};

getFakePerson();
```

There we go—now this code accomplishes the exact same task as the code in the pre- vious section that uses `then` functions. If the `fetch` call is successful, the results are logged to the console. If it's unsuccessful, then we'll log the error to the console using `console.error`. When using `async` and `await`, you need to surround your promise call in a `try...catch` block to handle any errors that may occur due to an unresolved promise.

Building Promises

When making an asynchronous request, one of two things can happen: everything goes as we hope, or there's an error. There can be many different types of successful or unsuccessful requests. For example, we could try several ways to obtain the data to reach success. We could also receive multiple types of errors. Promises give us a way to simplify back to a simple pass or fail.

The `getPeople` function returns a new promise. The promise makes a request to the API. If the promise is successful, the data will load. If the promise is unsuccessful, an error will occur:

```
const getPeople = count =>
  new Promise((resolves, rejects) => {
    const api = `https://api.randomuser.me/?nat=US&results=${count}`;
    const request = new XMLHttpRequest();
    request.open("GET", api);
    request.onload = () =>
      request.status === 200
        ? resolves(JSON.parse(request.response).results)
        : reject(Error(request.statusText));
    request.onerror = err => rejects(err);
    request.send();
  });
```

With that, the promise has been created, but it hasn't been used yet. We can use the promise by calling the `getPeople` function and passing in the number of members that should be loaded. The `then` function can be chained on to do something once the promise has been fulfilled. When a promise is rejected, any details are passed back to the `catch` function, or the `catch` block if using `async/await` syntax:

```
getPeople(5)
  .then(members => console.log(members))
  .catch(error => console.error(`getPeople failed: ${error.message}`))
);
```

Promises make dealing with asynchronous requests easier, which is good, because we have to deal with a lot of asynchronicity in JavaScript. A solid understanding of asyn- chronous behavior is essential for the modern JavaScript engineer.

Classes

Prior to ES2015, there was no official class syntax in the JavaScript spec. When classes were introduced, there was a lot of excitement about how similar the syntax of classes was to traditional object-oriented languages like Java and C++. The past few years saw the React library leaning on classes heavily to construct user interface compo- nents. Today, React is beginning to move away from classes, instead using functions to construct components. You'll still see classes all over the place, particularly in leg- acy React code and in the world of JavaScript, so let's take a quick look at them.

JavaScript uses something called `prototypical inheritance`. This technique can be wielded to create structures that feel object-oriented. For example, we can create a `Vacation` constructor that needs to be invoked with a `new` operator:

```
function Vacation(destination, length) {
  this.destination = destination;
  this.length = length;
}

Vacation.prototype.print = function() {
  console.log(this.destination + " | " + this.length + " days");
};

const maui = new Vacation("Maui", 7);

maui.print(); // Maui | 7 days
```

This code creates something that feels like a custom type in an object-oriented lan- guage. A `Vacation` has properties (destination, length), and it has a method (print). The `maui` instance inherits the `print` method through the prototype. If you are or were a developer accustomed to more standard classes, this might fill you with a deeprage. ES2015 introduced class declaration to quiet that rage, but the dirty secret is thatJavaScript still works the same way. Functions are objects, and inheritance is handled through the prototype. Classes provide a syntactic sugar on top of that gnarly proto- type syntax:

```
class Vacation {
  constructor(destination, length) {
    this.destination = destination;
    this.length = length;
  }

  print() {
    console.log(`${this.destination} will take ${this.length} days.`);
  }
}
```

When you're creating a class, the class name is typically capitalized. Once you've cre- ated the class, you can create a new instance of the class using the `new` keyword. Thenyou can call the custom method on the class:

```
const trip = new Vacation("Santiago, Chile", 7);

trip.print(); // Chile will take 7 days.
```

Now that a class object has been created, you can use it as many times as you'd like tocreate new vacation instances. Classes can also be extended. When a class is extended,the subclass inherits the properties and methods of the superclass. These properties and methods can be manipulated from here, but as a default, all will be inherited.

You can use `Vacation` as an abstract class to create different types of vacations. For instance, an `Expedition` can extend the `Vacation` class to include gear:

```
class Expedition extends Vacation {
  constructor(destination, length, gear) {
    super(destination, length);
    this.gear = gear;
  }

  print() {
    super.print();
    console.log(`Bring your ${this.gear.join(" and your ")}`);
  }
}
```

That's simple inheritance: the subclass inherits the properties of the superclass. By calling the `print` method of `Vacation`, we can append some new content onto what is printed in the `print` method of `Expedition`. Creating a new instance works the exact same way—create a variable and use the `new` keyword:

```
const trip = new Expedition("Mt. Whitney", 3, [
  "sunglasses",
  "prayer flags",
  "camera"
]);

trip.print();

// Mt. Whitney will take 3 days.
// Bring your sunglasses and your prayer flags and your camera
```

ES6 Modules

A JavaScript *module* is a piece of reusable code that can easily be incorporated into other JavaScript files without causing variable collisions. JavaScript modules are stored in separate files, one file per module. There are two options when creating and

exporting a module: you can export multiple JavaScript objects from a single module or one JavaScript object per module.

In *text-helpers.js*, two functions are exported:

```
export const print=(message) => log(message, new Date())

export const log=(message, timestamp) =>
  console.log(`${timestamp.toString()}: ${message}`)
```

`export` can be used to export any JavaScript type that will be consumed in another module. In this example, the `print` function and `log` function are being exported. Any other variables declared in *text-helpers.js* will be local to that module.

Modules can also export a single main variable. In these cases, you can use `export default`. For example, the *mt-freel.js* file can export a specific expedition:

```
export default new Expedition("Mt. Freel", 2, ["water", "snack"]);
```

`export default` can be used in place of `export` when you wish to export only one type. Again, both `export` and `export default` can be used on any JavaScript type: primitives, objects, arrays, and functions.

Modules can be consumed in other JavaScript files using the `import` statement. Mod- ules with multiple exports can take advantage of object destructuring. Modules that use `export default` are imported into a single variable:

```
import { print, log } from "./text-helpers";
import freel from "./mt-freel";

print("printing a message");
log("logging a message");

freel.print();
```

You can scope module variables locally under different variable names:

```
import { print as p, log as l } from "./text-helpers";

p("printing a message");
l("logging a message");
```

You can also import everything into a single variable using *:

```
import * as fns from './text-helpers`
```

This `import` and `export` syntax is not yet fully supported by all browsers or by Node. However, like any emerging JavaScript syntax, it's supported by Babel. This means you can use these statements in your source code and Babel will know where to find the modules you want to include in your compiled JavaScript.

CommonJS

CommonJS is the module pattern that's supported by all versions of Node (see the Node.js documentation on modules). You can still use these modules with Babel and webpack. With CommonJS, JavaScript objects are exported using `module.exports`.

For example, in CommonJS, we can export the `print` and `log` functions as an object:

```
const print(message) => log(message, new Date())

const log(message, timestamp) =>
console.log(`${timestamp.toString()}: ${message}`}

module.exports = {print, log}
```

CommonJS does not support an `import` statement. Instead, modules are imported with the `require` function:

```
const { log, print } = require("./txt-helpers");
```

JavaScript is indeed moving quickly and adapting to the increasing demands that engineers are placing on the language, and browsers are quickly implementing new features. For up-to-date compatibility information, see the ESNext compatibility table. Many of the features that are included in the latest JavaScript syntax are present because they support functional programming techniques. In functional JavaScript, we can think of our code as being a collection of functions that can be composed into applications. In the next chapter, we'll explore functional techniques in more detail and will discuss why you might want to use them.

CHAPTER 2

Functional Programming with JavaScript

When you start to explore React, you'll likely notice that the topic of functional pro- gramming comes up a lot. Functional techniques are being used more and more in JavaScript projects, particularly React projects.

It's likely that you've already written functional JavaScript code without thinking about it. If you've mapped or reduced an array, then you're already on your way to becoming a functional JavaScript programmer. Functional programming techniques are core not only to React but to many of the libraries in the React ecosystem as well.

If you're wondering where this functional trend came from, the answer is the 1930s, with the invention of *lambda calculus*, or λ-calculus.[1] Functions have been a part of calculus since it emerged in the 17th century. Functions can be sent to functions as arguments or returned from functions as results. More complex functions, called *higher-order functions*, can manipulate functions and use them as either arguments or results or both. In the 1930s, Alonzo Church was at Princeton experimenting with these higher-order functions when he invented lambda calculus.

In the late 1950s, John McCarthy took the concepts derived from λ-calculus and applied them to a new programming language called Lisp. Lisp implemented the con- cept of higher-order functions and functions as *first-class members* or *first-class citi- zens*. A function is considered a first-class member when it can be declared as a variable and sent to functions as an argument. These functions can even be returned from functions.

In this chapter, we're going to go over some of the key concepts of functional pro- gramming, and we'll cover how to implement functional techniques with JavaScript.

1 Dana S. Scott, "λ-Calculus: Then & Now".

What It Means to Be Functional

JavaScript supports functional programming because JavaScript functions are first- class citizens. This means that functions can do the same things that variables can do.The latest JavaScript syntax adds language improvements that can beef up your func- tional programming techniques, including arrow functions, promises, and the spread operator.

In JavaScript, functions can represent data in your application. You may have noticed that you can declare functions with the `var`, `let`, or `const` keywords the same wayyou can declare strings, numbers, or any other variables:

```
var log = function(message) {
  console.log(message);
};

log("In JavaScript, functions are variables");

// In JavaScript, functions are variables
```

We can write the same function using an arrow function. Functional programmers write a lot of small functions, and the arrow function syntax makes that much easier:

```
const log = message => {
  console.log(message);
};
```

Since functions are variables, we can add them to objects:

```
const obj = {
  message: "They can be added to objects like variables",
  log(message) {
    console.log(message);
  }
};

obj.log(obj.message);

// They can be added to objects like variables
```

Both of these statements do the same thing: they store a function in a variable called `log`. Additionally, the `const` keyword was used to declare the second function, which will prevent it from being overwritten.

We can also add functions to arrays in JavaScript:

```
const messages = [
  "They can be inserted into arrays",
  message => console.log(message),
  "like variables",
  message => console.log(message)
];
```

```
messages[1](messages[0]); // They can be inserted into arrays
messages[3](messages[2]); // like variables
```

Functions can be sent to other functions as arguments, just like other variables:

```
const insideFn = logger => {
  logger("They can be sent to other functions as arguments");
};

insideFn(message => console.log(message));

// They can be sent to other functions as arguments
```

They can also be returned from other functions, just like variables:

```
const createScream = function(logger) {
  return function(message) {
    logger(message.toUpperCase() + "!!!");
  };
};

const scream = createScream(message => console.log(message));

scream("functions can be returned from other functions");
scream("createScream returns a function");
scream("scream invokes that returned function");

// FUNCTIONS CAN BE RETURNED FROM OTHER FUNCTIONS!!!
// CREATESCREAM RETURNS A FUNCTION!!!
// SCREAM INVOKES THAT RETURNED FUNCTION!!!
```

The last two examples were of higher-order functions: functions that either take or return other functions. We could describe the same `createScream` higher-order func-tion with arrows:

```
const createScream = logger => message => {
  logger(message.toUpperCase() + "!!!");
};
```

If you see more than one arrow used during a function declaration, this means that you're using a higher-order function.

We can say that JavaScript supports functional programming because its functionsare first-class citizens. This means that functions are data. They can be saved, retrieved, or flow through your applications just like variables.

Imperative Versus Declarative

Functional programming is a part of a larger programming paradigm: *declarative pro- gramming*. Declarative programming is a style of programming where applications

are structured in a way that prioritizes describing *what* should happen over defining *how* it should happen.

In order to understand declarative programming, we'll contrast it with *imperative programming*, or a style of programming that's only concerned with how to achieve results with code. Let's consider a common task: making a string URL-friendly. Typi-cally, this can be accomplished by replacing all of the spaces in a string with hyphens, since spaces are not URL-friendly. First, let's examine an imperative approach to this task:

```
const string = "Restaurants in Hanalei";
const urlFriendly = "";

for (var i = 0; i < string.length; i++) {
  if (string[i] === " ") {
    urlFriendly += "-";
  } else {
    urlFriendly += string[i];
  }
}

console.log(urlFriendly); // "Restaurants-in-Hanalei"
```

In this example, we loop through every character in the string, replacing spaces as they occur. The structure of this program is only concerned with how such a task can be achieved. We use a `for` loop and an `if` statement and set values with an equality operator. Just looking at the code alone does not tell us much. Imperative programs require lots of comments in order to understand what's going on.

Now let's look at a declarative approach to the same problem:

```
const string = "Restaurants in Hanalei";
const urlFriendly = string.replace(/ /g, "-");

console.log(urlFriendly);
```

Here we are using `string.replace` along with a regular expression to replace all instances of spaces with hyphens. Using `string.replace` is a way of describing what's supposed to happen: spaces in the string should be replaced. The details of how spaces are dealt with are abstracted away inside the `replace` function. In a declarative program, the syntax itself describes what should happen, and the details of howthings happen are abstracted away.

Declarative programs are easy to reason about because the code itself describes what is happening. For example, read the syntax in the following sample. It details what happens after members are loaded from an API:

```
const loadAndMapMembers = compose(
  combineWith(sessionStorage, "members"),
  save(sessionStorage, "members"),
```

```
  scopeMembers(window),
  logMemberInfoToConsole,
  logFieldsToConsole("name.first"),
  countMembersBy("location.state"),
  prepStatesForMapping,
  save(sessionStorage, "map"),
  renderUSMap
);

getFakeMembers(100).then(loadAndMapMembers);
```

The declarative approach is more readable and, thus, easier to reason about. The details of how each of these functions is implemented are abstracted away. Those tinyfunctions are named well and combined in a way that describes how member data goes from being loaded to being saved and printed on a map, and this approach does not require many comments. Essentially, declarative programming produces applica-tions that are easier to reason about, and when it's easier to reason about an applica- tion, that application is easier to scale. Additional details about the declarative programming paradigm can be found at the Declarative Programming wiki.

Now, let's consider the task of building a document object model, or DOM. An imperative approach would be concerned with how the DOM is constructed:

```
const target = document.getElementById("target");
const wrapper = document.createElement("div");
const headline = document.createElement("h1");

wrapper.id = "welcome";
headline.innerText = "Hello World";

wrapper.appendChild(headline);
target.appendChild(wrapper);
```

This code is concerned with creating elements, setting elements, and adding them to the document. It would be very hard to make changes, add features, or scale 10,000 lines of code where the DOM is constructed imperatively.

Now let's take a look at how we can construct a DOM declaratively using a React component:

```
const { render } = ReactDOM;

const Welcome = () => (
  <div id="welcome">
    <h1>Hello World</h1>
  </div>
);

render(<Welcome />, document.getElementById("target"));
```

React is declarative. Here, the `Welcome` component describes the DOM that should be rendered. The `render` function uses the instructions declared in the component to build the DOM, abstracting away the details of how the DOM is to be rendered. We can clearly see that we want to render our `Welcome` component into the element with the ID of `target`.

Functional Concepts

Now that you've been introduced to functional programming and what it means to be "functional" or "declarative," we'll move on to introducing the core concepts of func- tional programming: immutability, purity, data transformation, higher-order func- tions, and recursion.

Immutability

To mutate is to change, so to be *immutable* is to be unchangeable. In a functional pro-gram, data is immutable. It never changes.

If you need to share your birth certificate with the public but want to redact orremove private information, you essentially have two choices: you can take a big Sharpie to your original birth certificate and cross out your private data, or you can find a copy machine. Finding a copy machine, making a copy of your birth certificate,and writing all over that copy with that big Sharpie would be preferable. This way you can have a redacted birth certificate to share and your original that's still intact.

This is how immutable data works in an application. Instead of changing the original data structures, we build changed copies of those data structures and use them instead.

To understand how immutability works, let's take a look at what it means to mutate data. Consider an object that represents the color `lawn`:

```
let color_lawn = {
  title: "lawn",
  color: "#00FF00",
  rating: 0
};
```

We could build a function that would rate colors and use that function to change the rating of the `color` object:

```
function rateColor(color, rating) {
  color.rating = rating;
  return color;
}

console.log(rateColor(color_lawn, 5).rating); // 5
console.log(color_lawn.rating); // 5
```

In JavaScript, function arguments are references to the actual data. Setting the color's rating like this changes or mutates the original color object. (Imagine if you tasked a business with redacting and sharing your birth certificate and they returned your original birth certificate with black marker covering the important details. You'd hope that a business would have the common sense to make a copy of your birth certificate and return the original unharmed.) We can rewrite the `rateColor` function so that it does not harm the original goods (the `color` object):

```
const rateColor = function(color, rating) {
  return Object.assign({}, color, { rating: rating });
};

console.log(rateColor(color_lawn, 5).rating); // 5
console.log(color_lawn.rating); // 0
```

Here, we used `Object.assign` to change the color rating. `Object.assign` is the copy machine. It takes a blank object, copies the color to that object, and overwrites the rating on the copy. Now we can have a newly rated color object without having to change the original.

We can write the same function using an arrow function along with the object spread operator. This `rateColor` function uses the spread operator to copy the color into a new object and then overwrite its rating:

```
const rateColor = (color, rating) => ({
  ...color,
  rating
});
```

This version of the `rateColor` function is exactly the same as the previous one. It treats color as an immutable object, does so with less syntax, and looks a little bit cleaner. Notice that we wrap the returned object in parentheses. With arrow func- tions, this is a required step since the arrow can't just point to an object's curly braces.

Let's consider an array of color names:

```
let list = [{ title: "Rad Red" }, { title: "Lawn" }, { title: "Party Pink" }];
```

We could create a function that will add colors to that array using `Array.push`:

```
const addColor = function(title, colors) {
  colors.push({ title: title });
  return colors;
};

console.log(addColor("Glam Green", list).length); // 4
console.log(list.length); // 4
```

However, `Array.push` is not an immutable function. This `addColor` function changes the original array by adding another field to it. In order to keep the `colors` array immutable, we must use `Array.concat` instead:

```
const addColor = (title, array) => array.concat({ title });

console.log(addColor("Glam Green", list).length); // 4
console.log(list.length); // 3
```

`Array.concat` concatenates arrays. In this case, it takes a new object with a new colortitle and adds it to a copy of the original array.

You can also use the spread operator to concatenate arrays in the same way it can be used to copy objects. Here's the emerging JavaScript equivalent of the previous `add Color` function:

```
const addColor = (title, list) => [...list, { title }];
```

This function copies the original list to a new array and then adds a new object con- taining the color's title to that copy. It is immutable.

Pure Functions

A *pure function* is a function that returns a value that's computed based on its argu- ments. Pure functions take at least one argument and always return a value or another function. They do not cause side effects, set global variables, or change any- thing about application state. They treat their arguments as immutable data.

In order to understand pure functions, let's first take a look at an impure function:

```
const frederick = {
  name: "Frederick Douglass",
  canRead: false,
  canWrite: false
};

function selfEducate() {
  frederick.canRead = true;
  frederick.canWrite = true;
  return frederick;
}

selfEducate();
console.log(frederick);

// {name: "Frederick Douglass", canRead: true, canWrite: true}
```

The `selfEducate` function is not a pure function. It does not take any arguments, andit does not return a value or a function. It also changes a variable outside of its scope: `Frederick`. Once the `selfEducate` function is invoked, something about the "world"has changed. It causes side effects:

```
const frederick = {
  name: "Frederick Douglass",
  canRead: false,
  canWrite:  false
};

const selfEducate = person => {
  person.canRead = true;
  person.canWrite = true;
  return person;
};

console.log(selfEducate(frederick));
console.log(frederick);

// {name: "Frederick Douglass", canRead: true, canWrite: true}
// {name: "Frederick Douglass", canRead: true, canWrite: true}
```

Pure Functions Are Testable

Pure functions are naturally *testable*. They do not change anything about their environment or "world," and therefore do not require a complicated test setup or teardown. Everything a pure function needs to operate it accesses via arguments. When testing a pure function, you control the arguments, and thus you can estimate the outcome. This `selfEducate` function is also impure: it causes side effects. Invoking this function mutates the objects that are sent to it. If we could treat the arguments sent to this function as immuta- ble data, then we would have a pure function.

Let's have this function take an argument:

```
const frederick = {
  name: "Frederick Douglass",
  canRead: false,
  canWrite:  false
};

const selfEducate = person => ({
  ...person,
  canRead: true,
  canWrite: true
});

console.log(selfEducate(frederick));
console.log(frederick);

// {name: "Frederick Douglass", canRead: true, canWrite: true}
// {name: "Frederick Douglass", canRead: false, canWrite: false}
```

Finally, this version of `selfEducate` is a pure function. It computes a value based on the argument that was sent to it: the `person`. It returns a new `person` object without mutating the argument sent to it and therefore has no side effects.

Now let's examine an impure function that mutates the DOM:

```
function Header(text) {
  let h1 = document.createElement("h1");
  h1.innerText = text;
  document.body.appendChild(h1);
}

Header("Header() caused side effects");
```

The `Header` function creates a heading—one element with specific text—and adds it to the DOM. This function is impure. It does not return a function or a value, and it causes side effects: a changed DOM.

In React, the UI is expressed with pure functions. In the following sample, `Header` is apure function that can be used to create `h1` elements just like in the previous example.However, this function on its own does not cause side effects because it does not mutate the DOM. This function will create an `h1` element, and it's up to some other part of the application to use that element to change the DOM:

```
const Header = props => <h1>{props.title}</h1>;
```

Pure functions are another core concept of functional programming. They will make your life much easier because they will not affect your application's state. When writ- ing functions, try to follow these three rules:

1. The function should take in at least one argument.
2. The function should return a value or another function.
3. The function should not change or mutate any of its arguments.

Data Transformations

How does anything change in an application if the data is immutable? Functional programming is all about transforming data from one form to another. We'll producetransformed copies using functions. These functions make our code less imperative and thus reduce complexity.

You do not need a special framework to understand how to produce one dataset that is based upon another. JavaScript already has the necessary tools for this task built into the language. There are two core functions that you must master in order to be proficient with functional JavaScript: `Array.map` and `Array.reduce`.

In this section, we'll take a look at how these and some other core functions trans- form data from one type to another.

Consider this array of high schools:

```
const schools = ["Yorktown", "Washington & Liberty", "Wakefield"];
```

We can get a comma-delimited list of these and some other strings by using the `Array.join` function:

```
console.log(schools.join(", "));

// "Yorktown, Washington & Liberty, Wakefield"
```

`Array.join` is a built-in JavaScript array method that we can use to extract a delimi- ted string from our array. The original array is still intact; `join` simply provides a dif- ferent take on it. The details of how this string is produced are abstracted away from the programmer.

If we wanted to create a function that creates a new array of the schools that begin with the letter "W," we could use the `Array.filter` method:

```
const wSchools = schools.filter(school => school[0] === "W");

console.log(wSchools);
// ["Washington & Liberty", "Wakefield"]
```

`Array.filter` is a built-in JavaScript function that produces a new array from a source array. This function takes a *predicate* as its only argument. A predicate is a function that always returns a Boolean value: `true` or `false`. `Array.filter` invokes this predicate once for every item in the array. That item is passed to the predicate as an argument, and the return value is used to decide if that item will be added to the new array. In this case, `Array.filter` is checking every school to see if its name begins with a "W."

When it's time to remove an item from an array, we should use `Array.filter` over `Array.pop` or `Array.splice` because `Array.filter` is immutable. In this next sample, the `cutSchool` function returns new arrays that filter out specific school names:

```
const cutSchool = (cut, list) => list.filter(school => school !== cut);

console.log(cutSchool("Washington & Liberty", schools).join(", "));

// "Yorktown, Wakefield"

console.log(schools.join("\n"));

// Yorktown
// Washington & Liberty
// Wakefield
```

In this case, the `cutSchool` function is used to return a new array that does not con- tain "Washington & Liberty." Then, the `join` function is used with this new array to create a string out of the remaining two school names. `cutSchool` is a pure function. It takes a list of schools and the name of the school that should be removed and returns a new array without that specific school.

Another array function that is essential to functional programming is `Array.map`. Instead of a predicate, the `Array.map` method takes a function as its argument. This function will be invoked once for every item in the array, and whatever it returns will be added to the new array:

```
const highSchools = schools.map(school => `${school} High School`);

console.log(highSchools.join("\n"));

// Yorktown High School
// Washington & Liberty High School
// Wakefield High School

console.log(schools.join("\n"));

// Yorktown
// Washington & Liberty
// Wakefield
```

In this case, the `map` function was used to append "High School" to each school name.The `schools` array is still intact.

In the last example, we produced an array of strings from an array of strings. The `map` function can produce an array of objects, values, arrays, other functions—any Java- Script type. Here's an example of the `map` function returning an object for every school:

```
const highSchools = schools.map(school => ({ name: school }));

console.log(highSchools);

// [
// { name: "Yorktown" },
// { name: "Washington & Liberty" },
// { name: "Wakefield" }
// ]
```

An array containing objects was produced from an array that contains strings.

If you need to create a pure function that changes one object in an array of objects, `map` can be used for this, too. In the following example, we'll change the school with the name of "Stratford" to "HB Woodlawn" without mutating the `schools` array:

```
let schools = [
  { name: "Yorktown" },
```

```
  { name: "Stratford" },
  { name: "Washington & Liberty" },
  { name: "Wakefield" }
];

let updatedSchools = editName("Stratford", "HB Woodlawn", schools);

console.log(updatedSchools[1]); // { name: "HB Woodlawn" }
console.log(schools[1]); // { name: "Stratford" }
```

The `schools` array is an array of objects. The `updatedSchools` variable calls the `editName` function and we send it the school we want to update, the new school, and the `schools` array. This changes the new array but makes no edits to the original:

```
const editName = (oldName, name, arr) =>
  arr.map(item => {
    if (item.name === oldName) {
      return {
        ...item,
        name
      };
    } else {
      return item;
    }
  });
```

Within `editName`, the `map` function is used to create a new array of objects based uponthe original array. The `editName` function can be written entirely in one line. Here's anexample of the same function using a shorthand `if/else` statement:

```
const editName = (oldName, name, arr) =>
  arr.map(item => (item.name === oldName ? { ...item, name } : item));
```

If you need to transform an array into an object, you can use `Array.map` in conjunc- tion with `Object.keys`. `Object.keys` is a method that can be used to return an arrayof keys from an object.

Let's say we needed to transform the `schools` object into an array of schools:

```
const schools = {
  Yorktown: 10,
  "Washington & Liberty": 2,
  Wakefield: 5
};

const schoolArray = Object.keys(schools).map(key => ({
  name: key,
  wins: schools[key]
}));

console.log(schoolArray);
```

```
// [
// {
// name: "Yorktown",
// wins: 10
// },
// {
// name: "Washington & Liberty",
// wins: 2
// },
// {
// name: "Wakefield",
// wins: 5
// }
// ]
```

In this example, `Object.keys` returns an array of school names, and we can use `map` on that array to produce a new array of the same length. The `name` of the new object will be set using the key, and `wins` is set equal to the value.

So far, we've learned that we can transform arrays with `Array.map` and `Array.filter`. We've also learned that we can change arrays into objects by combining `Object.keys` with `Array.map`. The final tool that we need in our functional arsenal is the ability to transform arrays into primitives and other objects.

The `reduce` and `reduceRight` functions can be used to transform an array into any value, including a number, string, boolean, object, or even a function.

Let's say we need to find the maximum number in an array of numbers. We need to transform an array into a number; therefore, we can use `reduce`:

```
const ages = [21, 18, 42, 40, 64, 63, 34];

const maxAge = ages.reduce((max, age) => {
  console.log(`${age} > ${max} = ${age > max}`);
  if (age > max) {
    return age;
  } else {
    return max;
  }
}, 0);

console.log("maxAge", maxAge);

// 21 > 0 = true
// 18 > 21 = false
// 42 > 21 = true
// 40 > 42 = false
// 64 > 42 = true
// 63 > 64 = false
// 34 > 64 = false
// maxAge 64
```

The `ages` array has been reduced into a single value: the maximum age, 64. `reduce` takes two arguments: a callback function and an original value. In this case, the origi- nal value is `0`, which sets the initial maximum value to `0`. The callback is invoked oncefor every item in the array. The first time this callback is invoked, `age` is equal to `21`, the first value in the array, and `max` is equal to `0`, the initial value. The callback returns the greater of the two numbers, `21`, and that becomes the `max` value during the next iteration. Each iteration compares each `age` against the `max` value and returns the greater of the two. Finally, the last number in the array is compared and returned from the previous callback.

If we remove the `console.log` statement from the preceding function and use a shorthand `if/else` statement, we can calculate the max value in any array of num- bers with the following syntax:

```
const max = ages.reduce((max, value) => (value > max ? value : max), 0);
```

Array.reduceRight

`Array.reduceRight` works the same way as `Array.reduce`; the dif- ference is that it starts reducing from the end of the array rather than the beginning.

Sometimes we need to transform an array into an object. The following example uses `reduce` to transform an array that contains colors into a hash:

```
const colors = [
  {
    id: "xekare",
    title: "rad red",
    rating: 3
  },
  {
    id: "jbwsof",
    title: "big blue",
    rating: 2
  },
  {
    id: "prigbj",
    title: "grizzly grey",
    rating: 5
  },
  {
    id: "ryhbhsl",
    title: "banana",
    rating: 1
  }
];

const hashColors = colors.reduce((hash, { id, title, rating }) => {
```

```
  hash[id] = { title, rating };
  return hash;
}, {});

console.log(hashColors);

// {
// "xekare": {
// title:"rad red",
// rating:3
// },
// "jbwsof": {
// title:"big blue",
// rating:2
// },
// "prigbj": {
// title:"grizzly grey",
// rating:5
// },
// "ryhbhsl": {
// title:"banana",
// rating:1
// }
// }
```

In this example, the second argument sent to the `reduce` function is an empty object. This is our initial value for the hash. During each iteration, the callback function addsa new key to the hash using bracket notation and sets the value for that key to the `id` field of the array. `Array.reduce` can be used in this way to reduce an array to a single value—in this case, an object.

We can even transform arrays into completely different arrays using `reduce`. Con- sider reducing an array with multiple instances of the same value to an array of unique values. The `reduce` method can be used to accomplish this task:

```
const colors = ["red", "red", "green", "blue", "green"];

const uniqueColors = colors.reduce(
  (unique, color) =>
    unique.indexOf(color) !== -1 ? unique : [...unique, color],
  []
);

console.log(uniqueColors);

// ["red", "green", "blue"]
```

In this example, the `colors` array is reduced to an array of distinct values. The secondargument sent to the `reduce` function is an empty array. This will be the initial value for `distinct`. When the `distinct` array does not already contain a specific color, it

will be added. Otherwise, it will be skipped, and the current `distinct` array will be returned.

`map` and `reduce` are the main weapons of any functional programmer, and JavaScript is no exception. If you want to be a proficient JavaScript engineer, then you must master these functions. The ability to create one dataset from another is a required skill and is useful for any type of programming paradigm.

Higher-Order Functions

The use of *higher-order functions* is also essential to functional programming. We've already mentioned higher-order functions, and we've even used a few in this chapter. Higher-order functions are functions that can manipulate other functions. They can take functions in as arguments or return functions or both.

The first category of higher-order functions are functions that expect other functions as arguments. `Array.map`, `Array.filter`, and `Array.reduce` all take functions asarguments. They are higher-order functions.

Let's take a look at how we can implement a higher-order function. In the following example, we create an `invokeIf` callback function that will test a condition and invoke a callback function when it's true and another callback function when the con-dition is false:

```
const invokeIf = (condition, fnTrue, fnFalse) =>
  condition ? fnTrue() : fnFalse();

const showWelcome = () => console.log("Welcome!!!");

const showUnauthorized = () => console.log("Unauthorized!!!");

invokeIf(true, showWelcome, showUnauthorized); // "Welcome!!!"
invokeIf(false, showWelcome, showUnauthorized); // "Unauthorized!!!"
```

`invokeIf` expects two functions: one for true and one for false. This is demonstrated by sending both `showWelcome` and `showUnauthorized` to `invokeIf`. When the condi- tion is true, `showWelcome` is invoked. When it's false, `showUnauthorized` is invoked.

Higher-order functions that return other functions can help us handle the complexi- ties associated with asynchronicity in JavaScript. They can help us create functions that can be used or reused at our convenience.

Currying is a functional technique that involves the use of higher-order functions.

The following is an example of currying. The `userLogs` function hangs on to some information (the username) and returns a function that can be used and reused whenthe rest of the information (the message) is made available. In this example, log

messages will all be prepended with the associated username. Notice that we're using the `getFakeMembers` function that returns a promise from Chapter 2:

```
const userLogs = userName => message =>
  console.log(`${userName} -> ${message}`);

const log = userLogs("grandpa23");

log("attempted to load 20 fake members");
getFakeMembers(20).then(
  members => log(`successfully loaded ${members.length} members`),
  error => log("encountered an error loading members")
);

// grandpa23 -> attempted to load 20 fake members
// grandpa23 -> successfully loaded 20 members

// grandpa23 -> attempted to load 20 fake members
// grandpa23 -> encountered an error loading members
```

`userLogs` is the higher-order function. The `log` function is produced from `userLogs`,and every time the `log` function is used, "grandpa23" is prepended to the message.

Recursion

Recursion is a technique that involves creating functions that recall themselves. Often, when faced with a challenge that involves a loop, a recursive function can be used instead. Consider the task of counting down from 10. We could create a `for`loop to solve this problem, or we could alternatively use a recursive function. In this example, `countdown` is the recursive function:

```
const countdown = (value, fn) => {
  fn(value);
  return value > 0 ? countdown(value - 1, fn) : value;
};

countdown(10, value => console.log(value));

// 10
// 9
// 8
// 7
// 6
// 5
// 4
// 3
// 2
// 1
// 0
```

`countdown` expects a number and a function as arguments. In this example, it's invoked with a value of `10` and a callback function. When `countdown` is invoked, the callback is invoked, which logs the current value. Next, `countdown` checks the value to see if it's greater than `0`. If it is, `countdown` recalls itself with a decremented value. Eventually, the value will be `0`, and `countdown` will return that value all the way back up the call stack.

Recursion is a pattern that works particularly well with asynchronous processes. Functions can recall themselves when they're ready, like when the data is available or when a timer has finished.

The `countdown` function can be modified to count down with a delay. This modified version of the `countdown` function can be used to create a countdown clock:

```
const countdown = (value, fn, delay = 1000) => {
  fn(value);
  return value > 0
    ? setTimeout(() => countdown(value - 1, fn, delay), delay)
    : value;
};

const log = value => console.log(value);
countdown(10, log);
```

In this example, we create a 10-second countdown by initially invoking `countdown` once with the number `10` in a function that logs the countdown. Instead of recalling itself right away, the `countdown` function waits one second before recalling itself, thus creating a clock.

Recursion is a good technique for searching data structures. You can use recursion to iterate through subfolders until a folder that contains only files is identified. You can also use recursion to iterate though the HTML DOM until you find an element that does not contain any children. In the next example, we'll use recursion to iterate deeply into an object to retrieve a nested value:

```
const dan = {
  type: "person",
  data: {
    gender: "male",
    info: {
      id: 22,
      fullname: {
        first: "Dan",
        last: "Deacon"
      }
    }
  }
};
```

```
deepPick("type", dan); // "person"
deepPick("data.info.fullname.first", dan); // "Dan"
```

`deepPick` can be used to access `Dan`'s type, stored immediately in the first object, or todig down into nested objects to locate `Dan`'s first name. Sending a string that uses dot notation, we can specify where to locate values that are nested deep within an object:

```
const deepPick = (fields, object = {}) => {
  const [first, ...remaining] = fields.split(".");
  return remaining.length
    ? deepPick(remaining.join("."), object[first])
    : object[first];
};
```

The `deepPick` function is either going to return a value or recall itself until it eventu- ally returns a value. First, this function splits the dot-notated fields string into anarray and uses array destructuring to separate the first value from the remaining val- ues. If there are remaining values, `deepPick` recalls itself with slightly different data, allowing it to dig one level deeper.

This function continues to call itself until the fields string no longer contains dots, meaning that there are no more remaining fields. In this sample, you can see how the values for `first`, `remaining`, and `object[first]` change as `deepPick` iterates through:

```
deepPick("data.info.fullname.first", dan); // "Dan"

// First Iteration
// first = "data"
// remaining.join(".") = "info.fullname.first"
// object[first] = { gender: "male", {info} }

// Second Iteration
// first = "info"
// remaining.join(".") = "fullname.first"
// object[first] = {id: 22, {fullname}}

// Third Iteration
// first = "fullname"
// remaining.join("." = "first"
// object[first] = {first: "Dan", last: "Deacon" }

// Finally...
// first = "first"
// remaining.length = 0
// object[first] = "Deacon"
```

Recursion is a powerful functional technique that's fun to implement.

Composition

Functional programs break up their logic into small, pure functions that are focused on specific tasks. Eventually, you'll need to put these smaller functions together. Specifically, you may need to combine them, call them in series or parallel, or com- pose them into larger functions until you eventually have an application.

When it comes to composition, there are a number of different implementations, pat- terns, and techniques. One that you may be familiar with is chaining. In JavaScript, functions can be chained together using dot notation to act on the return value of the previous function.

Strings have a replace method. The replace method returns a template string, which will also have a replace method. Therefore, we can chain together replace methods with dot notation to transform a string:

```
const template = "hh:mm:ss tt";
const clockTime = template
  .replace("hh", "03")
  .replace("mm", "33")
  .replace("ss", "33")
  .replace("tt", "PM");

console.log(clockTime);

// "03:33:33 PM"
```

In this example, the template is a string. By chaining replace methods to the end ofthe template string, we can replace hours, minutes, seconds, and time of day in the string with new values. The template itself remains intact and can be reused to create more clock time displays.

The `both` function is one function that pipes a value through two separate functions. The output of civilian hours becomes the input for `appendAMPM`, and we can change a date using both of these functions combined into one:

```
const both = date => appendAMPM(civilianHours(date));
```

However, this syntax is hard to comprehend and therefore tough to maintain or scale. What happens when we need to send a value through 20 different functions?

A more elegant approach is to create a higher-order function we can use to compose functions into larger functions:

```
const both = compose(
  civilianHours,
  appendAMPM
);

both(new Date());
```

This approach looks much better. It's easy to scale because we can add more functions at any point. This approach also makes it easy to change the order of the composed functions.

The `compose` function is a higher-order function. It takes functions as arguments and returns a single value:

```
const compose = (...fns) => arg =>
  fns.reduce((composed, f) => f(composed), arg);
```

`compose` takes in functions as arguments and returns a single function. In this imple- mentation, the spread operator is used to turn those function arguments into an array called `fns`. A function is then returned that expects one argument, `arg`. When this function is invoked, the `fns` array is piped starting with the argument we want to send through the function. The argument becomes the initial value for `compose`, then each iteration of the reduced callback returns. Notice that the callback takes two argu-ments: composed and a function `f`. Each function is invoked with `compose`, which is the result of the previous function's output. Eventually, the last function will be invoked and the last result returned.

This is a simple example of a `compose` function designed to illustrate composition techniques. This function becomes more complex when it's time to handle more than one argument or deal with arguments that are not functions.

Putting It All Together

Now that we've been introduced to the core concepts of functional programming, let's put those concepts to work for us and build a small JavaScript application.

Our challenge is to build a ticking clock. The clock needs to display hours, minutes, seconds, and time of day in civilian time. Each field must always have double digits, meaning leading zeros need to be applied to single-digit values like 1 or 2. The clock must also tick and change the display every second.

First, let's review an imperative solution for the clock:

```
// Log Clock Time every Second
setInterval(logClockTime, 1000);

function logClockTime() {
  // Get Time string as civilian time
  let time = getClockTime();

  // Clear the Console and log the time
  console.clear();
  console.log(time);
}

function getClockTime() {
```

```
    // Get the Current Time
    let date = new Date();
    let time = "";

    // Serialize clock time
    let time = {
      hours: date.getHours(),
      minutes: date.getMinutes(),
      seconds: date.getSeconds(),
      ampm: "AM"
    };

    // Convert to civilian time
    if (time.hours == 12) {
      time.ampm = "PM";
    } else if (time.hours > 12) {
      time.ampm = "PM";
      time.hours -= 12;
    }

    // Prepend a 0 on the hours to make double digits
    if (time.hours < 10) {
      time.hours = "0" + time.hours;
    }

    // prepend a 0 on the minutes to make double digits
    if (time.minutes < 10) {
      time.minutes = "0" + time.minutes;
    }

    // prepend a 0 on the seconds to make double digits
    if (time.seconds < 10) {
      time.seconds = "0" + time.seconds;
    }

    // Format the clock time as a string "hh:mm:ss tt"
    return time.hours + ":" + time.minutes + ":" + time.seconds + " " + time.ampm;
  }
```

This solution works, and the comments help us understand what's happening. How- ever, these functions are large and complicated. Each function does a lot. They're hard to comprehend, they require comments, and they're tough to maintain. Let's see how a functional approach can produce a more scalable application.

Our goal will be to break the application logic up into smaller parts: functions. Each function will be focused on a single task, and we'll compose them into larger func- tions that we can use to create the clock.

First, let's create some functions that give us values and manage the console. We'll need a function that gives us one second, a function that gives us the current time, and a couple of functions that will log messages on a console and clear the console. In

functional programs, we should use functions over values wherever possible. We'llinvoke the function to obtain the value when needed:

```
const oneSecond = () => 1000;
const getCurrentTime = () => new Date();
const clear = () => console.clear();
const log = message => console.log(message);
```

Next, we'll need some functions for transforming data. These three functions will beused to mutate the `Date` object into an object that can be used for our clock:

`serializeClockTime`
: Takes a date object and returns an object for clock time that contains hours,minutes, and seconds.

`civilianHours`
: Takes the clock time object and returns an object where hours are converted tocivilian time. For example: 1300 becomes 1:00.

`appendAMPM`
: Takes the clock time object and appends time of day (AM or PM) to that object.

```
const serializeClockTime = date => ({
  hours: date.getHours(),
  minutes: date.getMinutes(),
  seconds: date.getSeconds()
});

const civilianHours = clockTime => ({
  ...clockTime,
  hours: clockTime.hours > 12 ? clockTime.hours - 12 : clockTime.hours
});

const appendAMPM = clockTime => ({
  ...clockTime,
  ampm: clockTime.hours >= 12 ? "PM" : "AM"
});
```

These three functions are used to transform data without changing the original. Theytreat their arguments as immutable objects.

Next, we'll need a few higher-order functions:

`display`
: Takes a target function and returns a function that will send a time to the target.In this example, the target will be `console.log`.

`formatClock`
: Takes a template string and uses it to return clock time formatted based on thecriteria from the string. In this example, the template is "hh:mm:ss tt". From

there, `formatClock` will replace the placeholders with hours, minutes, seconds, and time of day.

`prependZero`
: Takes an object's key as an argument and prepends a zero to the value stored under that object's key. It takes in a key to a specific field and prepends values with a zero if the value is less than 10.

```
const display = target => time => target(time);

const formatClock = format => time =>
  format
    .replace("hh", time.hours)
    .replace("mm", time.minutes)
    .replace("ss", time.seconds)
    .replace("tt", time.ampm);

const prependZero = key => clockTime => ({
  ...clockTime,
  key: clockTime[key] < 10 ? "0" + clockTime[key] : clockTime[key]
});
```

These higher-order functions will be invoked to create the functions that will be reused to format the clock time for every tick. Both `formatClock` and `prependZero` will be invoked once, initially setting up the required template or key. The inner func- tions they return will be invoked once every second to format the time for display.

Now that we have all of the functions required to build a ticking clock, we'll need to compose them. We'll use the `compose` function that we defined in the last section to handle composition:

`convertToCivilianTime`
: A single function that takes clock time as an argument and transforms it into civilian time by using both civilian hours.

`doubleDigits`
: A single function that takes civilian clock time and makes sure the hours, minutes, and seconds display double digits by prepending zeros where needed.

`startTicking`
: Starts the clock by setting an interval that invokes a callback every second. The callback is composed using all our functions. Every second the console is cleared, `currentTime` is obtained, converted, civilianized, formatted, and displayed.

```
const convertToCivilianTime = clockTime =>
  compose(
    appendAMPM,
    civilianHours
  )(clockTime);
```

```
const doubleDigits = civilianTime =>
  compose(
    prependZero("hours"),
    prependZero("minutes"),
    prependZero("seconds")
  )(civilianTime);

const startTicking = () =>
  setInterval(
    compose(
      clear,
      getCurrentTime,
      serializeClockTime,
      convertToCivilianTime,
      doubleDigits,
      formatClock("hh:mm:ss tt"),
      display(log)
    ),
    oneSecond()
  );

startTicking();
```

This declarative version of the clock achieves the same results as the imperative ver- sion. However, there quite a few benefits to this approach. First, all of these functions are easily testable and reusable. They can be used in future clocks or other digital dis- plays. Also, this program is easily scalable. There are no side effects. There are no global variables outside of functions themselves. There could still be bugs, but they'll be easier to find.

In this chapter, we've introduced functional programming principles. Throughout thebook when we discuss best practices in React, we'll continue to demonstrate how many React concepts are based in functional techniques. In the next chapter, we'll dive into React officially with an improved understanding of the principles that gui- ded its development.