This is a fast-paced guide that will help you to write real-world applications by utilizing a wide range of functional techniques and styles.

The book first explores the core concepts of functional programming common to all functional languages, with examples of their use in JavaScript. It’s followed by a comprehensive roundup of functional programming libraries for JavaScript that minimizes the burden of digging deep into JavaScript to expose a set of tools that makes functional programming not just possible but highly convenient. The book then rounds off with an overview of methods to effectively use and mix functional programming with object-oriented programming.

Who this book is written for
If you are a JavaScript developer interested in learning functional programming, looking for the quantum leap towards mastering the JavaScript language, or just want to become a better programmer in general, then this book is ideal for you. It is aimed at programmers involved in developing reactive frontend apps, server-side apps that wrangle with reliability and concurrency, and everything in between.

What you will learn from this book
- Separate core logic from the program state to write more maintainable code
- Replace ugly for loops with pure functions and recursion
- Simplify code with method chains of pure, higher-order functions
- Create more reliable code with closures and immutable data
- Explore lazy evaluation strategies and what they can accomplish
- Develop more powerful applications with currying and function composition
- Use promises, functors, monads, and function factories

Unlock the powers of functional programming hidden within JavaScript to build smarter, cleaner, and more reliable web apps.

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 4 'Implementing Functional Programming Techniques in JavaScript'
- A synopsis of the book’s content
- More information on Functional Programming in JavaScript

About the Author

Dan Mantyla works as a web application developer for the University of Kansas. He enjoys contributing to open source web frameworks and wrenching on motorcycles. Dan is currently living in Lawrence, Kansas, USA—the birthplace of Python Django and home to Linux News Media.

Dan has also clicked the cover image, which was taken outside his home in Lawrence, Kansas, USA, where the sunflower fields are in bloom for only one short week in September.
Functional Programming in JavaScript

Functional programming is a style that emphasizes and enables the writing of smarter code, which minimizes complexity and increases modularity. It's a way of writing cleaner code through clever ways of mutating, combining, and using functions. JavaScript provides an excellent medium for this approach. JavaScript, the Internet's scripting language, is actually a functional language at heart. By learning how to expose its true identity as a functional language, we can implement web applications that are powerful, easier to maintain, and more reliable. By doing this, JavaScript's odd quirks and pitfalls will suddenly become clear and the language as a whole will make infinitely more sense. Learning how to use functional programming will make you a better programmer for life.

This book is a guide for both new and experienced JavaScript developers who are interested in learning functional programming. With a focus on the progression of functional programming techniques, styles, and detailed information about JavaScript libraries, this book will help you to write smarter code and become a better programmer.

What This Book Covers

Chapter 1, The Powers of JavaScript's Functional Side – a Demonstration, sets the pace of the book by creating a small web application with the help of both traditional methods and functional programming. It then compares these two methods to underline the importance of functional programming.

Chapter 2, Fundamentals of Functional Programming, introduces you to the core concepts of functional programming as well as built-in JavaScript functions.

Chapter 3, Setting Up the Functional Programming Environment, explores different JavaScript libraries and how they can be optimized for functional programming.

Chapter 4, Implementing Functional Programming Techniques in JavaScript, explains the functional paradigm in JavaScript. It covers several styles of functional programming and demonstrates how they can be employed in different scenarios.

Chapter 5, Category Theory, explains the concept of Category Theory in detail and then implements it in JavaScript.

Chapter 6, Advanced Topics and Pitfalls in JavaScript, highlights various drawbacks you may face while programming in JavaScript, and the various ways to successfully deal with them.
Chapter 7, *Functional and Object-oriented Programming in JavaScript*, relates both functional and object-oriented programming to JavaScript, and shows you how the two paradigms can complement each other and coexist side by side.


Appendix B, *Glossary of Terms*, includes a glossary of terms used throughout the book.
Hold on to your hats because we're really going to get into the functional mind-set now.

In this chapter, we're going to do the following:

• Put all the core concepts together into a cohesive paradigm
• Explore the beauty that functional programming has to offer when we fully commit to the style
• Step through the logical progression of functional patterns as they build upon each other
• All the while, we will build up a simple application that does some pretty cool stuff

You may have noticed a few concepts that were brought up in the last chapter when dealing with functional libraries for JavaScript, but not in Chapter 2, Fundamentals of Functional Programming. Well, that was for a reason! Compositions, currying, partial application, and more. Let's explore why and how these libraries implemented those concepts.

Functional programming can come in a variety of flavors and patterns. This chapter will cover many different styles of functional programming:

• Data generic programming
• Mostly functional programming
• Functional reactive programming and more
This chapter, however, will be as style-unbiased as possible. Without leaning too hard on one style of functional programming over another, the overall goal is to show that there are better ways to write code than what is often accepted as the correct and only way. Once you free your mind about the preconceptions of what is the right way and what is not the right way to write code, you can do whatever you want. When you just write code with childlike abandon for no reason other than the fact that you like it and when you’re not concerned about conforming to the traditional way of doing things, then the possibilities are endless.

**Partial function application and currying**

Many languages support optional arguments, but not in JavaScript. JavaScript uses a different pattern entirely that allows for any number of arguments to be passed to a function. This leaves the door open for some very interesting and unusual design patterns. Functions can be applied in part or in whole.

Partial application in JavaScript is the process of binding values to one or more arguments of a function that returns another function that accepts the remaining, unbound arguments. Similarly, currying is the process of transforming a function with many arguments into a function with one argument that returns another function that takes more arguments as needed.

The difference between the two may not be clear now, but it will be obvious in the end.

**Function manipulation**

Actually, before we go any further and explain just how to implement partial application and currying, we need a review. If we’re going to tear JavaScript's thick veneer of C-like syntax right off and expose it's functional underbelly, then we’re going to need to understand how primitives, functions, and prototypes in JavaScript work; we would never need to consider these if we just wanted to set some cookies or validate some form fields.

**Apply, call, and the this keyword**

In pure functional languages, functions are not invoked; they're applied. JavaScript works the same way and even provides utilities for manually calling and applying functions. And it's all about the this keyword, which, of course, is the object that the function is a member of.
The `call()` function lets you define the `this` keyword as the first argument. It works as follows:

```javascript
console.log(['Hello', 'world'].join(' ')) // normal way
console.log(Array.prototype.join.call(['Hello', 'world'], ' ')); // using call
```

The `call()` function can be used, for example, to invoke anonymous functions:

```javascript
console.log((function(){console.log(this.length)}).call([1,2,3]));
```

The `apply()` function is very similar to the `call()` function, but a little more useful:

```javascript
console.log(Math.max(1,2,3)); // returns 3
console.log(Math.max([1,2,3])); // won't work for arrays though
console.log(Math.max.apply(null, [1,2,3])); // but this will work
```

The fundamental difference is that, while the `call()` function accepts a list of arguments, the `apply()` function accepts an array of arguments.

The `call()` and `apply()` functions allow you to write a function once and then inherit it in other objects without writing the function over again. And they are both members themselves of the `Function` argument.

This is bonus material, but when you use the `call()` function on itself, some really cool things can happen:

```javascript
// these two lines are equivalent
func.call(thisValue);
Function.prototype.call.call(func, thisValue);
```

### Binding arguments

The `bind()` function allows you to apply a method to one object with the `this` keyword assigned to another. Internally, it's the same as the `call()` function, but it's chained to the method and returns a new bounded function.

It's especially useful for callbacks, as shown in the following code snippet:

```javascript
function Drum(){
    this.noise = 'boom';
    this.duration = 1000;
    this.goBoom = function(){console.log(this.noise)};
}
var drum = new Drum();
setInterval(drum.goBoom.bind(drum), drum.duration);
```
This solves a lot of problems in object-oriented frameworks, such as Dojo, specifically the problems of maintaining the state when using classes that define their own handler functions. But we can use the `bind()` function for functional programming too.

The `bind()` function actually does partial application on its own, though in a very limited way.

**Function factories**

Remember our section on closures in Chapter 2, *Fundamentals of Functional Programming*? Closures are the constructs that makes it possible to create a useful JavaScript programming pattern known as function factories. They allow us to manually bind arguments to functions.

First, we'll need a function that binds an argument to another function:

```javascript
function bindFirstArg(func, a) {
    return function(b) {
        return func(a, b);
    };
}
```

Then we can use this to create more generic functions:

```javascript
var powersOfTwo = bindFirstArg(Math.pow, 2);
console.log(powersOfTwo(3)); // 8
console.log(powersOfTwo(5)); // 32
```

And it can work on the other argument too:

```javascript
function bindSecondArg(func, b) {
    return function(a) {
        return func(a, b);
    };
}
var squareOf = bindSecondArg(Math.pow, 2);
var cubeOf = bindSecondArg(Math.pow, 3);
console.log(squareOf(3)); // 9
console.log(squareOf(4)); // 16
console.log(cubeOf(3)); // 27
console.log(cubeOf(4)); // 64
```
The ability to create generic functions is very important in functional programming. But there's a clever trick to making this process even more generalized. The `bindFirstArg()` function itself takes two arguments, the first being a function. If we pass the `bindFirstArg` function as a function to itself, we can create bindable functions. This can be best described with the following example:

```javascript
var makePowersOf = bindFirstArg(bindFirstArg, Math.pow);
var powersOfThree = makePowersOf(3);
console.log(powersOfThree(2)); // 9
console.log(powersOfThree(3)); // 27
```

This is why they're called function factories.

**Partial application**

Notice that our function factory example's `bindFirstArg()` and `bindSecondArg()` functions only work for functions that have exactly two arguments. We could write new ones that work for different numbers of arguments, but that would work away from our model of generalization.

What we need is partial application.

> Partial application is the process of binding values to one or more arguments of a function that returns a partially-applied function that accepts the remaining, unbound arguments.

Unlike the `bind()` function and other built-in methods of the `Function` object, we'll have to create our own functions for partial application and currying. There are two distinct ways to do this.

- As a stand-alone function, that is, `var partial = function(func){...}
- As a *polyfill*, that is, `Function.prototype.partial = function(){...`

Polyfills are used to augment prototypes with new functions and will allow us to call our new functions as methods of the function that we want to partially apply. Just like this: `myfunction.partial(arg1, arg2, ...)`;
Partial application from the left

Here's where JavaScript's apply() and call() utilities become useful for us. Let's look at a possible polyfill for the Function object:

```javascript
Function.prototype.partialApply = function(){
  var func = this;
  args = Array.prototype.slice.call(arguments);
  return function(){
    return func.apply(this, args.concat(
      Array.prototype.slice.call(arguments)
    ));
  };
};
```

As you can see, it works by slicing the arguments special variable.

Every function has a special local variable called arguments that is an array-like object of the arguments passed to it. It's technically not an array. Therefore it does not have any of the Array methods such as slice and forEach. That's why we need to use Array's slice.call method to slice the arguments.

And now let's see what happens when we use it in an example. This time, let's get away from the math and go for something a little more useful. We'll create a little application that converts numbers to hexadecimal values.

```javascript
function nums2hex() {
  function componentToHex(component) {
    var hex = component.toString(16);
    // make sure the return value is 2 digits, i.e. 0c or 12
    if (hex.length == 1) {
      return "0" + hex;
    } else {
      return hex;
    }
  }
  return Array.prototype.map.call(arguments, componentToHex).join('');
}

// the function works on any number of inputs
console.log(nums2hex()); // ''
console.log(nums2hex(100,200)); // '64c8'
```
console.log(nums2hex(100, 200, 255, 0, 123)); // '64c800ff7b'

// but we can use the partial function to partially apply
// arguments, such as the OUI of a mac address
var myOUI = 123;
var getMacAddress = nums2hex.partialApply(myOUI);
console.log(getMacAddress()); // '7b'
console.log(getMacAddress(100, 200, 2, 123, 66, 0, 1));
// '7b64c8027b420001'

// or we can convert rgb values of red only to hexadecimal
var shadesOfRed = nums2hex.partialApply(255);
console.log(shadesOfRed(123, 0)); // 'ff7b00'
console.log(shadesOfRed(100, 200)); // 'ff64c8'

This example shows that we can partially apply arguments to a generic function and
get a new function in return. *This first example is left-to-right*, which means that we can
only partially apply the first, left-most arguments.

**Partial application from the right**

In order to apply arguments from the right, we can define another polyfill.

```javascript
Function.prototype.partialApplyRight = function(){
    var func = this;
    args = Array.prototype.slice.call(arguments);
    return function(){
        return func.apply(
            this,
            [].slice.call(arguments, 0)
                .concat(args));
    };
};
```

var shadesOfBlue = nums2hex.partialApplyRight(255);
console.log(shadesOfBlue(123, 0)); // '7b00ff'
console.log(shadesOfBlue(100, 200)); // '64c8ff'

var someShadesOfGreen = nums2hex.partialApplyRight(255, 0);
console.log(shadesOfGreen(123)); // '7bff00'
console.log(shadesOfGreen(100)); // '64ff00'
Partial application has allowed us to take a very generic function and extract more specific functions out of it. But the biggest flaw in this method is that the way in which the arguments are passed, as in how many and in what order, can be ambiguous. And ambiguity is never a good thing in programming. There's a better way to do this: currying.

Currying

Currying is the process of transforming a function with many arguments into a function with one argument that returns another function that takes more arguments as needed. Formally, a function with N arguments can be transformed into a function chain of N functions, each with only one argument.

A common question is: what is the difference between partial application and currying? While it's true that partial application returns a value right away and currying only returns another curried function that takes the next argument, the fundamental difference is that currying allows for much better control of how arguments are passed to the function. We'll see just how that's true, but first we need to create function to perform the currying.

Here's our polyfill for adding currying to the Function prototype:

```javascript
Function.prototype.curry = function (numArgs) {
  var func = this;
  numArgs = numArgs || func.length;

  // recursively acquire the arguments
  function subCurry(prev) {
    return function (arg) {
      var args = prev.concat(arg);
      if (args.length < numArgs) {
        // recursive case: we still need more args
        return subCurry(args);
      }
      else {
        // base case: apply the function
        return func.apply(this, args);
      }
    };
  }
  return subCurry([]);
};
```

The `numArgs` argument lets us optionally specify the number of arguments the function being curried needs if it's not explicitly defined.
Let's look at how to use it within our hexadecimal application. We'll write a function that converts RGB values to a hexadecimal string that is appropriate for HTML:

```javascript
function rgb2hex(r, g, b) {
  // nums2hex is previously defined in this chapter
  return '#' + nums2hex(r) + nums2hex(g) + nums2hex(b);
}
var hexColors = rgb2hex.curry();
console.log(hexColors(11)); // returns a curried function
console.log(hexColors(11,12,123)); // returns a curried function
console.log(hexColors(11)(12)(123)); // returns #0b0c7b
console.log(hexColors(210)(12)(0));  // returns #d20c00
```

It will return the curried function until all needed arguments are passed in. And they're passed in the same left-to-right order as defined by the function being curried.

But we can step it up a notch and define the more specific functions that we need as follows:

```javascript
var reds = function(g,b){return hexColors(255)(g)(b)};
var greens = function(r,b){return hexColors(r)(255)(b)};
var blues  = function(r,g){return hexColors(r)(g)(255)};
console.log(reds(11, 12)); // returns #ff0b0c
console.log(greens(11, 12)); // returns #0bff0c
console.log(blues(11, 12)); // returns #0b0cff
```

So that's a nice way to use currying. But if we just want to curry our `nums2hex()` function directly, we run into a little bit of trouble. And that's because the function doesn't define any arguments, it just lets you pass as many arguments in as you want. So we have to define the number of arguments. We do that with the optional parameter to the curry function that allows us to set the number of arguments of the function being curried.

```javascript
var hexs = nums2hex.curry(2);
console.log(hexs(11)(12)); // returns 0b0c
console.log(hexs(11)); // returns function
console.log(hexs(110)(12)(0)); // incorrect
```

Therefore currying does not work well with functions that accept variable numbers of arguments. For something like that, partial application is preferred.

All of this isn't just for the benefit of function factories and code reuse. Currying and partial application play into a bigger pattern known as composition.
Function composition
Finally, we have arrived at function composition.

In functional programming, we want everything to be a function. We especially want unary functions if possible. If we can convert all functions to unary functions, then magical things can happen.

Unary functions are functions that take only a single input. Functions with multiple inputs are polyadic, but we usually say binary for functions that accept two inputs and ternary for three inputs. Some functions don't accept a specific number of inputs; we call those variadic.

Manipulating functions and their acceptable number of inputs can be extremely expressive. In this section, we will explore how to compose new functions from smaller functions: little units of logic that combine into whole programs that are greater than the sum of the functions on their own.

Compose
Composing functions allows us to build complex functions from many simple, generic functions. By treating functions as building blocks for other functions, we can build truly modular applications with excellent readability and maintainability.

Before we define the compose() polyfill, you can see how it all works with these following examples:

```javascript
var roundedSqrt = Math.round.compose(Math.sqrt);
console.log( roundedSqrt(5) ); // Returns: 2

var squaredDate = roundedSqrt.compose(Date.parse);
console.log( squaredDate("January 1, 2014") ); // Returns: 1178370
```

In math, the composition of the $f$ and $g$ variables is defined as $f(g(x))$.

In JavaScript, this can be written as:

```javascript
var compose = function(f, g) {
  return function(x) {
    return f(g(x));
  };
};
```
But if we left it at that, we would lose track of the *this* keyword, among other problems. The solution is to use the `apply()` and `call()` utilities. Compared to `curry`, the `compose()` polyfill is quite simple.

```javascript
Function.prototype.compose = function(prevFunc) {
    var nextFunc = this;
    return function() {
        return nextFunc.call(this, prevFunc.apply(this, arguments));
    }
}
```

To show how it's used, let's build a completely contrived example, as follows:

```javascript
function function1(a) { return a + ' 1'; }
function function2(b) { return b + ' 2'; }
function function3(c) { return c + ' 3'; }
var composition = function3.compose(function2).compose(function1);
console.log(composition('count')); // returns 'count 1 2 3'
```

Did you notice that the `function3` parameter was applied first? This is very important. Functions are applied from right to left.

### Sequence – compose in reverse

Because many people like to read things from the left to the right, it might make sense to apply the functions in that order too. We'll call this a sequence instead of a composition.

To reverse the order, all we need to do is swap the `nextFunc` and `prevFunc` parameters.

```javascript
Function.prototype.sequence = function(prevFunc) {
    var nextFunc = this;
    return function() {
        return prevFunc.call(this, nextFunc.apply(this, arguments));
    }
}
```

This allows us to now call the functions in a more natural order.

```javascript
var sequences = function1.sequence(function2).sequence(function3);
console.log(sequences('count')); // returns 'count 1 2 3'
```
Implementing Functional Programming Techniques in JavaScript

Compositions versus chains

Here are five different implementations of the same floorSqrt() functional composition. They seem to be identical, but they deserve scrutiny.

```javascript
function floorSqrt1(num) {
    var sqrtNum = Math.sqrt(num);
    var floorSqrt = Math.floor(sqrtNum);
    var stringNum = String(floorSqrt);
    return stringNum;
}

function floorSqrt2(num) {
    return String(Math.floor(Math.sqrt(num)));
}

function floorSqrt3(num) {
    return [num].map(Math.sqrt).map(Math.floor).toString();
}

var floorSqrt4 = String.compose(Math.floor).compose(Math.sqrt);
var floorSqrt5 = Math.sqrt.sequence(Math.floor).sequence(String);

// all functions can be called like this:
floorSqrt<N>(17); // Returns: 4
```

But there are a few key differences we should go over:

- Obviously the first method is verbose and inefficient.
- The second method is a nice one-liner, but this approach becomes very unreadable after only a few functions are applied.

To say that less code is better is missing the point. Code is more maintainable when the effective instructions are more concise. If you reduce the number of characters on the screen without changing the effective instructions carried out, this has the complete opposite effect—code becomes harder to understand, and decidedly less maintainable; for example, when we use nested ternary operators, or we chain several commands together on a single line. These approaches reduce the amount of `code on the screen`, but they don't reduce the number of steps actually being specified by that code. So the effect is to obfuscate and make the code harder to understand. The kind of conciseness that makes code easier to maintain is that which effectively reduces the specified instructions (for example, by using a simpler algorithm that accomplishes the same result with fewer and/or simpler steps), or when we simply replace code with a message, for instance, invoking a third-party library with a well-documented API.
• The third approach is a chain of array functions, notably the `map` function. This works fairly well, but it is not mathematically correct.

• Here's our `compose()` function in action. All methods are forced to be unary, pure functions that encourage the use of better, simpler, and smaller functions that do one thing and do it well.

• The last approach uses the `compose()` function in reverse sequence, which is just as valid.

Programming with compose

The most important aspect of compose is that, aside from the first function that is applied, it works best with pure, unary functions: functions that take only one argument.

The output of the first function that is applied is sent to the next function. This means that the function must accept what the previous function passed to it. This is the main influence behind type signatures.

Type Signatures are used to explicitly declare what types of input the function accepts and what type it outputs. They were first used by Haskell, which actually used them in the function definitions to be used by the compiler. But, in JavaScript, we just put them in a code comment. They look something like this:

```
foo :: arg1 -> argN -> output
```

Examples:

```
// getStringLength :: String -> Int
function getStringLength(s){return s.length};

// concatDates :: Date -> Date -> [Date]
function concatDates(d1,d2){return [d1, d2]};

// pureFunc :: (int -> Bool) -> [int] -> [int]
pureFunc(func, arr){return arr.filter(func)}
```

In order to truly reap the benefits of compose, any application will need a hefty collection of unary, pure functions. These are the building blocks that are composed into larger functions that, in turn, are used to make applications that are very modular, reliable, and maintainable.

Let's go through an example. First we'll need many building-block functions. Some of them build upon the others as follows:

```
// stringToArray :: String -> [Char]
function stringToArray(s) { return s.split(''); }

// arrayToString :: [Char] -> String
```
Implementing Functional Programming Techniques in JavaScript

```javascript
function arrayToString(a) { return a.join(''); }

// nextChar :: Char -> Char
function nextChar(c) {
  return String.fromCharCode(c.charCodeAt(0) + 1); }

// previousChar :: Char -> Char
function previousChar(c) {
  return String.fromCharCode(c.charCodeAt(0) - 1); }

// higherColorHex :: Char -> Char
function higherColorHex(c) {
  return c >= 'f' ? 'f' :
          c == '9' ? 'a' :
          nextChar(c);
}

// lowerColorHex :: Char -> Char
function lowerColorHex(c) {
  return c <= '0' ? '0' :
          c == 'a' ? '9' :
          previousChar(c);
}

// raiseColorHexes :: String -> String
function raiseColorHexes(arr) { return arr.map(higherColorHex); }

// lowerColorHexes :: String -> String
function lowerColorHexes(arr) { return arr.map(lowerColorHex); }

Now let's compose some of them together.

var lighterColor = arrayToString
    .compose(raiseColorHexes)
    .compose(stringToArray);

var darkerColor = arrayToString
    .compose(lowerColorHexes)
    .compose(stringToArray);

console.log(lighterColor('af0189')); // Returns: 'bf129a'
console.log(darkerColor('af0189'));  // Returns: '9e0078'
```

We can even use `compose()` and `curry()` functions together. In fact, they work very well together. Let's forge together the curry example with our compose example. First we'll need our helper functions from before.

```javascript
// component2hex :: Ints -> Int
function componentToHex(c) {
  var hex = c.toString(16);
  return hex.length == 1 ? "0" + hex : hex;
```
First we need to make the curried and partial-applied functions, then we can compose them to our other composed functions.

```javascript
var lighterColors = lighterColor
    .compose(nums2hex.curry());
var darkerRed = darkerColor
    .compose(nums2hex.partialApply(255));
var lighterRgb2hex = lighterColor
    .compose(nums2hex.partialApply());

console.log(lighterColors(123, 0, 22)); // Returns: 8cff11
console.log(darkerRed(123, 0)); // Returns: ee6a00
console.log(lighterRgb2hex(123,200,100)); // Returns: 8cd975
```

There we have it! The functions read really well and make a lot of sense. We were forced to begin with little functions that just did one thing. Then we were able to put together functions with more utility.

Let's look at one last example. Here's a function that lightens an RBG value by a variable amount. Then we can use composition to create new functions from it.

```javascript
// lighterColorNumSteps :: string -> num -> string
function lighterColorNumSteps(color, n) {
  for (var i = 0; i < n; i++) {
    color = lighterColor(color);
  }
  return color;
}

// now we can create functions like this:
var lighterRedNumSteps = lighterColorNumSteps.curry().compose(reds)(0,0);

// and use them like this:
console.log(lighterRedNumSteps(5)); // Return: 'ff5555'
console.log(lighterRedNumSteps(2)); // Return: 'ff2222'
```
In the same way, we could easily create more functions for creating lighter and
darker blues, greens, grays, purples, anything you want. This is a really great way to
construct an API.

We just barely scratched the surface of what function composition can do. What
compose does is take control away from JavaScript. Normally JavaScript will
evaluate left to right, but now the interpreter is saying “OK, something else is going
to take care of this, I'll just move on to the next.” And now the `compose()` function
has control over the evaluation sequence!

This is how Lazy.js, Bacon.js and others have been able to implement things such
as lazy evaluation and infinite sequences. Up next, we'll look into how those libraries
are used.

**Mostly functional programming**

What is a program without side effects? A program that does nothing.

Complementing our code with functional code with unavoidable side-effects can
be called "mostly functional programming." Using multiple paradigms in the same
codebase and applying them where they are most optimal is the best approach.
Mostly functional programming is how even the pure, traditional functional
programs are modelled: keep most of the logic in pure functions and interface with
imperative code.

And this is how we're going to write a little application of our own.

In this example, we have a boss that tells us that we need a web application for our
company that tracks the status of the employees' availability. All the employees at
this fictional company only have one job: using our website. Staff will sign in when
they get to work and sign out when they leave. But that's not enough, it also needs
to automatically update the content as it changes, so our boss doesn't have to keep
refreshing the pages.

We're going to use Lazy.js as our functional library. And we're also going to be lazy:
instead of worrying about handling all the users logging in and out, WebSockets,
databases, and more, we'll just pretend there's a generic application object that does
this for us and just happens to have the perfect API.

So for now, let's just get the ugly parts out of the way, the parts that interface and
create side-effects.

```javascript
function Receptor(name, available){
    this.name = name;
    this.available = available; // mutable state
```
this.render = function(){
  output = '<li>';
  output += this.available ?
    this.name + ' is available'
  :
    this.name + ' is not available';
  output += '</li>';
  return output;
}
}

var me = new Receptor;
var receptors = app.getReceptors().push(me);
app.container.innerHTML = receptors.map(function(r){
  return r.render();
}).join('');

This would be sufficient for just displaying a list of availabilities, but we want it to be reactive, which brings us to our first obstacle.

By using the Lazy.js library to store the objects in a sequence, which won't actually compute anything until the toArray() method is called, we can take advantage of its laziness to provide a sort of functional reactive programming.

    var lazyReceptors = Lazy(receptors).map(function(r){
      return r.render();
    });
    app.container.innerHTML = lazyReceptors.toArray().join('');

Because the Receptor.render() method returns new HTML instead of modifying the current HTML, all we have to do is set the innerHTML parameter to its output.

We'll also have to trust that our generic application for user management will provide callback methods for us to use.

    app.onUserLogin = function(){
      this.available = true;
      app.container.innerHTML = lazyReceptors.toArray().join('');
    };
    app.onUserLogout = function(){
      this.available = false;
      app.container.innerHTML = lazyReceptors.toArray().join('');
    };

This way, any time a user logs in or out, the lazyReceptors parameter will be computed again and the availability list will be printed with the most recent values.
Handling events

But what if the application doesn't provide callbacks for when the user logs in and out? Callbacks are messy and can quickly turn a program into spaghetti code. Instead, we can determine it ourselves by observing the user directly. If the user has the webpage in focus, then he/she must be active and available. We can use JavaScript's `focus` and `blur` events for this.

```javascript
window.addEventListener('focus', function(event) {
    me.available = true;
    app.setReceptor(me.name, me.available); // just go with it
    container.innerHTML = lazyReceptors.toArray().join('');
});
window.addEventListener('blur', function(event) {
    me.available = false;
    app.setReceptor(me.name, me.available);
    container.innerHTML = lazyReceptors.toArray().join('');
});
```

Wait a second, aren't events reactive too? Can they be lazily computed as well? They can in the Lazy.js library, where there's even a handy method for this.

```javascript
var focusedReceptors = Lazy.events(window, "focus").each(function(e){
    me.available = true;
    app.setReceptor(me.name, me.available);
    container.innerHTML = lazyReceptors.toArray().join('');
});
var blurredReceptors = Lazy.events(window, "blur").each(function(e){
    me.available = false;
    app.setReceptor(me.name, me.available);
    container.innerHTML = lazyReceptors.toArray().join('');
});
```

Easy as pie.

By using the Lazy.js library to handle events, we can create an infinite sequence of events. Each time the event is fired, the `Lazy.each()` function is able to iterate one more time.

Our boss likes the application so far, but she points out that if an employee never logs out before leaving for the day without closing the page, then the application says the employee is still available.
To figure out if an employee is active on the website, we can monitor the keyboard and mouse events. Let's say they're considered to be unavailable after 30 minutes of no activity.

```javascript
var timeout = null;
var inputs = Lazy.events(window, "mousemove").each(function(e){
    me.available = true;
    container.innerHTML = lazyReceptors.toArray().join('');
    clearTimeout(timeout);
    timeout = setTimeout(function(){
        me.available = false;
        container.innerHTML = lazyReceptors.toArray().join('');
    }, 1800000); // 30 minutes
});
```

The Lazy.js library has made it very easy for us to handle events as an infinite stream that we can map over. It makes this possible because it uses function composition to take control of the order of execution.

But there's a little problem with all of this. What if there are no user input events that we can latch onto? What if, instead, there is a property value that changes all the time? In the next section, we'll investigate exactly this issue.

**Functional reactive programming**

Let's build another kind of application that works in much the same way; one that uses functional programming to react to changes in state. But, this time, the application won't be able to rely on event listeners.

Imagine for a moment that you work for a news media company and your boss tells you to create a web application that tracks government election results on Election Day. Data is continuously flowing in as local precincts turn in their results, so the results to display on the page are very reactive. But we also need to track the results by each region, so there will be multiple objects to track.

Rather than creating a big object-oriented hierarchy to model the interface, we can describe it declaratively as immutable data. We can transform it with chains of pure and semi-pure functions whose only ultimate side effects are updating whatever bits of state absolutely must be held onto (ideally, not many).

And we'll use the Bacon.js library, which will allow us to quickly develop **Functional Reactive Programming (FRP)** applications. The application will only be used one day out of the year (Election Day), and our boss thinks it should take a proportional amount of time. With functional programming and a library such as Bacon.js, we'll get it done in half the time.
But first, we're going to need some objects to represent the voting regions, such as states, provinces, districts, and so on.

```javascript
function Region(name, percent, parties){
    // mutable properties:
    this.name = name;
    this.percent = percent; // % of precincts reported
    this.parties = parties; // political parties

    // return an HTML representation
    this.render = function(){
        var lis = this.parties.map(function(p){
            return '<li>' + p.name + ': ' + p.votes + '</li>;
        });
        var output = '<h2>' + this.name + '</h2>';
        output += '<ul>' + lis.join('') + '</ul>';
        output += 'Percent reported: ' + this.percent;
        return output;
    }
}

function getRegions(data) {
    return JSON.parse(data).map(function(obj){
        return new Region(obj.name, obj.percent, obj.parties);
    });
}
```

```javascript
var url = 'http://api.server.com/election-data?format=json';
var data = jQuery.ajax(url);
var regions = getRegions(data);
var regions = regions.map(function(r){
    return r.render();
}).join('');
```

While the above would be sufficient for just displaying a static list of election results, we need a way to update the regions dynamically. It's time to cook up some Bacon and FRP.

**Reactivity**

Bacon has a function, `Bacon.fromPoll()`, that lets us create an event stream, where the event is just a function that is called on the given interval. And the `stream.subscribe()` function lets us subscribe a handler function to the stream. Because it's lazy, the stream will not actually do anything without a subscriber.

```javascript
var eventStream = Bacon.fromPoll(10000, function(){
    return Bacon.Next;
});
var subscriber = eventStream.subscribe(function(){
    var url = 'http://api.server.com/election-data?format=json';
```
var data = jQuery.ajax(url);
var newRegions = getRegions(data);
container.innerHTML = newRegions.map(function(r){
    return r.render();
}).join('');
});

By essentially putting it in a loop that runs every 10 seconds, we could get the job done. But this method would hammer-ping the network and is incredibly inefficient. That would not be very functional. Instead, let's dig a little deeper into the Bacon.js library.

In Bacon, there are EventStreams and Properties parameters. Properties can be thought of as "magic" variables that change over time in response to events. They're not really magic because they still rely on a stream of events. The Property changes over time in relation to its EventStream.

The Bacon.js library has another trick up its sleeve. The Bacon.fromPromise() function is a way to emit events into a stream by using promises. And as of jQuery version 1.5.0, jQuery AJAX implements the promises interface. So all we need to do is write an AJAX search function that emits events when the asynchronous call is complete. Every time the promise is resolved, it calls the EventStream's subscribers.

```javascript
var url = 'http://api.server.com/election-data?format=json';
var eventStream = Bacon.fromPromise(jQuery.ajax(url));
var subscriber = eventStream.onValue(function(data){
    newRegions = getRegions(data);
    container.innerHTML = newRegions.map(function(r){
        return r.render();
    }).join('');
});
```

A promise can be thought of as an eventual value; with the Bacon.js library, we can lazily wait on the eventual values.

**Putting it all together**

Now that we have the reactivity covered, we can finally play with some code.

We can modify the subscriber with chains of pure functions to do things such as adding up a total and filtering out unwanted results, and we do it all within onclick() handler functions for buttons that we create.

```javascript
// create the eventStream out side of the functions
var eventStream = Bacon.onPromise(jQuery.ajax(url));
var subscribe = null;
```
var url = 'http://api.server.com/election-data?format=json';

// our un-modified subscriber
$('button#showAll').click(function() {
    var subscriber = eventStream.onValue(function(data) {
        var newRegions = getRegions(data).map(function(r) {
            return new Region(r.name, r.percent, r.parties);
        });
        container.innerHTML = newRegions.map(function(r) {
            return r.render();
        }).join('');
    });
});

// a button for showing the total votes
$('button#showTotal').click(function() {
    var subscriber = eventStream.onValue(function(data) {
        var emptyRegion = new Region('empty', 0, [{
            name: 'Republican', votes: 0
        }, {
            name: 'Democrat', votes: 0
        }]);
        var totalRegions = getRegions(data).reduce(function(r1, r2) {
            newParties = r1.parties.map(function(x, i) {
                return {
                    name: r1.parties[i].name,
                    votes: r1.parties[i].votes + r2.parties[i].votes
                };
            });
            newRegion = new Region('Total', (r1.percent + r2.percent) / 2,
                newParties);
            return newRegion;
        }, emptyRegion);
        container.innerHTML = totalRegions.render();
    });
});

// a button for only displaying regions that are reporting > 50%
$('button#showMostlyReported').click(function() {
    var subscriber = eventStream.onValue(function(data) {
        var newRegions = getRegions(data).map(function(r) {
            if (r.percent > 50) return r;
            else return null;
        }).filter(function(r) {return r != null;});
    });
container.innerHTML = newRegions.map(function(r) {
    return r.render();
}).join('');
});
});

The beauty of this is that, when users click between the buttons, the event stream doesn't change but the subscriber does, which makes it all work smoothly.

**Summary**

JavaScript is a beautiful language.

Its inner beauty really shines with functional programming. It's what empowers its excellent extendibility. Just the fact that it allows first-class functions that can do so many things is what opens the functional flood gates. Concepts build on top of each other, stacking up higher and higher.

In this chapter, we dove head-first into the functional paradigm in JavaScript. We covered function factories, currying, function composition and everything required to make it work. We built an extremely modular application that used these concepts. And then we showed how to use some functional libraries that use these same concepts themselves, namely function composition, to manipulate the order of execution.

Throughout the chapter, we covered several styles of functional programming: data generic programming, mostly-functional programming, and functional reactive programming. They're all not that different from each other, they're just different patterns for applying functional programing in different situations.

In the previous chapter, something called Category Theory was briefly mentioned. In the next chapter, we're going to learn a lot more about what it is and how to use it.
Where to buy this book

You can buy Functional Programming in JavaScript from the Packt Publishing website. Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers. Click here for ordering and shipping details.