Beginning Functional JavaScript

Functional Programming with JavaScript Using EcmaScript 6

Anto Aravinth

Apress®

Beginning Functional JavaScript

Functional Programming with JavaScript Using EcmaScript 6

Anto Aravinth

Apress®

Beginning Functional JavaScript

Anto Aravinth
Chennai, Tamil Nadu, India

ISBN-13 (pbk): 978-1-4842-2655-1 ISBN-13 (electronic): 978-1-4842-2656-8
DOI 10.1007/978-1-4842-2656-8

Library of Congress Control Number: 2017934504

Managing Director: Welmoed Spahr
Editorial Director: Todd Green
Acquisitions Editor: Pramila Balan
Development Editor: Anila Vincent
Technical Reviewers: Anand Kumar and Sakib Shaikh
Coordinating Editor: Prachi Mehta
Copy Editor: Karen Jameson
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global
Cover image designed by Freepik

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-2655-1. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Dedicated to God, Belgin Rayen (late), Susila, Kishore, Ramya and my beloved ones.

Contents at a Glance

Contents

About the Author

Anto Aravinth is a Senior Business Intelligence Developer at VisualBI, Chennai. He has been busy developing web applications using Java; JavaScript; and frameworks like ReactJs, Angular, etc., for the last five years. He has a solid understanding of the Web and its standards. Anto is also an open source contributor to popular frameworks such as ReactJs, Selenium, and Groovy.

Anto loves playing table tennis in his free time. He has a great sense of humor, too! Anto is also a Technical Development Editor for *React Quickly*, a book that will be published by Manning in 2017.

Acknowledgments

Writing a book is not easy as I would have thought. It's almost very like making a movie. You need to go to each publication unit with your table of contents. The table of contents are like your movie script. It needs to have a strong beginning, keep the audience engaged, and then have a good ending. The screenplay should be very good which is achieved via your text. The process of a book starts when an editorial team accepts your table of contents. I want to thank Pramila for that; she was helpful in the initial stages of the book. Of course, writing a technical book needs to be technically correct as well. And that's where I need to give a special thanks to our technical editorial team! They were very good at catching any technical issues in my writing. I want to give special thanks to Anila for working through the chapters and finding any grammatical errors – and making sure the contents are good enough for the readers to be engaged. All these phases are managed by our manager named Prachi. Thanks, Prachi, for making it happen!

I want to dedicate this book to my Lord Jesus, late father Belgin Rayen, and my beloved mother named Susila. I also want to give thanks to Kishore, my brother-in-law, for supporting me throughout my life and career. I never told my sister Ramya (one and only sibling) that I'm authoring a book. I just couldn't predict how she would react to this event. Special thanks to her as well.

Special thanks to all my friends & colleagues who have been very supportive in my career: Deepak, Vishal, Shiva, Mustafa, Anand, Ram (Juspay), Vimal (Juspay), Lalitha, Swetha, Vishwapriya. Final thanks to my close cousins: Bianca, Jennifer, Amara, Arun, Clinton, Shiny, Sanju.

There can be improvements made in my style of writing, content, authoring, etc. If you want to share your thoughts, please contact me at antoaravinthrayen@gmail.com. I'm also available on twitter @antoaravinth.

Thanks for purchasing this book! I hope you will enjoy it. Good luck!

Anto Aravinth, India

CHAPTER 1

Functional Programming in Simple Terms

The first rule of functions is that they should be small. The second rule of functions is that they should be smaller than that.

—ROBERT C. MARTIN

Welcome to the functional programming world. The world, which has only functions, living happily without any outside world dependencies, without states, without mutations – forever. Functional programming is a buzz in recent days. You might have heard about this term within your team, in your local group meeting, and have thought about this. If you're already aware of what that means, this is great! But for those who don't know the term, don't worry. This chapter is for that purpose: to introduce you to *Functional* terms in simple English.

We are going to begin this chapter by asking a simple question: what is a function in Mathematics? Then later on we are going to create a function in JavaScript with a simple example using our function definition. The chapter ends by explaining the benefits that functional programming gives to our developers.

What Is Functional Programming? Why It Matters?

Before we begin to see what the functional programming term means, we have to answer another question: what is a function in mathematics? A function in mathematics can be written like this:

```
f(X) = Y
```

Electronic supplementary material The online version of this chapter (doi:10.1007/978-1-4842-2656-8_1) contains supplementary material, which is available to authorized users.

A. Aravinth, *Beginning Functional JavaScript*, DOI 10.1007/978-1-4842-2656-8_1

The statement can be read like "A *Function F, which takes X as its argument, and returns the output Y.*" X and Y can be any number, for instance. That's a very simple definition. But there are key takeaways in the definitions:

- A function must always take an argument.
- A function must always return a value.
- A function should act only on its receiving arguments (i.e., X) not the outside world.
- For a given X, there will be only one Y.

You might be wondering why we saw the definition of functions in mathematics rather than in JavaScript. Did you? That sounds like a great question to me. The answer is pretty simple: functional programming techniques are heavily based on mathematical functions and its ideas. But hold your breath – we are not going to teach you functional programming in mathematics, but rather use JavaScript to teach them. But throughout the book, we will be seeing the ideas of mathematical functions and how they are used in order to understand functional programming.

Now with that definition in place, we are going to see the examples of *functions* in JavaScript.

Imagine we have to write a function that does the tax calculation. How are you going to do this in JavaScript?

■ **Note** All the examples in the book will be written with ES6. The code snippets in the book are stand-alone so that you can copy and paste them in any one of your favorite browsers that supports ES6. All the examples are run in the Chrome browser version **51.0.2704.84**. The ES6 spec is over here: http://www.ecma-international.org/ecma-262/6.0/

We can implement such a function like this as shown in Listing 1-1:

Listing 1-1. Calculate Tax Function in ES6

```
var percentValue = 5;
var calculateTax = (value) => { return value/100 * (100 + percentValue) }
```

The above function calculateTax does exactly what we want to do. You can call this function with the value, which will return the calculated tax value in the console. It looks neat, doesn't it? Let's pause for a moment and analyze the above function with respect to our mathematical definition. One of the key points of our mathematical function term is that the function logic shouldn't depend upon the outside world. In our above-defined function calculateTax, we have made the function depend on the *global* variable percentValue. Thus the above function we have created can't be called as a real *function* in a mathematical sense. So let's fix that. (Doubt – Can't change font in the template why?)

The fix is very straightforward: we have to just move the percentValue as our function argument:

Listing 1-2. Calculate Tax Function Rewritten

```
var calculateTax = (value, percentValue) => { return value/100 * (100 +
percentValue) }
```

Now our function calculateTax can be called as a real *function*. But what have we gained? We have just made the elimination of global variable access inside our calculateTax function. Removing global variable access inside a function makes it easy for testing. (We will talk about the functional programming benefits in this chapter later on)

Now we have made our relationship with the Math function to our JavaScript function. With this simple exercise, we can define functional programming in simple technical terms. *Functional programming is a paradigm in which we will be creating functions that are going to work out its logic by depending only on its input. This ensures that a function, when called multiple times, is going to return the same result. The function also won't change any data in the outside world, leading to cachable and testable codebase.*

FUNCTIONS VS. METHODS IN JAVASCRIPT

We have talked about the word "function" a lot in this text. Before we move on, I want to make sure you understand the difference between **Functions** and **Methods** in JavaScript.

Simply put, a **Function** is a piece of code that can be called by its name. It can be pass arguments and return values.

However, **Methods** is a piece of code that must be called by its name along with its *associated object* name.

We will look at a quick example of function and method in Listing 1-3:

Function

Listing 1-3. A Simple Function

```
var simple = (a) => {return a} // A simple function
simple(5) //called by its name
```

Method

Listing 1-4. A Simple Method

```
var obj = {simple : (a) => {return a} }
obj.simple(5) //called by its name along with its associated object
```

There are two more important characteristics of Functional programing that are missing in the definition. We are going to discuss them in detail in the upcoming sections before we dive into the benefits of functional programming.

Referential Transparency

With our above definition of function, we have made a statement that all the functions are going to return the same value for the same input. And this property of a function is called a ***Referential transparency***. We will take a simple example as shown in Listing 1-5:

Listing 1-5. Referential Transparency Example

```
var identity = (i) => { return i }
```

In the above code snippet we have defined a simple function called identity. This function is going to return whatever you're passing as its input; that is, if you're passing 5, it's going to return back the value 5 (i.e., the function is just acts as a mirror or identity). Note that our function does operate only on the incoming argument 'i', and there is no global reference inside our function (remember in Listing 1-2, we removed 'percentValue' from global access and made it an incoming argument). This function satisfies the conditions of*Referential Transparency*. Now imagine this function is used between other function calls like this:

```
sum(4,5) + identity(1)
```

With our *Referential Transparency* definition we can convert the above statement into this:

```
sum(4,5) + 1
```

Now this process is called a **Substitution model** as you can directly substitute the result of the function as is (mainly because the function doesn't depend on other global variables for its logic) with its value. This leads to **parallel** code and **caching**. Imagine that with this model, you can easily run the above function with multiple threads without even the need of synchronizing! Why? The reason for synchronizing comes from the fact that threads shouldn't act upon global data when running parallel. Functions that obey Referential Transparency are going to depend only on inputs from its argument; hence threads are free to run without any locking mechanism!

And since the function is going to return the same value for the given input, we can, in fact cache it! For example, imagine there is a function called 'factorial', which calculates the factorial of the given number. 'Factorial' takes the input as its argument for which the factorial needs to be calculated. We all know the 'factorial' of '5' going to be '120'. What if the user calls the 'factorial' of '5' a second time? If the 'factorial' function obeys Referential transparency, we know that the result is going to be '120' as before (and it only depends on the input argument). With this characteristic in mind, we can cache the values of our 'factorial' function. Thus if a 'factorial' is called for the second time with the input as '5', we can return the cached value instead of calculating once again

Here you can see how a simple idea helps in parallel code and cachable code. We will be writing a function in our library for caching the function results, later in the chapter.

REFERENTIAL TRANSPARENCY IS A PHILOSHOPY

Referential transparency is a word that came from **Analytic Philosophy** (https://en.wikipedia.org/wiki/Analytical_philosophy). This branch of philosophy deals with Natural language semantics and its meanings. Here the word "Referential" or "Referent" means to *the thing that the expression refers to.* A context in a sentence is "referentially transparent" if replacing a term in that context by another term that refers to the same entity doesn't alter the meaning.

And that's exactly how we have been defining Referential transparency in our section. We have replaced the value of the function, without affecting the context. Wow, functional programming is a philosophy!

Imperative, Declarative, Abstraction

Functional programming is also about being *declarative* and writing *abstracted* code. We need to understand these two terms before we proceed further. We all know and have worked on N imperative paradigm. We'll take a problem and see how to solve it in an imperative and declarative fashion.

Suppose you have a list or array and want to iterate through the array and print it to the console. The code for might look like this:

Listing 1-6. Iterating over the Array Imperative Approach

```
var array = [1,2,3]
for(i=0;i<array.length;i++)
    console.log(array[i]) //prints 1, 2, 3
```

It works fine. But in the above approach to solve our problem, we are telling *exactly* "how" we need to do it. For example, we have written an implicit for loop with an index calculation of the array length and printing the items. We will stop here. What was the task here? "Print the array elements," right? But it looks like we are telling the compiler what to do. In this case, we are telling "Get Array Length, Loop our array, Get each Element of array using index etc." We call it an "imperative" solution. *Imperative* programming is all about telling the compiler "how" to do the things.

We will now switch to the other side of the coin, *Declarative* programming. In Declarative programming, we are going to tell "what" the compiler needs to do rather than the "how" parts. The "how" parts are being abstracted into common functions (these functions are called as Higher-Order functions, which we will cover in the upcoming chapters). Now we can use the in-built forEach function to iterate the array and print it.

Listing 1-7. Iterating over the Array Declarative Approach

```
var array = [1,2,3]
array.forEach((element) => console.log(element)) //prints 1, 2, 3
```

The above code snippet does print exactly the same output in the previous Listing 1-5. But here we have removed the "how" parts like "Get Array Length, Loop our array, Get each Element of array using index, etc." We have used an *abstracted* function, which takes care of "how" part, leaving us the developers to worry about our problem in hand ("what" part). That's great! We will be creating these in-built functions throughout the textbook!

Functional programming is about creating functions in an abstracted way, which can be reused by other parts of the code. Now we have a solid understanding of what a functional programming is; with this in mind, we can go and explore the benefits of functional programming.

Functional Programming Benefits

We have seen the definition of functional programming and a very simple example of a function in the JavaScript language. But we have to answer a simple question: "What are the benefits of Functional programming?" This section will help you see through the lenses and see the huge benefits that functional programming is opting to offer us! Most of the benefits of Functional programming come from writing Pure Functions. So before we see the benefits of Functional programming, we will see what a pure function is.

Pure Functions

With our definition in place, we can define what is meant by *Pure functions*. Pure functions are the functions that return the same output for the given input. Take an example as given in Listing 1-8:

Listing 1-8. A Simple Pure Function

```
var double = (value) => value * 2;
```

The above function 'double' is a pure function just because given an input, it is going to *always* return the same output. You can try yourself. Calling the double function with input 5 always gives back the result as 10! Pure functions obey Referential transparency. Thus we can replace double(5) with 10, without any hesitations.

So what's the big deal about *Pure Functions*? It has many benefits to give to us. We'll discuss one after the other in the following text.

Pure Functions Lead to Testable Code

Functions that are not pure have side effects. Take our previous tax calculation example (Listing 1-1):

```
var percentValue = 5;
var calculateTax = (value) => { return value/100 * (100 + percentValue) } //
depends on external environment percentValue variable
```

The function calculateTax is not a pure function, mainly because for calculating its logic it depends on the *external environment*. However, the function works but the function is very tough to test! Let's see the reason for this.

Imagine we are planning to run a test for our calculateTax function three times for three different tax calculations. We set up the environment like this:

```
calculateTax(5) === 5.25

calculateTax(6) === 6.3

calculateTax(7) === 7.3500000000000005
```

The entire test passes! But hold on, since our original calculateTax function depends on the external environment variable percentValue, things can go wrong. Imagine the external environment is changing the variable percentValue while you are running the same test cases:

```
calculateTax(5) === 5.25

// percentValue is changed by other function to 2
calculateTax(6) === 6.3  //will the test pass?

// percentValue is changed by other function to 0
calculateTax(7) === 7.3500000000000005 //will the test pass or throw
exception?
```

As you can see here the function is very hard to test. However we can easily fix the issue, by removing the external environment dependency from our function, leading the code to this:

```
var calculateTax = (value, percentValue) => { return value/100 * (100 +
percentValue) }
```

Now you can test the above function without any pain! Before we close this section, we need to mention an important property about Pure function, which is *"Pure Function also shouldn't mutate any external environment variables."*

In other words, the pure function shouldn't *depend* on any external variables (like shown in the example) and also *change* any external variables. We'll not take a quick look what we meant by changing any *external* variables. For example, consider the following code in Listing 1-9:

Listing 1-9. BadFunction Example

```
var global = "globalValue"
var badFunction = (value) => { global = "changed"; return value * 2 }
```

When 'badfunction' function is called it *changes the global variable 'global' to value 'changed.' Is it something to worry about? Yes! Imagine another function that depends on 'global' variable for its business logic! Thus, calling 'badFunction' affects other functions' behavior.* Functions of this nature (i.e., functions that have side effects) make the code base hard to test. Apart from testing, these side effects will make the system behavior very hard to predict in case of debugging!

So we have seen with simple example of how a pure function can help us in easy testing the code. Now we'll look at other benefits we get out of Pure Functions – **Reasonable Code**.

Reasonable Code

As developers we should be good at reasoning about the code or a function. By creating and using Pure functions we can achieve that very simply. To make this point clearer, we are going to use a simple example of function double (from Listing 1-8):

```
var double = (value) => value * 2
```

By looking at this function name we can easily reason that this function doubles the given number and nothing else! In fact, using our Referential transparency concept, we can easily go ahead and replace the double function call with the corresponding result! Developers spend most of their time in reading others' code. Having a function with side effects in your code base is hard to read for other developers in your team. Code base with pure functions are easy to read, understand, and test. Remember that a function (regardless if it's pure function) must always have a meaningful name. With that said, you can't name function 'double' as 'dd' given what it does.

SMALL MIND GAME

We are just replacing the function with value, as if we know the result without seeing its implementation! That's a great improvement in your thinking process about functions. We are substituting the function value as if that's the result it will return!

To give your mind a quick exercise, see this reasoning ability with our in-built Math. max function.

Given the function call:

Math.max(3,4,5,6)

What will be the result?

Did you see the implementation of max to give the result? No, Right? Why? The answer to that question is Math.max is pure function. Now have a cup of coffee; you have done a great job!

Parallel Code

Pure function allows us to run the code in parallel. As pure function is not going to change any of its environments, this means we do not need to worry about the *synchronizing* at all! Of course JavaScript doesn't have real threads to run the functions in parallel, but what if your project uses WebWorkers for running multiple things in parallel? Or a server-side code in node environment that runs the function in parallel?

For example, imagine we have the following code as given in Listing 1-10:

Listing 1-10. Impure Functions

```
let global = "something"
let function1 = (input) => {
        // works on input
        //changes global
        global = "somethingElse"
}
let function2 = () => {
        if(global === "something")
        {
                //business logic
        }
}
```

What if we need to run both 'function1' and 'function2' in parallel? Imagine thread one (T-1) picks 'function1' to run. Thread two (T-2) picks 'function' to run. Now both threads are ready to run and here comes the problem. What if T-1 runs before T-2? Since both functions ('function1' and 'function2') depend on global variable 'global', running these functions in parallel causes undesirable effects. Now change these functions into a pure function as explained in Listing 1-11:

Listing 1-11. Pure Functions

```
let function1 = (input,global) => {
        // works on input
        //changes global
        global = "somethingElse"
}
let function2 = (global) => {
        if(global === "something")
        {
                //business logic
        }
}
```

Here we have moved 'global' variable as arguments for both the functions making them pure. Now we can run both functions on parallel without any issues. Since the functions don't depend on an external environment ('global' variable), we aren't worried about thread execution order as with Listing 1-10.

This section shows us how pure function helps our code to run in parallel without any problems.

Cachable

Since the pure function is going to always return the same output for the given input, we can cache the function outputs. To make this more concrete, we can take a simple example. Imagine we have a function that does do time-consuming calculations. We will name this function longRunningFunction:

```
var longRunningFunction = (ip) => { //do long running tasks and return }
```

If longRunningFunction function is a pure function, then we know that for the given input, it's going to return us the same output! Now with that point in mind, why do we need to call the function again with its input multiple times? Can't we just replace the function call with the function's previous result? (Again note here how we are using the referential transparency concept, thus replacing the function with the previous result value and leaving the context unchanged). Imagine we have a bookkeeping object, which does keep all the function call results of longRunningFunction like this:

```
var longRunningFnBookKeeper = { 2 : 3, 4 : 5 . . .  }
```

The longRunningFnBookKeeper is a simple JavaScript object, which is going to hold all the input (as keys) and outputs (as values) in it as a result of invoking longRunningFunction functions. Now with our pure function definition in place, we can check if the key is present in longRunningFnBookKeeper before invoking our original function, like what is shown in Listing 1-12:

Listing 1-12. Caching Achieved via Pure Functions

```
var longRunningFnBookKeeper = { 2 : 3, 4 : 5 }
//check if the key present in longRunningFnBookKeeper
//if get back the result else update the bookkeeping object
longRunningFnBookKeeper.hasOwnProperty(ip) ?
      longRunningFnBookKeeper[ip] :
      longRunningFnBookKeeper[ip] = longRunningFunction(ip)
```

The above code is relatively straightforward. Before calling our real function, we are checking if the result of that function with the corresponding ip is in the bookkeeping object. If yes, we are returning it, or else we are calling our original function and updating the result in our bookkeeping object as well. Did you see how easily we have made the function calls cachable by using less code? That's the power of pure functions!

We will be writing a functional lib, which does the caching, or technical *memorization,* of our pure function calls later in the book!

Pipelines and Composable

With pure functions we are going to do only one thing in that function. We have seen already how the pure function is going to act as a self-understanding of what that function does by seeing its name. Pure functions should be designed such a way that it should do only one thing. Doing only one thing and doing it perfectly is a UNIX philosophy; we will be following the same while implementing our pure functions. There are many commands in UNIX and LINUX platforms, which we are using for day-to-day tasks. For example, we use cat to print the contents of the file, grep to search the files, wc to count the lines, etc. These commands do solve one problem at a time. But we can *compose* or *pipeline* to do the complex tasks. Imagine we want to find a specific name in a text file and count its occurrences. How we will be doing that in our command prompt? The command looks like this:

```
cat jsBook | grep –i “composing” | wc
```

The above command does solve our problem via composing many functions. Composing is not only unique to UNIX/LINUX command lines, but they are the heart of the Functional programming paradigm. We call them *Functional Composition* in our world. Imagine these same command lines have been implemented in JavaScript functions. We can use them with the same principles to solve our problem!

Now think about another problem in a different way to solve. You want to count the number of lines in text. How will you solve it? Ahaa! You got the answer. Isn’t?

The commands are in fact a pure function with respect to our definition. It takes an argument and returns the output to the caller without affecting any of the external environments!

■ **Note** You might be thinking, does JavaScript support the operator “|” for composing functions? The answer is no; however we can create one. We will be creating the corresponding

---

That’s a lot of benefits we are getting by following a simple definition. Before I close this chapter, I want to show the relationship between *Pure function* and an *Mathematical Function*. We will tackle that next!

Pure Function Is a Mathematical Function

In the section “Cachable” we saw a code snippet (Listing 1-12):

```
var longRunningFunction = (ip) => { //do long running tasks and return }
var longRunningFnBookKeeper = { 2 : 3, 4 : 5 }
//check if the key present in longRunningFnBookKeeper
//if get back the result else update the bookkeeping object
longRunningFnBookKeeper.hasOwnProperty(ip) ?
      longRunningFnBookKeeper[ip] :
      longRunningFnBookKeeper[ip] = longRunningFunction(ip)
```

The main aim was to cache the function calls. We did so by the bookkeeping object. Imagine we have called the longRunningFunction many times so that our longRunningFnBookKeeper grows into the object, which looks like this:

```
longRunningFnBookKeeper = {
   1 : 32,
   2 : 4,
   3 : 5,
   5 : 6,
   8 : 9,
   9 : 10,
   10 : 23,
   11 : 44
}
```

Now imagine that longRunningFunction input ranges only from 1-11 integers (just for an example). And since we have already built the bookkeeping object for this particular range, we can refer only the longRunningFnBookKeeper to say the output longRunningFunction for the given input.

Let's analyze this bookkeeping object. This object gives us the clear picture that our function longRunningFunction does takes a input and *maps* over the output for the given range (in this case it's 1-11). And the important point to note over here is that the inputs (in this case, the keys) have, mandatorily, a corresponding output (in this case, the result) in the object. And also there is no input in the key section that maps to two outputs!

With this analysis we can revisit the Mathematical Function definition (this time a more concrete definition from Wikipedia). (https://en.wikipedia.org/wiki/Function_(mathematics):

In mathematics, a function is a relation between a set of inputs and a set of permissible outputs with the property that each input is related to exactly one output. The input to a function is called the argument and the output is called the value. The set of all permitted inputs to a given function is called the *domain* of the function, while the set of permissible outputs is called the *codomain*.

The above definition is exactly the same as our Pure functions! Have a look at our longRunningFnBookKeeper object. Can you find the *domain* and *codomain* of our function? Yeah, you can! With this very simple example you can easily see how the Mathematical Function idea is borrowed into Functional paradigm world (as I stated in the beginning of the chapter).

What We Are Going to Build

We have talked a lot about functions and functional programming in this chapter. With this fundamental knowledge we are going to build the functional library called ES6-Functional. This library will be built chapter by chapter throughout the text. By building the functional library you will be exploring how JavaScript functions can be used (in a functional way) and also at the same time how functional programming can be applied in day-to-day activities (using our created function to solve the problem in our code base)!

Is JavaScript a Functional Programming Language?

Before we close this chapter, we have to take a step back and answer a fundamental question. Is JavaScript a functional programming language? The answer is yes and no. We said in the beginning of the chapter that functional programming is all about functions, which have to take at least an argument and return a value. But to be frank we can create a function in JavaScript that can take no argument and in fact return nothing. For example, the below code is a valid code in the JavaScript engine:

```
var useless = () => {}
```

The above code will execute without any error in the JavaScript world! The reason is that being JavaScript is not a pure functional language (like Haskell) but rather a Multi-paradigm language. However the language is very much suitable for the functional programming paradigm as discussed in this chapter. The techniques and the benefits that we have discussed up to now can be applied in pure JavaScript! And that's the reason for this book's title!

JavaScript is a language that has a support for Function as arguments, passing functions to another functions, etc – mainly because JavaScript treats *functions as its first-class citizens* (we will talk more about this in upcoming chapters). Because of the constraints according to the definition of the term function, we as developers need to take them into account while creating them in the JavaScript world. By doing so, we will gain many advantages from the functional paradigm as discussed in this chapter.

Summary

In this chapter we have seen what Functions are in Math and in the programming world. We started with a simple definition of function in mathematics and went over seeing small and solid examples of functions and functional programming paradigm in JavaScript. We also defined what Pure function is and discussed, in detail, their benefits. At the end of the chapter we also showed the relationship between Pure functions and Mathematical functions. We also discussed how JavaScript could be treated as a functional programming language. A lot of progress has been made in this chapter.

In the next chapter, we will be reading about creating and executing functions in the ES6 context. Now with ES6 we have several ways to create functions; that's exactly what we will be reading about in the next chapter!

CHAPTER 2

Fundamentals of JavaScript Functions

Note The chapter examples and library source code are in branch `chap02`. The repo's URL is: `https://github.com/antoaravinth/functional-es6.git`

Once checkout the code, please checkout branch `chap02`:

```
...
git checkout -b chap02 origin/chap02
...
```

For running the codes, as before run:

```
...
npm run playground
...
```

In the previous chapter we saw what functional programming is all about. We saw how functions in the software world are nothing but Mathematical functions. We spent a lot of time discussing how Pure functions can bring us huge advantages such as parallel code execution, being cachable, etc. We are now convinced that functional programming is all about functions.

In this chapter we are going to see how functions in JavaScript can be used. We will be looking at the latest JavaScript version *ES6*. This chapter will be a refresher on how to create functions, call them, and pass arguments in ES6 but not explaining all the features of ES6. But that's not the goal of this book. I strongly recommend you to try all the code snippets in the book to get a gist of how to use ES6 functions (more precisely we will be working on *arrow functions*).

A. Aravinth, *Beginning Functional JavaScript*, DOI 10.1007/978-1-4842-2656-8_2

Once we have a solid understanding of how to use functions, we will be turning our focus onto seeing how to run the ES6 code in our system. As of today, browsers don't support all features of ES6. In order to tackle that, we will be using a tool called *Babel*. At the end of the chapter we will be starting our groundwork on creating a functional library. For this purpose we will be using a node project that will be set up using Babel-Node tool to run our ES6 codes in your system!

ECMAScript A Bit of History

ECMASCRIPT is a specification of JavaScript, which is maintained by Ecma International in ECMA-262 and ISO/IEC 16262. There are three versions of ECMASCRIPT; to be more specific they are the following:

1. **ECMAScript 1 –** was the very first version of JavaScript language, which was released in the year 1997.
2. **ECMAScript 2 –** is the second version of the JavaScript language, which contains very minor changes with respect to the previous versions. This version got released in the year 1998.
3. **ECMAScript 3 –** this version introduced several features, which got released in the year 1999.
4. **ECMAScript 5 –** this version is supported by almost all of the browsers today. This is the version that had introduced *strict* mode into the language. It was released in the year 2009. **ECMAScript 5.1** also released with minor corrections in June 2011.
5. **ECMAScript 6 –** this version is where JavaScript has seen many changes like introducing classes, Symbols, Arrow Functions, and Generators, etc. It is not yet supported by many browsers today.

We will be referring to ECMAScript as ES6 in this book. So these terms are interchangeable.

Creating and Executing Functions

In this section we are going to see how to create and execute functions in several ways in ES6. This section is going to be long and interesting!

Since many browsers do not yet support ES6 today, we want to find a way to run ES6 code smoothly. Meet *Babel*. Babel is a *transpiler*, which can convert ES6 code into valid ES5 code (note that in our history section, we mentioned ES5 code can be run in all browsers today). By converting the code into ES5 the developers have a way of seeing and using the features of ES6 without any problem. Using Babel, we can run all the code samples that are presented in this book. Installation of Babel is covered in Appendix A. Kindly refer to this appendix and install Babel before we begin.

Now having installed Babel, let's get our hands dirty by seeing our first simple function in ES6.

First Function

We will define our first simple function in ES6. The simplest function one can write in ES6 is as follows (Listing 2-1):

Listing 2-1. A Simple Function

```
() => "Simple Function"
```

If you try to run this function in babel-repl, you can see the result as:

```
[Function]
```

■ **Note** It's not necessary that you run the code samples in the Babel world. If you're using the latest browser and you're sure that it supports ES6 (you can check it here: https://kangax.github.io/compat-table/es6/), then you can use your browser console to run the code snippets. After all it's a matter of choice. And if you're running the code, say in Chrome, for example, the above code snippet should give you this result:

```
function () => "Simple Function"
```

The point to note over here is the results might differ in showing the function *representation* based on where you're running the code snippets.

Yeah, that's it – we have function! Take a moment to analyze the above function. Let's split them:

```
() => "Simple Function"

//where () represents function arguments
//=> starts the function body/definition
//content after => are the function body/definition.
```

In ES6 we can skip the function keyword to define functions. You can see we have used => operator to define the function body. Functions created this way in ES6 are called *Arrow Functions*. We will be using Arrow functions throughout the book.

Now that the function is defined, we can execute it to see the result. Oh wait! The function we have created doesn't have a name. Then how do I call it?

■ **Note** Functions that don't have names are called Anonymous functions. We will understand the usage of anonymous functions in the functional programming paradigm, when seeing Higher Order functions in the next chapter.

Let's assign a name for it as shown in Listing 2-2:

Listing 2-2. A Simple Function with Name

```
var simpleFn = () => "Simple Function"
```

Since we now have access to the function `simpleFn` we can use this reference to execute the function:

```
simpleFn()
//returns "Simple Function" in the console
```

That's great! We have created a function and also executed it in ES6.

We can see how the same function looks alike in ES5. We can use babel to convert our code into ES5, using the following command:

```
babel simpleFn.js --presets babel-preset-es2015 --out-file script-compiled.js
```

This will generate the file called the `script-compiled.js` in your current directory. Now open up the generated file in your favorite editor:

```
"use strict";

var simpleFn = function simpleFn() {
  return "Simple Function";
};
```

That's our equivalent code in ES5! You can sense how it's much easier and concise to write functions in ES6! Don't you? There are two important points to note in the converted code snippets. We will discuss them one after the other.

Strict Mode

In this section we will discuss "Strict Mode" in JavaScript. We'll see its benefits and why one should prefer "Strict Mode."

You can see that the converted code runs in the `strict` mode, as shown here:

```
"use strict";

var simpleFn = function simpleFn() {
  return "Simple Function";
};
```

Strict modes have nothing to do with ES6, but discussing them here is the right choice. As we have already discussed in the ECMAScript History section, strict mode was introduced to the JavaScript language at ES5.

Simply put, strict mode is a *restricted variant of JavaScript*. The same JavaScript code that is running in *strict* mode can be *semantically* different from the code, which is not using strict. All the code snippets, which don't add the use strict in their js files, are going to be in non-strict mode.

Why should we use strict mode? What are the advantages? There are many advantages of using strict mode style in the world of JavaScript. One simple thing is if you are defining a variable in global state (i.e., without specifying var command) like this:

```
"use strict";

globalVar = "evil"
```

In strict mode it's going to be an error! That's a good catch for our developers, because global variables are very evil in JavaScript! However if the same code were run in non-strict mode, then it wouldn't have complained about the error!

Now as you can guess, the same code in JavaScript can produce different results whether you're running in strict or non-strict mode. Since strict mode is going to be very helpful for us, we will leave Babel to use strict mode while transpiling our ES6 codes.

■ **Note** We can place use stricts in either the beginning of your JavaScript file, in which case it's going to apply its check for the full functions defined in the particular file. Or else you can use strict mode only to specific functions. In that case, strict mode will be applied only to that particular function, leaving other function behaviors in non-strict mode. More on MDN (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode)

Return Statement Is Optional

In ES5 converted code snippet, we saw that Babel adds the return statement in our `simpleFn`.

```
"use strict";

var simpleFn = function simpleFn() {
  return "Simple Function";
};
```

Where as in our real ES6 code, we didn't specify any return statement:

```
var simpleFn = () => "Simple Function"
```

Thus in ES6, if you have a function with only a single statement then it implicitly means that it returns the value. What about multiple statement functions? How we are going to create them in ES6?

Multiple Statement Functions

Now we are going to see how to write multiple statement functions in ES6. Let's make our `simpleFn` a bit more complicated as follows in Listing 2-3:

Listing 2-3. Multistatement Function

```
var simpleFn = () => {
   let value = "Simple Function"
   return value;
} //for multiple statement wrap with { }
```

Run the above function, and you will get the same result as before. But here we have used the multiple arguments to achieve the same behavior. Apart from that, you could notice that we have used `let` a keyword define our `value` variable. The `let` keyword is new to the JavaScript keyword family. The `let` keyword allows you to declare variables that are *limited to a particular scope of block!* This is unlike the `var` keyword that defines the variable globally to a function regardless of the block in which it's defined.

To make the point concrete, we can write the same function with `var` and the `let` keyword, inside an if block as shown in Listing 2-4.

Listing 2-4. SimpleFn with var and let Keywords

```
var simpleFn = () => { //function scope
   if(true) {
      let a = 1;
      var b = 2;
      console.log(a)
      console.log(b)
   } //if block scope
   console.log(b) //function scope
   console.log(a) //function scope
}
```

Running this function gives the following output:

```
1
2
2
Uncaught ReferenceError: a is not defined(...)
```

As you can see from the output, the variable declared via `let` keyword is accessible only within the if block not outside the block. As you notice, JavaScript throws the error when we access the variable a variable outside the block! But whereas the variable declared with `var` when doesn't act that way. Rather, it declares the variable scope for the whole function. That's the reason variable `b` can be accessed outside the if block.

Since block scope is very much needed going further, we will be using the `let` keyword for defining variables throughout the book. Now let's see how to create a function with arguments as the final section.

Function Arguments

Creating functions with arguments is the same as in ES5. Look at a quick example as follows (Listing 2-5):

Listing 2-5. Function with Argument

```
var identity = (value) => value
```

Here we create a function called `identity`, which takes `value` as its argument and returns the same. As you can see, creating functions with arguments are the same as ES5; only the syntax of creating the function is changed.

ES5 Functions Are Valid in ES6

Before we close this section, we need to make an important point clear. The functions that were written in ES5 are still valid in ES6! It's just a small matter that ES6 has introduced Arrow functions, it but doesn't replace the old function syntax or anything else. However we will be using ES6 functions throughout this book to showcase the functional programming approach.

Setting Up Our Project

After having an understanding of how to create Arrow functions in ES6, we will shift our focus onto project setup in this section. We are going to set up our project as a node project and at the end of the section, we will be writing our first Functional Function. Let's begin!

■ **Note** Make sure you have installed node and npm by following Appendix A.

Initial Setup

In this section, we will be following a simple step-by-step guide to set up our environment. The steps are as follows:

1. The first step is to create a directory where our source code is going to be. Create a directory and name it whatever you want.
2. Go into that particular directory and run the following command from your terminal:

   ```
   npm init
   ```

3. After running step 2, it will be asking you a set of questions; you can provide the value you want. Once it's done, it will create a file called `pacakage.json` in your current directory.

The project `package.json` that I have created looks like this as shown here in Listing 2-6:

Listing 2-6. Package.json Contents

```
{
  "name": "learning-functional",
  "version": "1.0.0",
  "description": "Functional lib and examples in ES6",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Anto Aravinth @antoaravinth",
  "license": "ISC"
}
```

Now we need to add a few libraries, which will allow us to write ES6 code and execute them. Run the following command in the current directory:

```
npm install --save-dev babel-preset-es2015-node5
```

■ **Note** The book uses Babel version "babel-preset-es2015-node5." It's highly possible that this specific version may be outdated by the time you read this text. You are free to install the latest version and everything should work smoothly. However, in the context of the book, we will be using the specified version.

The above command downloads the babel package called ES2015-Node5; the main aim of this package is to allow ES6 code to run on Node Js platform. The reason is that Node Js, at the time of writing this book; is not fully compatible with ES6 features.

Once the above command is run, you will be able to see a folder called `node_modules` created in the directory, which has the `babel-preset-es2015-node5` folder.

Since we have used `--save-dev` while installing, npm does add the corresponding babel dependencies to our `package.json`. Now if you open your `package.json`, it looks like this:

Listing 2-7. After Adding the devDependencies

```
{
  "name": "learning-functional",
  "version": "1.0.0",
  "description": "Functional lib and examples in ES6",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Anto Aravinth @antoaravinth>",
  "license": "ISC",
  "devDependencies": {
    "babel-preset-es2015-node5": "^1.2.0",
    "babel-cli": "^6.23.0"
  }
}
```

Now that this is in place, we can go ahead and create two directories called `lib` and `functional-playground`. So now your directory looks like the following:

```
learning-functional
  - functional-playground
  - lib
  - node_modules
    - babel-preset-es2015-node5/*
  - package.json
```

Now we are going to put all our functional library code into `lib` and use `functional-playground` to play and understand our functional techniques.

Our First Functional Approach to the Loop Problem

Imagine we have to iterate through the array and print the data to the console. How do we achieve this in JavaScript?

Listing 2-8. Looping an Array

```
var array = [1,2,3]
for(i=0;i<array.length;i++)
    console.log(array[i])
```

As we have already discussed in Functional Programming in Simple Terms, Chapter 1, abstracting the operations into functions is one of the pillars of functional programming. So let's go and abstract this operation into function, so that we can reuse it any time we need to rather than repeating ourselves in telling "how" to iterate the loop.

Create a file called `es6-functional.js` in `lib` directory. Our directory structure looks like this:

```
learning-functional
  - functional-playground
  - lib
    - es6-functional.js
  - node_modules
    - babel-preset-es2015-node5/*
  - package.json
```

Now with that file in place, go ahead and place the below content into that file:

Listing 2-9. forEach Function

```
const forEach = (array,fn) => {
   let i;
   for(i=0;i<array.length;i++)
      fn(array[i])
}
```

■ **Note** For now don't worry about how this function works. We are going to see how Higher-Order functions work in JavaScript in the next chapter and provide loads of examples.

You might notice that we have started with a keyword `const` for our function definition. This keyword is part of ES6, which makes the declaration constant. For example, if someone tries to reassign the variable with the same name like this:

```
forEach = "" //making your function as string!
```

The above code will throw an error like this:

```
TypeError: Assignment to constant variable.
```

This will prevent it from being accidently reassigned! Now we'll go and use the above created function to print all the data of the array to the console. In order to do that, create a file called `play.js` function in `functional-playground` directory. So now the current file looks like:

```
learning-functional
  - functional-playground
     - play.js
  - lib
```

```
    - es6-functional.js
  - node_modules
    - babel-preset-es2015-node5/*
  - package.json
```

We will call the `forEach` in our play.js file. But how are we are going to call this function, which resides in a different file?

Gist on Exports

ES6 also introduced the concept called *modules*. ES6 modules are stored in files. In our case we can think of `es6-functional.js` file itself as a module. Along with the concept of modules came imports and exports statements. In our running example, we have to *export* the `forEach` function so that others can use them. So that we can change the following code into

Listing 2-10. Exporting forEach Functionconst forEach = (array,fn) => {

```
   let i;
   for(i=0;i<array.length;i++)
      fn(array[i])
}
export default forEach
```

in our es6-functional.js file.

Gist on Imports

Now that we have exported our function as you can see in Listing 2-10, let's go and consume it via import! Open the file `play.js` and add the following into it as shown in Listing 2-11:

Listing 2-11. Importing forEach Function

```
import forEach from '../lib/es6-functional.js'
```

The above line tells JavaScript to import the function called `forEach` from `es6-functional.js`. Now the function is available to the whole file with the name `forEach`. Now add the code into `play.js` like this as shown here in Listing 2-12:

Listing 2-12. Using the Imported forEach function

```
import forEach from '../lib/es6-functional.js'
var array = [1,2,3]
forEach(array,(data) => console.log(data)) //refereing to imported forEach
```

Running the Code Using Babel-Node

Let's run our `play.js` file. Since we are using ES6 in our file, we have to use Babel-Node to run our code. Babel-Node is used to transpile our ES6 code and run it on Node js. Babel-Node should be installed along with babel-cli.

■ **Note** Babel-node will be available in the terminal, only if you have installed babel-cli globally. Kindly refer to Appendix A for installing cli globally.

So from our project root directory, we can call the babel-node like this:

```
babel-node functional-playground/play.js --presets es2015-node5
```

The above command tells us that our `play.js` file should be transpiled with `es2015-node5` and run into node js. This should give the output as follows:

```
1
2
3
```

Hurray! Now we have abstracted out for logic into a function. Imagine you want to iterate and print the array contents with multiples of 2. How will we do it? Super simple – reuse our `forEach`:

```
forEach(array,(data) => console.log(2 * data))
```

which will print the output as expected!

■ **Note** We will be using this pattern throughout the book. We will be discussing the problem with an imperative approach. Then will go ahead and implement our functional techniques and capture them in a function into es6-functional.js. And then use that to play around in play.js file!

Creating Script in Npm

We have seen how to run our play.js file. But it's lot to type! Each time we need to run the following:

```
babel-node functional-playground/play.js --presets es2015-node5
```

Rather than this, we can bind the following command to our npm script. We will change the `package.json accordingly:`

Listing 2-13. Adding npm Scripts to package.json

```
{
  "name": "learning-functional",
  "version": "1.0.0",
  "description": "Functional lib and examples in ES6",
  "main": "index.js",
  "scripts": {
    "playground" : "babel-node functional-playground/play.js --presets
    es2015-node5"
  },
  "author": "Anto Aravinth @antoaravinth",
  "license": "ISC",
  "devDependencies": {
    "babel-preset-es2015-node5": "^1.2.0"
  }
}
```

Now we have added the babel-node command to scripts. So we can run our playground file (node functional-playground/play.js) as follows:

```
npm run playground
```

which will run the same as before.

Running the Source Code from Git

Whatever we are discussing in the chapter will go into a git repository (https://github.com/antoaravinth/functional-es6). You can clone them into your system using git like this:

```
git clone https://github.com/antoaravinth/functional-es6.git
```

Once you clone the repo, you can move into a specific chapter source code branch. Each chapter has its own branch in the repo. For example, in order to see the code samples used in Chapter 2, then you need to do this:

```
git checkout -b chap02 origin/chap02
```

Once you check out the branch, you can run the playground file as before!

Summary

In this chapter, we have spent a lot of time in seeing how to use functions in ES6 modes. We saw how Arrow functions are being introduced and used in ES6. We have taken the advantage of Babel tools for running our ES6 code seamlessly in our Node platform. We also created our project as a Node project. In our node project, we saw how to use Babel Node to convert the ES6 code and run them in node environment using presets. We also saw how to download the book source code and run it. With all these techniques under our belt, in the next chapter we will be focusing on what Higher-Order Functions mean!

CHAPTER 3

Higher-Order Functions

■ **Note** The chapter examples and library source code are in branch `chap03`. The repo's URL is: https://github.com/antoaravinth/functional-es6.git

Once checkout the code, please checkout branch `chap03`:

```
...
git checkout -b chap03 origin/chap03
...
```

For running the codes, as before run:

```
...
npm run playground
...
```

In the previous chapter we saw how to create simple functions in ES6. We also set up our environment to play around with functional programs using node ecosystem. In fact, we have created our first functional program API called `forEach` in the previous chapter. There is something special about the `forEach` function that we have developed in Chapter 2. We passed a function itself as an argument to our `forEach` function. There is no trick involved there; it's part of JavaScript specification that a function can be passed as an argument. JavaScript as a language treats functions as data. This is a very powerful concept that allows us to pass functions in place of data. A function that takes another function as its argument is called a Higher-Order function.

We are going to see Higher-Order functions (HOC for short) in this chapter in depth. We will be starting the chapter with a simple example and definition of HOC. Later we will be moving to see more real-world examples of how HOC can help a programmer to solve complex problems easily. As before, we will be adding the HOC functions that we are creating in the chapter in our library. Let's get started!

A. Aravinth, *Beginning Functional JavaScript*, DOI 10.1007/978-1-4842-2656-8_3

■ **Note** We will be creating few higher-order functions and adding it to our library. We are doing this for understanding how things work behind the scenes. The library is good for learning the current resources but they are not production ready for the library. So keep that in mind. :)

Understanding Data

As programmers we know our programs act on *data*. Data is something that is very important for the consumption of our written program to execute. Hence almost all programming languages give several data for the programmer to work with. For example, we can store the name of a person in `String` data type. JavaScript offers several data types that we will be seeing in the next subsection. At the end of the section, we will be introduced to a solid definition of higher-order functions, with simple and concise examples.

Understanding JavaScript Data Types

Every programming language has data types. These data types can hold data and allow our program to act upon it. In this little section, we will be seeing JavaScripts' data types.

In a nutshell, JavaScript as a language supports the following data types:

- .Numbers
- .Strings
- .Booleans
- .Objects
- .null
- .undefined

and importantly, we also have our friend `functions` as a data type in JavaScript language. Since `functions` are data types like `String`, we can pass them around, store them in a variable, etc., very similarly as we do for `String` and `Numbers` data types. Functions are *First Class Citizens* when the language permits them to be used as any other data type, that is, functions can be assigned to variables, passed around as arguments, and can be returned from other functions. In the next section we will see a quick example of what we mean by storing and passing functions around.

Storing a Function

As mentioned in the previous section, `functions` are nothing but data. Since it's data, we can hold them in a variable! The below code (Listing 3-1) is literally a valid code in JavaScript context:

Listing 3-1. Storing a Function to Variable

```
let fn = () => {}
```

In the above code snippet, `fn` is nothing but a variable that is pointing to a data type function. We can quickly check that `fn` is of type `function` by running the following:

```
typeof fn
=> "function"
```

Since `fn` is just a reference to our function, we can call it like this:

```
fn()
```

the above will execute the function that `fn` points to.

Passing a Function

As day-to-day JavaScript programmers, we know how to pass data to a function. Consider the following function (Listing 3-2), which takes an argument and consoles the type of the argument:

Listing 3-2. tellType Function

```
var tellType = (arg) => {
        console.log(typeof arg)
}
```

One can pass the argument to `tellType` function to see it in action:

```
let data = 1
tellType(data)
=> number
```

Nothing fancy here. As seen in the previous section, we can store even `functions` in our variable (as functions in JavaScript are data). So how about passing a variable that has reference to a function? Let's quickly check it:

```
var dataFn = () => {
        console.log("I'm a function")
}
tellType(dataFn)
=> function
```

That's great! Now we will make our `tellType` to execute the passed argument as shown in Listing 3-3 if it's of type `function`:

Listing 3-3. tellType Executes arg if It's Function

```
var tellType = (arg) => {
   if(typeof arg === "function")
      arg()
   else
          console.log("The passed data is " + arg)
}
```

Here we are checking whether the passed `arg` is of type `function`; if so, call it. Remember if a variable is of type `function`, it means it has a reference to a function that can be executed. That is the reason we are calling arg() if it enters an `if` statement in the above code snippet.

Let's execute our `tellType` function by passing our `dataFn` variable to it:

```
tellType(dataFn)
=> I'm a function
```

We have successfully passed a function `dataFn` to another function `tellType`, which has executed the passed function. That's so simple.

Returning a Function

We have seen how to pass a function to another function. Since functions are simple data in JavaScript, we can return them from other functions, too (like other data types).

We'll take a simple example of a function that returns another function as shown below in Listing 3-4:

Listing 3-4. Crazy Function Return String

```
let crazy = () => { return String }
```

■ **Note** JavaScript has an in-built function called `String`. We can use this function to create new string values in JavaScript like this:

```
String("HOC")
=> HOC
```

Note that our `crazy` function returns a function reference that is pointing to `String` function. Let's go and call our `crazy` function:

```
crazy()
=> String() { [native code] }
```

As you can see, calling the `crazy` function returns a `String` function. Note that it just returns the function reference not *executing* the function. So we can hold back the returned function reference and call them like this:

```
let fn = crazy()
fn("HOC")
=> HOC
```

or even better like this:

```
crazy()("HOC")
=> HOC
```

■ **Note** We will be using simple documentation on top of all functions, which are going to return another function. It will be really helpful going forward as it makes reading the source code easy. For example, the `crazy` function will be documented like this:

```
//Fn => String

let crazy = () => { return String }
```

`Fn => String` comment helps the reader understand that crazy function, which executes and returns another function that points to `String`.

We will be using these sorts of readable comments in our book.

In these sections we have seen functions, which take other functions as its argument and have also seen examples on functions that do not return another function. Now it's time to bring you to the higher-order function definition:

> A Higher-Order Function is a function that receives the function as its argument and/or returns them as outputs.

Abstraction and Higher-Order Functions

Now we have seen how to create and execute higher-order functions. Generally speaking, higher-order functions are written usually to *abstract* the common problems. In other words, higher-order functions are nothing but defining *Abstractions*.

In this section we are going to discuss the relationship that higher-order functions has with the term *abstraction*.

Abstraction Definitions

Wikipedia helps us in getting the definitions of Abstraction:

> In software engineering and computer science, abstraction is a technique for managing complexity of computer systems. It works by establishing a level of complexity on which a person interacts with the system, suppressing the more complex details below the current level. The programmer works with an idealized interface (usually well defined) and can add additional levels of functionality that would otherwise be too complex to handle.

and it also includes the following text (which is what we are interested in):

> For example, a programmer writing code that involves numerical operations may not be interested in the way numbers are represented in the underlying hardware (e.g. whether they're 16 bit or 32 bit integers), and where those details have been suppressed it can be said that they were *abstracted away*, leaving simply numbers with which the programmer can work.

The above text clearly gives the idea on abstraction. Abstraction allows us to work on the desired goal but not worrying about the underlying system concepts.

Abstraction via Higher-Order Functions

In this section we will see how higher-order functions help us to achieve the abstraction concept we discussed in the previous section. Here is the code snippet of our forEach function defined in the previous chapter (Listing 2-9):

```
const forEach = (array,fn) => {
        for(let i=0;array.length;i++)
                fn(array[i])
}
```

The above function `forEach` here has *abstracted* away the problem of traversing the array. The user of the API `forEach` doesn't need to understand how `forEach` have implemented the traversing part, thus abstracting away the problem.

■ **Note** In the `forEach` function, the passed function `fn` is called with a single argument as the current iteration content of the array as you can see here:

```
. . .
     fn(array[i])
. . .
```

So when the user of the `forEach` function calls it like this:

```
forEach([1,2,3],(data) => {
       //data is passed from forEach function
       //to this current function as argument
})
```

---

forEach essentially traverses the array. What about traversing a JavaScript object? Traversing a JavaScript object has steps like this:

1. Iterate all the keys of the given object.
2. Identify that the key belongs to its own object.
3. Get the value of the key if step 2 is true.

Let's *abstract* these steps into a higher-order function named forEachObject:

Listing 3-5. forEachObject Function Definition

```
const forEachObject = (obj,fn) => {
    for (var property in obj) {
            if (obj.hasOwnProperty(property)) {
                //calls the fn with key and value as its argument
                fn(property, obj[property])
            }
    }
}
```

■ **Note** forEachObject takes the first argument as a JavaScript object (as obj) and the second argument is a function fn. It traverses the object using the above algorithm and calls the fn with key and value as its argument, respectively.

Here they are in action:

```
let object = {a:1,b:2}
forEachObject(object, (k,v) => console.log(k + ":" + v))
=> a:1
=> b:1
```

Cool! An important point to note is that both forEach and forEachObject functions are higher-order functions, which allow the developer to work on task (by passing the corresponding function) abstracting away the traversing part! And also since these traversing functions are being abstracted away, we can test them thoroughly, leading to a concise code base. We will be more functional about higher-order functions. Let's go and implement an abstracted way for handling control flows.

For that, let us create a function called unless. Unless is a simple function, which takes a predicate (that should be either true or false); and if the predicate is false; call the fn as shown below here in Listing 3-6:

Listing 3-6. unless Function Definition

```
const unless = (predicate,fn) => {
        if(!predicate)
                fn()
}
```

With the unless function in place, we can go and write a concise piece of code to find the list of even numbers. The code for it looks like this:

```
forEach([1,2,3,4,5,6,7],(number) => {
        unless((number % 2), () => {
                console.log(number, " is even")
        })
})
```

The above code when executed is going to print the following:

```
2 ' is even'
4 ' is even'
6 ' is even'
```

In the above case we are getting the even numbers from the array list. What if we want to get the list of even numbers from, say, 0 to 100? We can't use forEach here (of course we can, if we have the array that has [0,1,2.....,100] content). Let's meet another higher-order function called *times*. Times is yet another simple higher-order function, which takes the number and calls the passed function as many times as the caller mentioned. The times function looks like what is shown here in Listing 3-7:

Listing 3-7. times Function Definition

```
const times = (times, fn) => {
  for (var i = 0; i < times; i++)
        fn(i);
}
```

Times function looks very similar to the forEach function; it's just that we are operating on a Number rather than an Array. Now with the times function in place, we can go ahead and solve our problem in hand like this:

```
times(100, function(n) {
  unless(n % 2, function() {
    console.log(n, "is even");
  });
});
```

That's going to print our expected answer

```
0 'is even'
2 'is even'
4 'is even'
6 'is even'
8 'is even'
10 'is even'
. . .
```

```
. . .
94 'is even'
96 'is even'
98 'is even'
```

With the above code we have abstracted away looping, and the condition checks into a simple and concise higher-order function!

Having seen a few examples of higher-order functions, it's time to go into serious mode! In the upcoming section, we will be discussing the real-world higher-order functions and how to create them. So here we go.

■ **Note** All the higher-order functions that we are creating in the current chapter will be in chap03 branch.

Higher-Order Functions in the Real World

In this section we are going to see real-world examples of higher-order functions. We are going to start with simple higher-order functions and slowly grow into complex higher-order functions, which are used by JavaScript developers in their day-to-day lives. Excited? So what are you waiting for? Read on.

■ **Note** The examples will be continued in the next chapters after we introduce the concept of *closures*. Most of the higher-order Functions work with the help of closures.

---

every Function

Often as a JavaScript developer we need to check if the array of content is a number, custom object, or anything else. We usually write our typical for loop approach to solve these problems. But let's abstract these away into a function called *every*. The `every` function takes two arguments: an array and a function. It checks if all the elements of the array are evaluated to `true` by the passed function. The implementation looks like this as shown in Listing 3-8:

Listing 3-8. every Function Definition

```
const every = (arr,fn) => {
    let result = true;
    for(let i=0;i<arr.length;i++)
       result = result && fn(arr[i])
    return result
}
```

Here we are simply iterating over the passed `array` and calling the `fn` by passing the current content of the array element at the iteration. Note that the passed `fn` should be returning a Boolean value. Then we do a `&&` to make sure all the contents of the array are obeying the criteria that is given by the `fn`.

We need to quickly check that our `every` function works fine. Then pass on the array of `NaN` and pass `fn` as `isNaN`, which does check if the given number is `NaN` or not:

```
every([NaN, NaN, NaN], isNaN)
=> true
every([NaN, NaN, 4], isNaN)
=> false
```

Great. The `every` is a typical higher-order function that is easy to implement and at the same time it's very useful, too! Before we go further, we need to make ourselves comfortable with the new `for..of` loop, which is a part of ES6 specifications. The `for..of` loops can be used to iterate the array elements. Let's rewrite our `ever` function with `for` loop (Listing 3-9:

Listing 3-9. every Function with for-of loop

```
const every = (arr,fn) => {
    let result = true;
    for(const value of arr)
       result = result && fn(value)
    return result
}
```

The `for..of` loop is just an abstraction over our old `for` loop. As you can see here, the `for..of` has eliminated the traversing of an array by hiding the index variable, etc. We have abstracted away `for..of` with `every`. It's all about abstraction. What if the next version of JavaScript changes the way of `for..of`? We just need to change it in the `every` function. This is one of the biggest advantages of abstraction.

some Function

Similar to `every` function, we also have a function called `some`. The `some` works quite the opposite way of `every` function such that the `some` function returns `true` if either one of the elements in the array returns true for the passed function. The `some` function is also called as `any` function. In order to implement `some` function we use `||` rather than `&&` :

Listing 3-10. some Function Definition

```
const some = (arr,fn) => {
    let result = false;
    for(const value of arr)
       result = result || fn(value)
    return result
}
```

■ **Note** Both every and some functions are inefficient implementations. Every should traverse the array until the first element that doesn't match the criteria, and some should traverse the array only until the first match. For large arrays they will be inefficient. Remember that we are trying to understand the concepts of higher-order functions in this chapter rather than writing code for efficiency and accuracy.

With some function in place, we can go and check its result by passing the array's like this:

```
some([NaN,NaN, 4], isNaN)
=>true
some([3,4, 4], isNaN)
=>false
```

Having seen both some and every function, let's go and look at the sort function and how a higher-order function plays an important role there.

sort Function

The sort is an in-built function that is available in the Array prototype of JavaScript. Suppose we need to sort a list of fruits:

```
var fruit = ['cherries', 'apples', 'bananas'];
```

you can simply call the sort function that is available on the Array prototype:

```
fruit.sort()
=> ["apples", "bananas", "cherries"]
```

That's so simple. The sort function is a higher-order function that takes up a function as its argument, which will help the sort function to decide the sorting logic. Simply put, the signature of the sort function looks like this:

```
arr.sort([compareFunction])
```

Here the compareFunction is optional. If the compareFunction is not supplied, elements are sorted by converting them to strings and comparing strings in Unicode code point order. You don't need to worry about Unicode conversion in this section as we are more focused on the higher-order functions. The important point to note here is that in order to compare the element with our own logic while sorting is performed, we need to pass our compareFunction. We can sense how the sort function is designed to be so flexible in such a way that it can sort any data on the JavaScript world, provided we pass a compareFunction. The sort function is flexible due to the nature of higher-order functions!

Before writing our `compareFunction`, let's see what it should really implement. The `compareFunction` should implement the following logic as mentioned here : https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/sort

Listing 3-11. Skeleton of compare Function

```
function compare(a, b) {
  if (a is less than b by some ordering criterion) {
    return -1;
  }
  if (a is greater than b by the ordering criterion) {
    return 1;
  }
  // a must be equal to b
  return 0;
}
```

As a simple example, imagine we have a list of `people`:

```
var people = [
    {firstname: "aaFirstName", lastname: "cclastName"},
    {firstname: "ccFirstName", lastname: "aalastName"},
    {firstname:"bbFirstName", lastname:"bblastName"}
];
```

Now we need to sort `people` using `firstname` key in the object, then we need to pass on our own `compareFunction` like this:

```
people.sort((a,b) => { return (a.firstname < b.firstname) ? -1 :
(a.firstname > b.firstname) ? 1 : 0 })
```

which is going to return the following data:

```
 [ { firstname: 'aaFirstName', lastname: 'cclastName' },
  { firstname: 'bbFirstName', lastname: 'bblastName' },
  { firstname: 'ccFirstName', lastname: 'aalastName' } ]
```

Sorting with respect to `lastname` looks like this:

```
people.sort((a,b) => { return (a.lastname < b.lastname) ? -1 : (a.lastname >
b.lastname) ? 1 : 0 })
```

will return:

```
[ { firstname: 'ccFirstName', lastname: 'aalastName' },
  { firstname: 'bbFirstName', lastname: 'bblastName' },
  { firstname: 'aaFirstName', lastname: 'cclastName' } ]
```

Hooking again into the logic of compareFunction:

```
function compare(a, b) {
  if (a is less than b by some ordering criterion) {
    return -1;
  }
  if (a is greater than b by the ordering criterion) {
    return 1;
  }
  // a must be equal to b
  return 0;
}
```

Having known the algorithm for our compareFunction, can we do it better? Rather than writing the compareFunction every time, can we abstract away the above logic into a function? As you can see in the above example, we wrote two functions each for comparing firstName and lastName with almost the same duplicate code. Let's solve this problem with our higher-order function. Now the function that we are going to design won't take function as its argument but rather return a function. (Remember HOC can also return a function).

Let's call this function sortBy, which allows the user to sort the array of objects based on the passed property as shown below in Listing 3-12:

Listing 3-12. sortBy function Definition

```
const sortBy = (property) => {
    return (a,b) => {
        var result = (a[property] < b[property]) ? -1 : (a[property] >
b[property]) ? 1 : 0;
        return result;
    }
}
```

The sortBy function takes an argument named property and returns a new function that takes two arguments:

```
. . .
        return (a,b) => { }
. . .
```

The returned function has a very simple function body that clearly tells the compareFunction logic:

```
. . .
(a[property] < b[property]) ? -1 : (a[property] > b[property]) ? 1 : 0;
. . .
```

Imagine we are going to call the function with the property name `firstname, and` then the function body with the replaced `property` argument looks like the one below:

```
(a,b) => return (a['firstname'] < b['firstname']) ? -1 : (a['firstname'] >
b['firstname']) ? 1 : 0;
```

That's exactly what we did by manually writing a function. :) Here is our `sortBy` function in action:

```
people.sort(sortBy("firstname"))
```

will return:

```
[ { firstname: 'aaFirstName', lastname: 'cclastName' },
  { firstname: 'bbFirstName', lastname: 'bblastName' },
  { firstname: 'ccFirstName', lastname: 'aalastName' } ]
```

Sorting with respect to `lastname` looks like this:

```
people.sort(sortBy("lastname"))
```

returns:

```
[ { firstname: 'ccFirstName', lastname: 'aalastName' },
  { firstname: 'bbFirstName', lastname: 'bblastName' },
  { firstname: 'aaFirstName', lastname: 'cclastName' } ]
```

as before!

Wow, that's truly amazing! The `sort` function takes the `compareFunction`, which is returned by the `sortBy` function! That's a lot of higher-order functions floating around! Again we have abstracted away the logic behind `compareFunction` leaving the user to focus on what he or she really needs. After all, a *higher-order function is all about abstractions!*

But pause for a moment here and think about the `sortBy` function. Remember that our `sortBy` function takes a property and returns another function. The returned function is what passed as `compareFunction` to our `sort` function. The question here is how come the returned function carries the `property` argument value that we have passed?

Welcome to the world of closures! The `sortBy` function works just because JavaScript supports closures. We need to clearly understand what closures are before we go ahead and write higher-order functions. Closures will be the topic of the next chapter.

Remember though, we will be writing our real-world higher-order function after explaining closures in the next chapter!

Summary

We started with simple data types that JavaScript supports. We found that `Function` is also a data type in JavaScript. Thus, we can keep functions in all the places where we can keep our data. Put in other words, `Function` can be stored, passed, and reassigned like other data types in JavaScript. This extreme feature of JavaScript allows the `Function` to be passed over to another function, which we call a Higher-Order Function. Remember that a Higher-Order Function is a function that takes another function as its argument or returns a function. We saw a handful of examples in this chapter showcasing how these Higher-Order Function concepts help the developer to write the code that abstracts away the difficult part! We have created and added a few such functions in our own library! We concluded the chapter by mentioning that Higher-Order Functions work with the blessing of another important concept in JavaScript called Closures. Closures will be the topic of Chapter 4!

CHAPTER 4

Closures and Higher-Order Functions

■ **Note** The chapter examples and library source code are in branch `chap04`. The repo's URL is: https://github.com/antoaravinth/functional-es6.git

Once checkout the code, please checkout branch `chap04`:

```
...
git checkout -b chap04 origin/chap04
...
```

For running the codes, as before run:

```
...
npm run playground
...
```

In the previous chapter we saw how higher-order functions help us as a developer to create abstraction over the common problems! It's a very powerful concept as we learned. We have created our `sortBy` higher-order function to showcase a valid and relevant example of the use case. Even though the `sortBy` function is working on the basis of higher-order functions (which is again the concept of passing functions as arguments to the other functions), it has something to do with yet another concept called Closures in JavaScript.

Closures is a concept that we need to understand in the JavaScript world before we go further in our journey of functional programming techniques. And that's where this chapter comes into the picture. In this chapter we are going to discuss in detail what is meant by closures and at the same time continue our journey of writing useful and real-world higher-order functions. The concept of closures has to do with scopes in JavaScript. So let's get started with closures in the next section.

A. Aravinth, *Beginning Functional JavaScript*, DOI 10.1007/978-1-4842-2656-8_4

Understanding Closures

In this section we are going to see what we mean by closures with a simple example and then move on to our `sortBy` function by unwrapping how it works with closures.

What Are Closures?

Simply put closure is an inner function. So what is an inner function? Well, its a function within an another function. Something like the following:

```
function outer() {
   function inner() {
   }
}
```

Yes, thats exactly what a closure is. The function `inner` is called a closure function. The reason why closure is so powerful is because of its access to the scope chains (or scope levels). We will be talking about scope chains in this section.

■ **Note** Scope chains and Scope levels are used interchangeably in this chapter.

Technically the closure has access to three scopes:

1. Variables that are declared in its own declaration
2. Access to the global variables.
3. Access to the outer function's variable (interesting!)

Lets talk about theses three points separately with simple example. Consider the following snippet:

```
function outer() {
   function inner() {
        let a = 5;
        console.log(a)
   }
   inner() //call the inner function.
}
```

what will be printed to the console when `inner` function gets called? The value will be 5. This is mainly due to the point number 1. A closure function can access all the variables declared in its own declaration (see point 1). No rocket science here!

■ **Note** A strong take away from the above code snippet is that, `inner` function won't be visible outside `outer` function! Go ahead and test it.

Now modifying the above code snippet to the following:

```
let global = "global"
function outer() {
   function inner() {
        let a = 5;
        console.log(global)
   }
   inner() //call the inner function.
}
```

now when `inner` function executed, it does print the value `global`. Thus closures can access the global variable (see point 2).

Points 1 and 2 are now clear with the example. The 3rd point is very interesting, the claim can be seen in the following code:

```
let global = "global"
function outer() {
   let outer = "outer"
   function inner() {
        let a = 5;
        console.log(outer)
   }
   inner() //call the inner function.
}
```

now when `inner` function executed, it does print the value `outer`. This looks reasonable, but its a very important property of a closure.

Closure has the access to the outer function's variable(s). Here outer function mean, the function which encloses the closure function.

This property is what make the closures so powerful!

■ **Note** Closure also has access to the enclosing function parameters. Try adding a paramater to our `outer` function and try to access it from `inner` function. I will wait here till your done with this small exercise.

Remembering Where It Is Born

In the previous section we saw what a closure is. Now we will be seeing slightly a complicated example, which explains yet another important concept in closure -- closure remembering its context!

Take a look at the following code:

```
var fn = (arg) => {
        let outer = "Visible"
        let innerFn = () => {
                console.log(outer)
                console.log(arg)
        }
        return innerFn
}
```

The code is simple. The `innerFn` is a closure function to `fn` and `fn` returns the `innerFn` when called. Nothing fancy here.

Lets play around with `fn`:

```
var closureFn = fn(5);
closureFn()
```

will print the following:

```
Visible
5
```

How does calling `closureFn` prints `Visible` and `5` to the console? What is happening behind the scenes? Lets break it down.

There are two steps happening in this case:

1. When the below line is called:

   ```
   var closureFn = fn(5);
   ```

   our `fn` gets called with argument `5`. As per our `fn` definition, it returns the `innerFn`.

2. This where interesting things happens. When `innerFn` is returned, the javascript execution engine sees `innerFn` as a closure and sets its scope accordingly. As we have seen in the previous section, closures will have access to the 3 scope level. All these 3 scope level are set (`arg`, `outer` values will be set in scope level of `innerFn`) when the `innerFn` is returned! The returned function reference is stored in `closureFn`. Thus `closureFn` will have remember `arg`, `outer` values when called via scope chains!

3. When we finally call the `closureFn`:

   ```
   closureFn()
   ```

   it prints:

   ```
   Visible
   5
   ```

As now you can guess it out, `closureFn` remembers its context (the scopes i.e `outer` and `arg`) when it born in the second step! Thus the calls to `console.log` print appropriately.

You might be wondering, what is the use case of closure? We have already seen it in action in our sortBy function. Let's quickly revisit them.

Revisiting `sortBy` Function

Let's quickly revisit the `sortBy` function that we have defined and used in the previous chapter:

```
const sortBy = (property) => {
    return (a,b) => {
        var result = (a[property] < b[property]) ? -1 : (a[property] >
b[property]) ? 1 : 0;
        return result;
    }
}
```

When we called `sortBy` function like this:

```
sortBy("firstname")
```

this is what happened:

`sortBy` returned a new function that takes two argument like this:

```
(a,b) => { /* implementation */ }
```

Now we are comfortable with closures and we are aware that the returned function will have access to the `sortBy` function argument `property`. Since this function will be returned only when `sortBy` is called, the `property` argument is revolved with a value; hence the returned function will carry this *context* throughout its life:

```
//scope it carries via closure
property = "passedValue"
(a,b) => { /* implementation */ }
```

Now since the returned function carries the value of `property` in its context, it will use the returned value where it is appropriate and when it is needed! Now with that explanation in place, we can fully understand how closures and higher-order functions that allow us to write a function like `sortBy` that is going to abstract away the inner details, moving ahead to our functional world!

That's a lot to take in for this section; in the next section we will be continuing our journey of writing more abstract functions using closures and higher-order functions.

Higher-Order Functions in the Real World (Continued)

With our understanding of closures in place, we will go ahead and implement some useful higher-order functions that are used in the real world.

tap Function

Since we are going to deal with lots of functions in functional the programming world, we need a way to debug what is happening between them. As we have seen in previous chapters, we are designing the functions, which take arguments and returns another function, which again takes a few arguments, etc., and so on.

Let's design a simple function called `tap`:

```
const tap = (value) =>
  (fn) => (
    typeof(fn) === 'function' && fn(value),
    console.log(value)
  )
```

Here the `tap` function takes a `value` and returns a function that has the closure over `value` and it will be executed.

■ **Note** In JavaScript the `(exp1,exp2)` means it will execute the two arguments and return the result of the second expression, which is `exp2`. In our above example, the syntax will call the function `fn` and also prints the `value` to the console.

Let's play around with the `tap` function:

```
tap("fun")((it) => console.log("value is ",it))
=>value is fun
=>fun
```

As you can see in the above example, the value '`value is fun`' gets printed and then the value '`fun`' is printed. Easy and straightforward.

So where can the `tap` function be used? Let's imagine you are iterating an array that has data come from a server. You are iterating the array and you feel that the data is wrong so you want to debug and see what the array really contains, while iterating. How will you do that? Nope, don't be imperative, let's be functional. This is where the `tap` function comes into the picture. For the current scenario, we can do this:

```
forEach([1,2,3],(a) =>
   tap(a)(() =>
     {
       console.log(a)
     }
   )
)
```

which does print the value as expected. A simple but yet powerful function in our toolkit.

unary Function

There is a default method in the `array` prototype called `map`. Don't worry; we are going to discover a whole lot of functional functions for array in the next chapter, where we will be seeing how to create our own `map`, too. For now, `map` is a function, which is very similar to the `forEach` function we have defined. The only difference is that `map` returns the result of the callback function.

To get the gist of it, let's say we want to double the array and get back the result; then using the `map` function, we can do like this:

```
[1, 2, 3].map((a) => { return a * a })
=>[1, 4, 9]
```

The interesting point to note over here is that the `map` calls the function with three arguments, which are `element, index, and arr`. Imagine we want to parse the array of strings to the array of int; we have an in-built function called `parseInt` that takes two argument `parses` and `radixes` and converts the passed `parse` into a number if possible. If we pass the `parseInt` to our `map` function, `map` will pass the `index` value to the `radix` argument of `parseInt`, which will result in unexpected behavior.

```
['1', '2', '3'].map(parseInt)
=>[1, NaN, NaN]
```

Oops! As you can see in the above result, the array `[1, NaN, NaN]` is not what we expect. Here we need to convert the `parseInt` function to another function that will be expecting only one argument. How can we achieve that? Meet our next friend, `unary` function. The task of `unary` function is to take the given function with n argument and convert it into a single argument.

Our `unary` function looks like the following:

```
const unary = (fn) =>
  fn.length === 1
    ? fn
    : (arg) => fn(arg)
```

We are checking if the passed `fn` has an argument list of size `1` (which we can find via `length` property); if so we are not going to do anything. If not, we return a new function, which takes only one argument `arg` and calls the function with that argument.

To see our `unary` function in action, we can rerun our problem with `unary`:

```
['1', '2', '3'].map(unary(parseInt))
=>[1, 2, 3]
```

Here our `unary` function returns a new function (a clone of `parseInt`), which is going to take only one argument! Thus the map function passing `index`, `arr` argument, becomes unaffected as we are getting back the expected result.

■ **Note** There are also functions like `binary`, etc., which will convert the function to accept corresponding arguments.

The next two functions that we are going to see are special higher-order functions, which will allow the developer to control the number of times the function is getting called. They have a lot of use cases in the real world.

once Function

There are a lot of situations in which we need to run a given function only once. This scenario occurs to JavaScript developers in their day-to-day life, as they want to set up a third-party library only once, initiate the payment set up only once, do a bank payment request only once, etc. These are common cases that the developers face.

In this section we are going to write a higher-order function called `once`, which will allow the developer to run the given function only once! Again the point to note over here is that we have keep on abstracting away our day-to-day activities into our functional toolkits!

```
const once = (fn) => {
  let done = false;

  return function () {
    return done ? undefined : ((done = true), fn.apply(this, arguments))
  }
}
```

The above `once` function takes an argument `fn` and returns the result of it by calling it with the `apply` method (note on the `apply` method is down below). The important point to note here is that we have declared a variable called `done` and set it to `false` initially. The returned function will have a closure scope over it; hence it will access it to check if `done` is true, if return `undefined` else set `done` to true (thus preventing next time execution) and calling the function with necessary arguments.

■ **Note** The `apply` function will allow us to set the context for the function and also pass on the arguments for the given function. You can find more about it over here: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/apply

With the `once` function in place, we can go and do a quick check of it.

```
var doPayment = once(() => {
   console.log("Payment is done")
})

doPayment()
=>Payment is done

//oops bad, we are doing second time!
doPayment()
=>undefined!
```

The above code snippet showcases the function `doPayment` that is wrapped over `once` will be executed only once regardless how many times we call them! The `once` is a simple but effective function in our toolkit!

Memoize Function

Before we close this exciting section, let's see my favorite function called `memoize`. We know that the pure function is all about working on its argument and nothing else. They don't depend on the outside world for anything. The results of the pure function are purely based on its argument only. Imagine that we have a pure function called `factorial`, which calculates the factorial for the given number. The function looks like the following:

```
var factorial = (n) => {
  if (n === 0) {
    return 1;
  }

  // This is it! Recursion!!
  return n * factorial(n - 1);
}
```

You can quickly check that `factorial` function with a few inputs:

```
factorial(2)
=>2
factorial(3)
=>6
```

Nothing fancy here. But we knew that the factorial of the value 2 is 2, 3 is 6, and so on. Mainly because we know the `factorial` function does work but only based on its argument and nothing else! So the question arises here: why can't we store back the result for each input (some sort of an object) and give back the output if the input is already present in the object? And moreover for calculating the factorial for 3, we need to calculate the factorial for 2, so why can't we reuse those calculations in our function? Yup, that's exactly what the `memoize` function is going to do. The `memoize` is a special higher-order function that allows the function to remember or memorize its result. :)

Let's see how we can implement such a function in JavaScript. No worries; it is as simple as it looks - like the following:

```
const memoized = (fn) => {
  const lookupTable = {};

  return (arg) => lookupTable[arg] || (lookupTable[arg] = fn(arg));
}
```

Here in the above function we have a local variable called `lookupTable` that will be in the closure context for the returned function. This will take the argument and check if that argument is in the `lookupTable` :

```
. . lookupTable[arg] . .
```

if so return the value else, update the object with new input as key and result from `fn` as its value:

```
(lookupTable[arg] = fn(arg))
```

Perfect. Now we can go and wrap our `factorial` function into a `memoize` function to keep remembering its output:

```
let fastFactorial = memoized((n) => {
  if (n === 0) {
    return 1;
  }

  // This is it! Recursion!!
  return n * fastFactorial(n - 1);
})
```

Now go and call `fastFactorial`:

```
fastFactorial(5)
=>120
=>lookupTable will be like: Object {0: 1, 1: 1, 2: 2, 3: 6, 4: 24, 5: 120}
fastFactorial(3)
=>6 //returned from lookupTable
fastFactorial(7)
```

```
=> 5040
=>lookupTable will be like: Object {0: 1, 1: 1, 2: 2, 3: 6, 4: 24, 5: 120,
6: 720, 7: 5040}
```

It is going to work the same way, but now much faster than before. While running the `fastFactorial`, I would like you to inspect the `lookupTable` object and how it helps in speeding things up as shown in the above snippet!

That is the beauty about the higher-order function – closure and pure functions in action!

■ **Note** That our `memoized` function is written for the functions that take up only one argument. Can you come up with a solution for all functions with n number of arguments?

We have abstracted away many common problems into higher-order functions that allowed us to write the solution with elegant and ease.

Summary

We started this chapter with a set of questions on what a function can see. By starting small and building up examples, we understand how closures make the function to remember the context where it is born. With this understanding in place, we went ahead and implemented few more higher-order functions that are used in the day-to-day life of a JavaScript programmer. Throughout we have seen how to abstract away the common problems into a specific function and reuse them! Now we understand the importance of Closures, Higher-Order Functions, Abstraction, and Pure Functions! In the next chapter we are going to continue building the higher-order functions, but with respect to Arrays!

CHAPTER 5

Being Functional on Arrays

■ **Note** The chapter examples and library source code are in branch `chap05`. The repo's URL is: `https://github.com/antoaravinth/functional-es6.git`

Once checkout the code, please checkout branch `chap05`:

```
...
git checkout -b chap05 origin/chap05
...
```

For running the codes, as before run:

```
...
npm run playground
...
```

Welcome to the chapter on Arrays and Objects. In this chapter we are going to continue our journey of exploring higher-order functions that are useful for `arrays`.

Arrays are used literally in our JavaScript programming world. We use them to store data, manipulate data, find data, and convert (project) the data to another format. In this chapter we are going to see how to improve all these activities using our functional programming techniques learned so far.

We will be creating a set of functions on Array, and we will be solving the common problems *functionally* rather than imperatively.

■ **Note** The functions that we are creating in this chapter may or may not be defined already in the Array/Object prototype. It's been advised these are for understanding how the real functions themselves work, rather than overriding them.

A. Aravinth, *Beginning Functional JavaScript*, DOI 10.1007/978-1-4842-2656-8_5

Working Functionally on Arrays

In this section we are going to create a set of useful functions, and using them we are going to solve the common problems with `Array`.

■ **Note** All the functions that we are going to create in this section are called *Projecting functions*. Applying a function to a value and creating a new value is called a projection. Don't get worried about the term, it will make sense when we see our first projecting function `map`.

---

map

We have already seen how to iterate over the `Array` using `forEach`. `forEach` is a higher-order function, which is going to iterate over the given array and call the passed function with the current index as its argument. `forEach` hides away the common problem of iteration. But we can't use `forEach` in all cases.

Imagine we want to square all the contents of the `array` and get back the result in a new array. How we can achieve this using `forEach`? Using `forEach` we can't return the data; instead it just executes the passed function. And that's where our first projecting function comes into the picture, and it's called `map`.

Implementing `map` is an easy and straightforward task given that we have already seen how to implement `forEach` itself. The implementation of `forEach` looks like what is shown in Listing 5-1:

Listing 5-1. forEach Function Definition

```
const forEach = (array,fn) => {
   for(const value of arr)
      fn(value)
}
```

`map` function implementation looks like that below:

Listing 5-2. map Function Definition

```
const map = (array,fn) => {
        let results = []
        for(const value of array)
                  results.push(fn(value))

        return results;
}
```

The map implementation looks very similar to forEach; it's just that we are capturing the results in a new array as:

```
. . .
        let results = []
. . .
```

and returning the results from the function. Now it's a good time to talk about the word *projecting function*. We have mentioned earlier that the map function is a projecting function. Why do we call the map function so? The reason is quite simple and straightforward; since map returns the *transformed* value of the given function, we call them projecting functions. Of course, few people do call map a transforming function. But we are going to stick to the term projection (which I feel is very good!).

Now let's go and solve the problem of squaring the contents of the array using our map function defined in Listing 5-2.

```
map([1,2,3], (x) => x * x)
=>[1,4,9]
```

As you can see in the above code snippet, we have achieved our task with simple elegance. Since we are going to create many functions, which are specifically to the Array type, we are going to wrap all the functions into a const called arrayUtils and export arrayUtils.

So it typically looks like the following (Listing 5-3):

Listing 5-3. Wrapping Functions into arrayUtils Object

```
//map function from Listing 5-2
const map = (array,fn) => {
  let results = []
  for(const value of array)
      results.push(fn(value))

  return results;
}

const arrayUtils = {
  map : map
}

export {arrayUtils}

//another file
import arrayUtils from 'lib'
arrayUtils.map //use map

//or

const map = arrayUtils.map
//so that we can call them map!
```

■ **Note** In the text, however, we are going to call them as `map` rather than `arrayUtils.map` for clarity purposes.

Perfect. In order to make the chapter examples more realistic, we are going to build an array of objects, which looks as shown below in Listing 5-4:

Listing 5-4. apressBooks Object Describing Book Details

```
let apressBooks = [
        {
                "id": 111,
                "title": "C# 6.0",
                "author": "ANDREW TROELSEN",
                "rating": [4.7],
                "reviews": [{good : 4 , excellent : 12}]
        },
        {
                "id": 222,
                "title": "Efficient Learning Machines",
                "author": "Rahul Khanna",
                "rating": [4.5],
                "reviews": []
        },
        {
                "id": 333,
                "title": "Pro AngularJS",
                "author": "Adam Freeman",
                "rating": [4.0],
                "reviews": []
        },
        {
                "id": 444,
                "title": "Pro ASP.NET",
                "author": "Adam Freeman",
                "rating": [4.2],
                "reviews": [{good : 14 , excellent : 12}]
        }
];
```

■ **Note** Kindly note that the array does contain the real titles that are published by Apress. But the review key values are my own interpretations.

Now all the functions that we are going to create in this chapter will be run for the above array of objects. Now suppose we need to get the array of object, which only has a `title` and `author` name in it? How are we going to achieve the same using `map` function? Do you see a solution that is running in your mind?

The solution is so simple using the `map` function, which looks like the following:

```
map(apressBooks,(book) => {
        return {title: book.title,author:book.author}
})
```

which is going to return the result as you would expect. The object in the returned array will be having only two properties: one is `title` and another one is `author`, as you have specified in your function:

```
[ { title: 'C# 6.0', author: 'ANDREW TROELSEN' },
  { title: 'Efficient Learning Machines', author: 'Rahul Khanna' },
  { title: 'Pro AngularJS', author: 'Adam Freeman' },
  { title: 'Pro ASP.NET', author: 'Adam Freeman' } ]
```

Not always do we just want to transform *all* our array contents into a new one. Rather we want to filter the content of array and then do the transformation!

Meet the next function in the queue called `filter`.

filter

Imagine we want to get the list of books whose rating is more than `4.5`? How we are going to achieve this? Definitely not a problem for `map` to solve. But we need a similar to `map`, which just checks a condition, before pushing the results into the results array.

So first we'll take another look at the `map` function (from Listing 5-2):

```
const map = (array,fn) => {
  let results = []
  for(const value of array)
      results.push(fn(value))

  return results;
}
```

Now here we need to check a condition or predicate before we do this:

```
. . .
        results.push(fn(value))
. . .
```

so let's add that into a separate function called `filter as shown in Listing 5-5`:

Listing 5-5. filter Function Definition

```
const filter = (array,fn) => {
  let results = []
  for(const value of array)
     (fn(value)) ? results.push(value) : undefined

  return results;
}
```

Now with the `filter` function in place, we can solve our problem in hand like the following way:

```
filter(apressBooks, (book) => book.rating[0] > 4.5)
```

which is going to return you the expected result:

```
[ { id: 111,
    title: 'C# 6.0',
    author: 'ANDREW TROELSEN',
    rating: [ 4.7 ],
    reviews: [ [Object] ] } ]
```

That's perfect! We are constantly improving the way to deal with arrays using these higher-order functions. Before we go further looking into the next functions on the array, we are going to see how to chain the projection function (`map`,`filter`) to get our desired results in complex situations.

Chaining Operations

It's always the case that we need to *chain* lots of functions to achieve our goal. For example, imagine the problem of getting the `title` and `author` object out of our `apressBooks` for which the review is greater than `4.5`. The initial step to tackle this problem is to solve via `map` and `filter`; the code might look like this:

```
let goodRatingBooks =
 filter(apressBooks, (book) => book.rating[0] > 4.5)

map(goodRatingBooks,(book) => {
        return {title: book.title,author:book.author}
})
```

which is going to return the result as expected:

```
[ {
        title: 'C# 6.0',
    author: 'ANDREW TROELSEN'
    }
]
```

An important point to note here is that both `map` and `filter` are projection functions. So they always return a data after applying the transformation (via the passed higher-order function) on the `array`. So we can chain both `filter` and `map` (the order is very important) to get the task done (without the need for additional variables – i.e., `goodRatingBooks`):

```
map(filter(apressBooks, (book) => book.rating[0] > 4.5),(book) => {
        return {title: book.title,author:book.author}
})
```

The above code literally tells the problem we are solving: "Map over the filtered array whose rating is 4.5 and return their `title` and `author` keys in an object!" Due to the nature of both `map` and `filter` we have abstracted away the details of array themselves, and we started focusing on the problem in hand.

We will be seeing examples of chaining methods in the upcoming sections.

■ **Note** We will be seeing another way to achieve the same thing via function composition in Chapter X (TODO: Mention it)

---

concatAll

Let's now tweak the `apressBooks` a bit, so that we have a data structure that looks like the following as shown in Listing 5-6:

Listing 5-6. Updated apressBooks Object with Book Details

```
let apressBooks = [
        {
                name : "beginners",
                bookDetails : [
                        {
                                "id": 111,
                                "title": "C# 6.0",
                                "author": "ANDREW TROELSEN",
                                "rating": [4.7],
                                "reviews": [{good : 4 , excellent : 12}]
                        },
                        {
                                "id": 222,
                                "title": "Efficient Learning Machines",
                                "author": "Rahul Khanna",
                                "rating": [4.5],
                                "reviews": []
                        }
                ]
        },
```

```
        {
            name : "pro",
            bookDetails : [
                        {
                                "id": 333,
                                "title": "Pro AngularJS",
                                "author": "Adam Freeman",
                                "rating": [4.0],
                                "reviews": []
                        },
                        {
                                "id": 444,
                                "title": "Pro ASP.NET",
                                "author": "Adam Freeman",
                                "rating": [4.2],
                                "reviews": [{good : 14 , excellent : 12}]
                        }
                ]
        }
];
```

Now let's take up the same problem that we had in the previous section - to get the `title` and `author` for the books whose rating is above `4.5`. We can start solving the problem by first mapping over data:

```
map(apressBooks,(book) => {
        return book.bookDetails
})
```

which is going to return us the value:

```
[ [ { id: 111,
      title: 'C# 6.0',
      author: 'ANDREW TROELSEN',
      rating: [Object],
      reviews: [Object] },
    { id: 222,
      title: 'Efficient Learning Machines',
      author: 'Rahul Khanna',
      rating: [Object],
      reviews: [] } ],
  [ { id: 333,
      title: 'Pro AngularJS',
      author: 'Adam Freeman',
      rating: [Object],
      reviews: [] },
```

```
    { id: 444,
      title: 'Pro ASP.NET',
      author: 'Adam Freeman',
      rating: [Object],
      reviews: [Object] } ] ]
```

As you can see, the return data from our `map` function contains `Array` inside `Array`. It's because our `bookDetails` itself is an array. Now if we pass the above data to our `filter`, we are going to have problems, as `filters` can't work on nested arrays!

And that's where `concatAll` function comes in! The job of `concatAll` function is simple enough that it needs to concatenate all the nested arrays into a single array! You can also call `concatAll` as a `flatten` method. The implementation of `concatAll` looks like the following (Listing 5-7):

Listing 5-7. concatAll function Definition

```
const concatAll = (array,fn) => {
  let results = []
  for(const value of array)
     results.push.apply(results, value);

  return results;
}
```

Here we just pushed up the inner array while iterating into our `results` array.

■ **Note** We have used JavaScript Function's `apply` method to set the `push` context to `results` itself and passing the argument as the current index of the iteration - `value`.

The main goal of 'concatAll' is to un-nest the nested arrays into a single array. The below code explains the concept in action:

```
concatAll(
        map(apressBooks,(book) => {
                return book.bookDetails
        })
)
```

which is going to return us the result we expected:

```
[ { id: 111,
   title: 'C# 6.0',
   author: 'ANDREW TROELSEN',
   rating: [ 4.7 ],
   reviews: [ [Object] ] },
```

```
 { id: 222,
   title: 'Efficient Learning Machines',
   author: 'Rahul Khanna',
   rating: [ 4.5 ],
   reviews: [] },
 { id: 333,
   title: 'Pro AngularJS',
   author: 'Adam Freeman',
   rating: [ 4 ],
   reviews: [] },
 { id: 444,
   title: 'Pro ASP.NET',
   author: 'Adam Freeman',
   rating: [ 4.2 ],
   reviews: [ [Object] ] } ]
```

Now we can go ahead and easily do a `filter` with our condition like this:

```
let goodRatingCriteria = (book) => book.rating[0] > 4.5;
filter(
        concatAll(
                map(apressBooks,(book) => {
                        return book.bookDetails
                })
        )
,goodRatingCriteria)
```

which is going to return the expected value:

```
[ { id: 111,
   title: 'C# 6.0',
   author: 'ANDREW TROELSEN',
   rating: [ 4.7 ],
   reviews: [ [Object] ] } ]
```

Brilliant! We have seen how designing a higher-order function within the world of Array does solve a lot of problems in elegant fashion. We have done a really good job up to now. We still have to see a few more functions with respect to Array in the upcoming sections.

Reducing Function

If you talk about functional programming anywhere, you often hear about the term *reduce functions*. What are they? Why they are so useful? `reduce` is a beautiful function that is designed for keeping the power of closure in JavaScript. In this section, we are going to see the usefulness of reducing an array.

reduce Function

In order to give a solid example of `reduce` function and where it's been used, let's look at the problem of finding the summation of the given array. To start with, suppose we have an array called '':

```
let useless = [2,5,6,1,10]
```

We need to find the sum of the given above array, but how we can achieve that? A simple solution would be the following:

```
let result = 0;
forEach(useless,(value) => {
   result = result + value;
})
console.log(result)
=> 24
```

With the above problem, we are reducing the array (which has several data) into a single value. We start with a simple *accumulator*; in this case we call it as `result` to store our summation result while traversing the array itself. Note that we have set the `result` value to default `0` in case of summation. But what if we need to find the product of all the elements in the given array? In that case we will be setting up the `result` value to `1`. This whole process of setting up the accumulator and traversing the array (remembering the previous value of accumulator) to produce a single element is called reducing an array.

Since we are going to repeat the above process for all array-reducing operations, can't we abstract away these into a function? You can – that's where `reduce` function comes in. The implementation of our `reduce` function looks like the following shown in Listing 5-8:

Listing 5-8. reduce Function First Implementation

```
const reduce = (array,fn) => {
        let accumlator = 0;
        for(const value of array)
                accumlator = fn(accumlator,value)

        return [accumlator]
}
```

Now with `reduce` function in place, we can solve our summation problem using it like this:

```
reduce(useless,(acc,val) => acc + val)
=>[24]
```

Great. But what if we want to find a product of the given array? Our `reduce` function is going to fail, mainly due to the fact that we are using an `accumulator` value to `0`. So our product result will be `0` too:

```
reduce(useless,(acc,val) => acc * val)
=>[0]
```

We can solve this by rewriting our `reduce` function from Listing 5-8 such that it takes an argument for setting up the initial value for the accumulator. Let's do this right away in Listing 5-9:

Listing 5-9. reduce Function Final Implementation

```
const reduce = (array,fn,initialValue) => {
        let accumlator;

        if(initialValue != undefined)
                accumlator = initialValue;
        else
                accumlator = array[0];

        if(initialValue === undefined)
                for(let i=1;i<array.length;i++)
                        accumlator = fn(accumlator,array[i])
        else
                for(const value of array)
                accumlator = fn(accumlator,value)
        return [accumlator]
}
```

We have made the changes to our `reduce` function so that now if `initialValue` is not passed, the `reduce` function will take the first element in the array as its accumulator value. Cool.

■ **Note** Have a look at the two for loop statements. When the initialValue is undefined, we need to start looping the array from the second element, as the first value of the accumulator will be used as the initial value. If the initialValue is passed by the caller, then we need to iterate the full array.

Now let's try our product problem using the `reduce` function:

```
reduce(useless,(acc,val) => acc * val,1)
=>[600]
```

Now we'll use reduce in our running example, apressBooks. Bringing apressBooks (updated in Listing 5-6) in here, for easy reference, we have this:

```
let apressBooks = [
        {
                name : "beginners",
                bookDetails : [
                        {
                                "id": 111,
                                "title": "C# 6.0",
                                "author": "ANDREW TROELSEN",
                                "rating": [4.7],
                                "reviews": [{good : 4 , excellent : 12}]
                        },
                        {
                                "id": 222,
                                "title": "Efficient Learning Machines",
                                "author": "Rahul Khanna",
                                "rating": [4.5],
                                "reviews": []
                        }
                ]
        },
        {
            name : "pro",
            bookDetails : [
                        {
                                "id": 333,
                                "title": "Pro AngularJS",
                                "author": "Adam Freeman",
                                "rating": [4.0],
                                "reviews": []
                        },
                        {
                                "id": 444,
                                "title": "Pro ASP.NET",
                                "author": "Adam Freeman",
                                "rating": [4.2],
                                "reviews": [{good : 14 , excellent : 12}]
                        }
                ]
        }
];
```

On a good day, your boss comes to your desk and asks you to implement the logic of finding the number of good and excellent reviews from our apressBooks. And you think, this is a perfect problem that can be solved easily via reduce function. Remember that our

apressBooks contains array inside array (as we saw in the previous section), so we need to concatAll to make it a flat array. Since reviews are a part of bookDetails, we don't name a key, so we can just map bookDetails and concatAll in the following way:

```
concatAll(
        map(apressBooks,(book) => {
                return book.bookDetails
        })
)
```

Now let's solve our problem using reduce:

```
let bookDetails = concatAll(
        map(apressBooks,(book) => {
                return book.bookDetails
        })
)

reduce(bookDetails,(acc,bookDetail) => {
        let goodReviews = bookDetail.reviews[0] != undefined ? bookDetail.
        reviews[0].good : 0
        let excellentReviews = bookDetail.reviews[0] != undefined ?
        bookDetail.reviews[0].excellent : 0
        return {good: acc.good + goodReviews,excellent : acc.excellent +
        excellentReviews}
},{good:0,excellent:0})
```

which is going to return the following result:

```
[ { good: 18, excellent: 24 } ]
```

Now let's walk down the reduce function to see how this magic happened. The first point to note here is that we are passing an accumulator to an initialValue, which is nothing but:

```
{good:0,excellent:0}
```

In our reduce function body, we are getting the good and excellent review details (from our bookDetail object) and storing it in the corresponing variables namely, goodReviews and excellentReviews:

```
let goodReviews = bookDetail.reviews[0] != undefined ? bookDetail.
reviews[0].good : 0
let excellentReviews = bookDetail.reviews[0] != undefined ? bookDetail.
reviews[0].excellent : 0
```

With that in place, we can walk through our `reduce` function call trace to understand better what's happening. For the first iteration, `goodReviews` and `excellentReviews` will be the following:

```
goodReviews = 4
excellentReviews = 12
```

and our accumulator will be the following:

```
{good:0,excellent:0}
```

as we have passed the initial line. Once `reduce` function executes the line:

```
  return {good: acc.good + goodReviews,excellent : acc.excellent +
excellentReviews}
```

our internal accumulator value gets changed to:

```
{good:4,excellent:12}
```

And we are done with the first iteration of our array. In the second and third iterations, we don't have `reviews;` hence, both `goodReviews` and `excellentReviews` will be `0`, but not affecting our accumulator value, which remains the same:

```
{good:4,excellent:12}
```

and in our final fourth iteration, we will be having `goodReviews` and `excellentReviews` as:

```
goodReviews = 14
excellentReviews = 12
```

and accumulator value being:

```
{good:4,excellent:12}
```

and now when we execute the line:

```
return {good: acc.good + goodReviews,excellent : acc.excellent +
excellentReviews}
```

our accumulator value changes to:

```
{good:18,excellent:28}
```

Since we are done with iterating all our array content, the latest accumulator value will be returned, which is the result!

Wow, as you can see here, in the above process we have abstracted away internal details into higher-order functions, leading to elegant code! Before we close this chapter, let's implement `zip` function, which is another useful function.

Zipping Arrays

Life is not always as easy as you think. We had `reviews` within our `bookDetails` in our `apressBooks` details such that we could easily work with it. However, if data like `apressBooks` does come from the server, they do return data like `reviews` as a separate array, rather than the embedded data, which will look like the following (Listing 5-10):

Listing 5-10. Splitting the apressBooks Object

```
let apressBooks = [
        {
                name : "beginners",
                bookDetails : [
                        {
                                "id": 111,
                                "title": "C# 6.0",
                                "author": "ANDREW TROELSEN",
                                "rating": [4.7]
                        },
                        {
                                "id": 222,
                                "title": "Efficient Learning Machines",
                                "author": "Rahul Khanna",
                                "rating": [4.5],
                                "reviews": []
                        }
                ]
        },
        {
            name : "pro",
            bookDetails : [
                        {
                                "id": 333,
                                "title": "Pro AngularJS",
                                "author": "Adam Freeman",
                                "rating": [4.0],
                                "reviews": []
                        },
                        {
                                "id": 444,
                                "title": "Pro ASP.NET",
                                "author": "Adam Freeman",
                                "rating": [4.2]
                        }
                ]
        }
];
```

Listing 5-11. reviewDetails Object Contains Review Details of the Book

```
let reviewDetails = [
        {
                "id": 111,
                "reviews": [{good : 4 , excellent : 12}]
        },
        {
                "id" : 222,
                "reviews" : []
        },
        {
                "id" : 333,
                "reviews" : []
        },
        {
                "id" : 444,
                "reviews": [{good : 14 , excellent : 12}]
        }
]
```

here in the above snippet (Listing 5-11), the `reviews` are fleshed out into a separate array; they are matched with the book `id`. It's a typical example of how data are segregated into different parts.

But how do we work with these sorts of split data?

zip Function

The task of the `zip` function is to merge two given arrays. As with our example, we need to merge both `apressBooks` and `reviewDetails` into a single array, so that we have all necessary data under a single tree.

The implementation of `zip` looks like the following (Listing 5-12):

Listing 5-12. zip Function Definition

```
const zip = (leftArr,rightArr,fn) => {
        let index, results = [];

        for(index = 0;index < Math.min(leftArr.length, rightArr.
        length);index++)
                results.push(fn(leftArr[index],rightArr[index]));

        return results;
}
```

zip is a very simple function; we just iterate over the two given arrays. Since here we are dealing with two array details, we get the minimum length of the given two arrays using Math.min:

```
. . .
Math.min(leftArr.length, rightArr.length)
. . .
```

Once you get the minimum length, we call our passed higher-order function fn with current leftArr value and rightArr value.

Suppose we want to add the two contents of the array, then we can do via zip like the following:

```
zip([1,2,3],[4,5,6],(x,y) => x+y)
=> [5,7,9]
```

Now let's solve the same problem that we have solved in the previous section. Find the total count of good and excellent review for Apress collection. Since the data are split into two different structures, we are going to use zip to solve our current problem:

```
//same as before get the
//bookDetails
let bookDetails = concatAll(
        map(apressBooks,(book) => {
                return book.bookDetails
        })
)

//zip the results
let mergedBookDetails = zip(bookDetails,reviewDetails,(book,review) => {
  if(book.id === review.id)
  {
    let clone = Object.assign({},book)
    clone.ratings = review
    return clone
  }
})
```

Let us break down what's happening in the zip function. The result of the zip function is nothing but the same old data structure we had, precisely, mergedBookDetails:

```
[ { id: 111,
    title: 'C# 6.0',
    author: 'ANDREW TROELSEN',
    rating: [ 4.7 ],
    ratings: { id: 111, reviews: [Object] } },
```

```
  { id: 222,
    title: 'Efficient Learning Machines',
    author: 'Rahul Khanna',
    rating: [ 4.5 ],
    reviews: [],
    ratings: { id: 222, reviews: [] } },
  { id: 333,
    title: 'Pro AngularJS',
    author: 'Adam Freeman',
    rating: [ 4 ],
    reviews: [],
    ratings: { id: 333, reviews: [] } },
  { id: 444,
    title: 'Pro ASP.NET',
    author: 'Adam Freeman',
    rating: [ 4.2 ],
    ratings: { id: 444, reviews: [Object] } } ]
```

The way we have arrived at this result is very simple; while doing the `zip` operation we are taking the `bookDetails` array and `reviewDetails` array. We are checking if both the `id`s match, and if so we clone a new object out of the `book` and call it a `clone`:

```
. . .
 let clone = Object.assign({},book)
. . .
```

Now `clone` gets a copy of what's there in the `book` object. However, the important point to note is that `clone` is pointing to a separate reference. Adding/manipulating `clone` doesn't change the real `book` reference itself. In JavaScript, objects are used by reference, so changing the `book` object by default within our `zip` function will affect the contents of `bookDetails` itself, which we don't want to do.

So once we took up the `clone`, we added to it a `ratings` key with `review` object as its value:

```
clone.ratings = review
```

and finally we are returning it! Now you can apply the `reduce` function as before to solve the problem. `zip` is yet another small and simple function, but its usages are very powerful.

Summary

We have made a lot of progress in this chapter. We created a bunch of useful functions such as `map`, `filter`, `concatAll`, `reduce`, and `zip` to make it easier to work with Arrays. We term these functions *projection* functions, as these functions always return the array after applying the transformation (which is passed via a higher-order function). An important

point to keep in mind is that these are just higher-order functions, which we will be using in daily tasks. Understanding how these functions work will help us to think in more functional terms. But our functional journey is not yet over.

Having created many useful functions on Arrays in this chapter, in the next one we will be discussing *Currying* and *Partial Application* concepts. Don't worry if these terms make you fear; they are simple concepts but become very powerful when put it in action. See you in Chapter 6!

CHAPTER 6

Currying and Partial Application

■ **Note** The chapter examples and library source code are in branch `chap06`. The repo's URL is: `https://github.com/antoaravinth/functional-es6.git`

Once checkout the code, please checkout branch `chap06`:

```
...
git checkout -b chap06 origin/chap06
...
```

For running the codes, as before run:

```
...
npm run playground
...
```

In this chapter we are going to see what the term *currying* means. Once we understand what currying does and why it's useful, we will move to another concept in functional programming called Partial application. Both currying and partial application are important to understand as we will be using them during functional composition!

As seen before in the previous chapters, we are going to look at a sample problem and explain how functional techniques like currying and partial application work.

A Few Terminologies

Before explaining what currying/partial application does, we need to understand a few terminologies that we will be using in this chapter.

A. Aravinth, *Beginning Functional JavaScript*, DOI 10.1007/978-1-4842-2656-8_6

unary Function

A function is called *unary* if it takes a single function argument. For example, function `identity` is a unary function:

Listing 6-1. unary Identity Function

```
const identity = (x) => x;
```

The above function (Listing 6-1) takes only one argument `x`, so we can call the above function as a unary function.

Binary Function

A function is called *binary* if it takes two arguments. For example, in Listing 6-2, function `add` is called a binary function:

Listing 6-2. binary add Function

```
const add = (x,y) => x + y;
```

`add` function takes two arguments `x,y`; hence we call it a binary function.

And now as you can guess there are ternary functions that take three arguments and so on. But JavaScript does allow a special type of function that we call a *variadic* function, which takes a variable number of arguments.

variadic Functions

A variadic function is a function that takes a variable number of arguments. Remember that we had `arguments` in older versions of JavaScript, which we can use to capture the variable number of arguments?

Listing 6-3. variadic Function

```
function variadic(a){
        console.log(a);
        console.log(arguments)
}
```

if we call the function `variadic` like this:

```
variadic(1,2,3)
=> 1
=> [1,2,3]
```

■ **Note** As you can see in the output, `arguments` does capture all the arguments that are passed to a function.

As you can see in the above output (Listing 6-3), using `arguments` we are able to capture the additional arguments one could call on a function. Using this technique, we used to achieve the variadic functions in ES5 versions. However, starting with ES6, we have a new operator called *Spread Operator* that we can use to achieve the same result.

Listing 6-4. variadic Function Using Spread Operator

```
const variadic = (a,...variadic) => {
        console.log(a)
        console.log(variadic)
}
```

Now if we call the above function we get exactly what we would expect:

```
variadic(1,2,3)
=> 1
=> [2,3]
```

As you can see in the result, we were pointed to the first passed argument `1` and all other remaining arguments captured by our `variadic` variable that uses the `...` spread operator! ES6 style is more concise as it clearly mentions that a function does take variadic arguments for processing.

Now that we have some common terminologies in mind with respect to functions, it's time to turn our attention into the fancy term Currying!

Currying

Have you seen the term Currying n number of times in functional programming blogs and still wonder what it means? Don't worry; we are going to break the Currying definition into smaller n definitions, which will make sense to you!

We'll start with a simple question: what is Currying? A simple answer to that question would be this:

> Currying is a process of converting a function with n number of arguments into a *nested* unary function.

Don't worry if that doesn't make sense to you yet! Let's see what it means by a simple example. Imagine we have a function called `add`:

```
const add = (x,y) => x + y;
```

It's a simple function. We can call this function like `add(1,1)`, which is going to give me the result `2`. Nothing fancy here. Now here is the curried version of the `add` function:

```
const addCurried = x => y => x + y;
```

The above `addCurried` function is now a curried version of `add`. If we call `addCurried` with a single argument:

```
addCurried(4)
```

it returns a function where x value is captured via the closure concept as we have seen in the previous chapters:

```
=> fn = y => 4 + y
```

So we can call the addCurried function like the following to get the proper result:

```
addCurried(4)(4)
=> 8
```

Here we have manually converted add a function, which takes the 2 argument into an addCurried function, which has nested unary functions. Here is how to convert this process into a method called curry (Listing 6-5):

Listing 6-5. curry Function Definition

```
const curry = (binaryFn) => {
  return function (firstArg) {
    return function (secondArg) {
      return binaryFn(firstArg, secondArg);
    };
  };
};
```

■ **Note** I have written the curry function in ES5 format, as I want the reader to visualize the process of returning a *nested* unary function.

Now we can use our curry function to convert our add function to a curried version like this:

```
let autoCurriedAdd = curry(add)
autoCurriedAdd(2)(2)
=> 4
```

The output is exactly what we wanted to get! Now it's time to revise the definition of *Currying*:

> Currying is a process of converting a function with n number of arguments into a *nested* unary function.

As you can see in our curry function definition, we are converting the binary function into nested functions, each of which takes only one argument; that is, we are returning the nested unary functions! Hopefully I have clarified the term currying in your head. But the obvious questions you still have are these: Why do we need currying? What is its use?

Currying Use Cases

To begin with, we'll start simple. Imagine we have to create a function for creating tables. For example, we need to create `tableOf2`, `tableOf3`, `tableOf4` and so on.

We can achieve the same via the following in Listing 6-6:

Listing 6-6. tables Function without Currying

```
const tableOf2 = (y) => 2 * y
const tableOf3 = (y) => 3 * y
const tableOf4 = (y) => 4 * y
```

with that in place, the functions can be called this:

```
tableOf2(4)
=> 8
tableOf3(4)
=> 12
tableOf4(4)
=> 16
```

Now you see that you can generalize the tables concept into a single function like this:

```
const genericTable = (x,y) => x * y
```

and then you can use `genericTable` to get `tableOf2` like the following:

```
genericTable(2,2)
genericTable(2,3)
genericTable(2,4)
```

and the same for `tableOf3` and `tableOf4`. But if you notice the pattern, we are filling up `2` in the first argument for `tableOf2`, `3` for `tableOf3`, and so on! Perhaps you are thinking that we can solve this problem via `curry`? Let's build tables from `genericTable` using `curry`:

Listing 6-7. tables Function Using Currying

```
const tableOf2 = curry(genericTable)(2)
const tableOf3 = curry(genericTable)(3)
const tableOf4 = curry(genericTable)(4)
```

now you can do your testing with these curried version of tables:

```
console.log("Tables via currying")
console.log("2 * 2 =",tableOf2(2))
console.log("2 * 3 =",tableOf2(3))
console.log("2 * 4 =",tableOf2(4))
```

```
console.log("3 * 2 =",tableOf3(2))
console.log("3 * 3 =",tableOf3(3))
console.log("3 * 4 =",tableOf3(4))

console.log("4 * 2 =",tableOf4(2))
console.log("4 * 3 =",tableOf4(3))
console.log("4 * 4 =",tableOf4(4))
```

which is going to print the value we expect:

```
Table via currying
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
3 * 2 = 6
3 * 3 = 9
3 * 4 = 12
4 * 2 = 8
4 * 3 = 12
4 * 4 = 16
```

A logger Function - Using Currying

The previous section example helped you to understand what currying does. But let's use a bit more complicated example in this section. As developers when we write code, we do a lot of logging at several stages of the application. We could write a helper logger function that looks like the following (Listing 6-8):

Listing 6-8. Simple loggerHelper Function

```
const loggerHelper = (mode,initialMessage,errorMessage,lineNo) => {
        if(mode === "DEBUG")
                console.debug(initialMessage,errorMessage + "at line: " +
                lineNo)
        else if(mode === "ERROR")
                console.error(initialMessage,errorMessage + "at line: " +
                lineNo)
        else if(mode === "WARN")
                console.warn(initialMessage,errorMessage + "at line: " +
                lineNo)
        else
                throw "Wrong mode"
}
```

and when any developer from our team needs to print an error to the console from `Stats.js` file, he/she can use the function like the following:

```
loggerHelper("ERROR","Error At Stats.js","Invalid argument passed",23)
loggerHelper("ERROR","Error At Stats.js","undefined argument",223)
loggerHelper("ERROR","Error At Stats.js","curry function is not defined",3)
loggerHelper("ERROR","Error At Stats.js","slice is not defined",31)
```

and similarly we can use the 'loggerHelper' function for debug and warn messages. As you can tell, we are repeating the arguments mainly `mode` and `initialMessage` for all the calls. Can we do it better? Yes, we can do the above calls better via currying. Can we use our `curry` function that is defined in the previous section? Unfortunately no; the reason is the `curry` function that we have designed can handle only the binary functions, not a function like `loggerHelper` that takes 4 arguments.

Let us solve this problem and implement the fully functional `curry` function, which handles any function with n number of arguments.

Revisit Curry

We all know that we can `curry` (Listing 6-5) only a function. How about many functions? It's simple but important to have it in our implementation of `curry`. Let's add the rule first:

Listing 6-9. Revisting curry Function Definition

```
let curry =(fn) => {
    if(typeof fn!=='function'){
        throw Error('No function provided');
    }
};
```

With that check in place, if others call our `curry` function with an integer like `2`, etc., they get back the error! That's perfect! The next requirement to our curried function is that if anyone provided all arguments to a curried function, we need to execute the real function by passing the arguments. Let's add that (Listing 6-10):

Listing 6-10. curry Function Handling Arguments

```
let curry =(fn) => {
    if(typeof fn!=='function'){
        throw Error('No function provided');
    }

    return function curriedFn(...args){
      return fn.apply(null, args);
    };
};
```

Now if we have function called `multiply`:

```
const multiply = (x,y,z) => x * y * z;
```

we can use our new `curry` function like the following:

```
curry(multiply)(1,2,3)
=> 6
curry(multiply)(1,2,0)
=> 0
```

So let's look at how it really works; we have added the logic in our `curry` function like this:

```
return function curriedFn(...args){
        return fn.apply(null, args);
};
```

The returned function is a `variadic` function, which returns the function result by calling the function via `apply` along by passing the args:

```
. . .
fn.apply(null, args);
. . .
```

With our `curry(multiply)(1,2,3)` example, `args` will be pointing to `[1,2,3]` and since we are calling `apply` on `fn`, it's equivalent to:

```
multiply(1,2,3)
```

which is exactly what we wanted! Thus we get back the expected result from the function.

Now let us get back to the problem of converting the n argument function into a nested unary function (that's the definition of `curry` itself)!

Listing 6-11. curry Function Converting n arg Function to unary Function

```
let curry =(fn) => {
    if(typeof fn!=='function'){
        throw Error('No function provided');
    }

    return function curriedFn(...args){

      if(args.length < fn.length){
        return function(){
```

```
          return curriedFn.apply(null, args.concat( [].slice.call(arguments) ));
        };
      }

      return fn.apply(null, args);
    };
};
```

We have added the part:

```
if(args.length < fn.length){
        return function(){
          return curriedFn.apply(null, args.concat( [].slice.call(arguments)
          ));
        };
}
```

Let's understand what's happening in this piece of code, one by one:

```
args.length < fn.length
```

This particular line checks if the argument that is passed via `...args` length and the function argument list `length` is less or not. If so we go into the `if` block, or else we fall back to call the full function as before.

Once we enter the if block, we used the `apply` function to call the `curriedFn` recursively like this:

```
curriedFn.apply(null, args.concat( [].slice.call(arguments) ));
```

The snippet:

```
args.concat( [].slice.call(arguments) )
```

is important. Using the `concat` function, we are concating the arguments that are passed one at a time and calling the `curriedFn` recursively. Since we are combining all the passed arguments and calling it recursively, we will meet a point in which the line:

```
 if (args.length < fn.length)
```

condition fails. As the argument list length ('args') and function argument length (fn.length) will be equal, thus skipping the `if` block and calling up:

```
return fn.apply(null, args);
```

which is going to yield the functions' full result!

With that understanding in place, we can use our `curry` function to invoke the `multiply` function:

```
curry(multiply)(3)(2)(1)
=> 6
```

Perfect! We have created our own `curry` function.

■ **Note** You can call the above code snippet like the following, too:

```
let curriedMul3 = curry(multiply)(3)
let curriedMul2 = curriedMul3(2)
let curriedMul1 = curriedMul2(1)
```

where `curriedMul1` will be equal to `6`. But we do it like `curry(multiply)(3)(2)(1)` as it is much more readable!

An important point to note is that our `curry` function is now converting a function of n arguments into a function that can be called as a unary function as the example shows.

Back to logger Function

Now let's solve our logger function using the defined `curry` function. Bringing up the function here for easy reference (Listing 6-8):

```
const loggerHelper = (mode,initialMessage,errorMessage,lineNo) => {
        if(mode === "DEBUG")
                console.debug(initialMessage,errorMessage + "at line: " +
                lineNo)
        else if(mode === "ERROR")
                console.error(initialMessage,errorMessage + "at line: " +
                lineNo)
        else if(mode === "WARN")
                console.warn(initialMessage,errorMessage + "at line: " +
                lineNo)
        else
                throw "Wrong mode"
}
```

The developer used to call the function:

```
loggerHelper("ERROR","Error At Stats.js","Invalid argument passed",23)
```

Now let's solve the repeating first two arguments problem via curry:

```
let errorLogger = curry(loggerHelper)("ERROR")("Error At Stats.js");
let debugLogger = curry(loggerHelper)("DEBUG")("Debug At Stats.js");
let warnLogger = curry(loggerHelper)("WARN")("Warn At Stats.js");
```

Now we can easily refer to the above curried functions and use them under the respective context:

```
//for error
errorLogger("Error message",21)
=> Error At Stats.js Error messageat line: 21

//for debug
debugLogger("Debug message",233)
=> Debug At Stats.js Debug messageat line: 233

//for warn
warnLogger("Warn message",34)
=> Warn At Stats.js Warn messageat line: 34
```

That's brilliant! We have seen how curry function helps in the real world to remove a lot of boilerplates in function calls! Don't forget to say thanks to the closures concept, which is backing up the curry function.

Currying in Action

In the previous section we created our own 'curry' function. We have also seen a simple example of using the 'curry' function.

In this section we are going to see small but compact examples where the currying technique will be used. The examples shown in this section will make you better understand how to use currying in your day-to-day activities.

Finding number in Array Contents

Imagine we want to find the array content that has a number. We can solve the problem via the following code snippet:

```
let match = curry(function(expr, str) {
  return str.match(expr);
});
```

The returned match function is a curried function. We can give the first argument expr a regular expression /[0-9]+/ that will indicate whether the content has a number in it.

```
let hasNumber = match(/[0-9]+/)
```

Now we will create a curried `filter` function:

```
let filter = curry(function(f, ary) {
  return ary.filter(f);
});
```

Now with `hasNumber` and `filter` in place, we can create a new function called `findNumbersInArray`:

```
let findNumbersInArray = filter(hasNumber)
```

Now you can test it:

```
findNumbersInArray(["js","number1"])
=> ["number1"]
```

That's great!

squaring an Array

We know how to square contents of an array. We have also seen the same problem in previous chapters. We use `map` function and pass on square function to achieve the solution to our problem. But here we can use the `curry` function to solve the same problem in another way:

```
let map = curry(function(f, ary) {
  return ary.map(f);
});

let squareAll = map((x) => x * x)

squareAll([1,2,3])
=> [1,4,9]
```

As you can see in the above example, we have created a new function `squareAll` that we can now use elsewhere in our code base! Similarly you can also do this for `findEvenOfArray`, `findPrimeOfArray`, etc.

Data Flow

In both sections of using currying, we have designed the curried functions such that it always takes the `array` at the end. This is intentional way of creating a curried function! As talked about in previous chapters, we as programmers often work on data structures like `array`, so making the `array` as the last argument allows us to create lot of reusable functions like `squareAll` and `findNumbersInArray` that we can use throughout the code base!

■ **Note** In our source code companion, we have termed the curry function as `curryN`. It's just to keep the old `curry` remains as is, which was supposed to do currying on binary functions!

Partial Application

In this section we are going to see yet another function called `partial` that allows developers to apply the function arguments partially!

Imagine we want to do a set of operations after every 10 ms. Using the `setTimeout` function, we can do this:

```
setTimeout(() => console.log("Do X task"),10);
setTimeout(() => console.log("Do Y task"),10);
```

As you can see, we are passing on `10` for every one of our `setTimeout` function calls. Can we hide that from the code? Can we use `curry` function to solve this problem? The answer is no. The reason is that the `curry` function applies the argument from the leftmost to rightmost lists! Since we want to pass on the functions as needed and keep `10` as a constant (which is most of the argument list), we can't use `curry` as such. One workaround is that we can wrap our `setTimeout` function so that the function argument becomes the rightmost one:

```
const setTimeoutWrapper = (time,fn) => {
  setTimeout(fn,time);
}
```

Then we can use our `curry` function to wrap our setTimeout to a `10`-second delay:

```
const delayTenMs = curry(setTimeoutWrapper)(10)
delayTenMs(() => console.log("Do X task"))
delayTenMs(() => console.log("Do Y task"))
```

which will work as we needed it to. But the problem is we have to create wrappers like `setTimeoutWrapper`, which will be an overhead! And that's where we can use *partial application* techniques!

Implementing `partial` Function

In order to fully understand how the partial application technique is working, we will be creating our own 'partial' function in this section. Once the implementation is done, we will learn how to use our 'partial' function with a simple example.

The implementation of `partial` function looks like the following (Listing 6-12):

Listing 6-12. partial Function Definition

```
const partial = function (fn,...partialArgs){
  let args = partialArgs;
  return function(...fullArguments) {
    let arg = 0;
    for (let i = 0; i < args.length && arg < fullArguments.length; i++) {
      if (args[i] === undefined) {
        args[i] = fullArguments[arg++];
        }
      }
      return fn.apply(null, args);
  };
};
```

Let's quickly use the `partial` function with our current problem:

```
let delayTenMs = partial(setTimeout,undefined,10);
delayTenMs(() => console.log("Do Y task"))
```

which will print to the console as you expect. Now let's walk through the implementation details of `partial` function. Using closures, we are capturing the arguments that are passed onto the function for the first time:

```
partial(setTimeout,undefined,10)

//will lead to
let args = partialArgs
=> args = [undefined,10]
```

And we return a function that will remember the `args` value (yeah, again we are using closures!). The returned function is super easy. It takes an argument called `fullArguments`. So we call functions like `delayTenMs` by passing the argument:

```
delayTenMs(() => console.log("Do Y task"))

//fullArguments points to
//[() => console.log("Do Y task")]

//args using closures will have
//args = [undefined,10]
```

Now in the `for` loop we iterate and create the necessary arguments array for our function:

```
if (args[i] === undefined) {
      args[i] = fullArguments[arg++];
  }
}
```

Now let's start with value `i` as `0`:

```
//args = [undefined,10]
//fullArguments = [() => console.log("Do Y task")]
args[0] => undefined === undefined //true

//inside if loop
args[0] = fullArguments[0]
=> args[0] = () => console.log("Do Y task")

//thus args will become
=> [() => console.log("Do Y task"),10]
```

As you can see in the above code snippet examples, our `args` point to the array as we would expect for `setTimeout` function calls. Once we have the necessary arguments in `args`, we call the function via `fn.apply(null, args)`!

Remember that we can apply `partial` for any function that has n arguments. To make the point concrete, let's look at an example. In JavaScript we use the following function call to do JSON pretty print:

```
let obj = {foo: "bar", bar: "foo"}
JSON.stringify(obj, null, 2);
```

As you can see, the last two arguments for the function call `stringify` are always going to be the same `null,2`. We can use `partial` to remove the boilerplate:

```
let prettyPrintJson = partial(JSON.stringify,undefined,null,2)
```

and then you can use `prettyPrintJson` to print the json:

```
prettyPrintJson({foo: "bar", bar: "foo"})
```

which will give you the output:

```
"{
  "foo": "bar",
  "bar": "foo"
}"
```

■ **Note** There is a slight bug in our implementation of `partial` function. What if you call `prettyPrintJson` again with a different argument? Does it work?

It always gives the result for the first invoked argument, but why? Can you see where we are making the mistake?

HINT: Remember, we are modifying the `partialArgs` by replacing the `undefined` values with our argument, and `Arrays` are used for reference!

Currying vs. Partial Application

We have seen both of these techniques. So the question is when to use which one? The answer depends on how your API is been defined. If your API is defined as such `map`,`filter` then we can easily use the `curry` function to solve our problem. But as discussed in the previous section, life is not always easy. There could be functions that are not designed for `curry` such as `setTimeout` in our examples. In those cases, the best fit option would be to use `partial` functions! After all, we use `curry` or `partial` to make function arguments/ function setup easy and more powerful!

And also it's important to note that currying will return nested unary functions; we have implemented `curry` so that it takes n arguments just for our convenience. And also it's a proven fact that developers need either `curry` or `partial` but not both.

And that brings us to the end of our discussion!

Summary

Currying and partial application are always a tool in functional programming. We started the chapter by explaining the definition of currying, which is nothing but converting a function of n arguments into nested unary functions. We saw the examples of currying and where it can be very useful. But there are cases where you want to fill the first 2 arguments of a function and the last argument, leaving the middle argument to be unknown for a certain time! And that's where partial application comes into the picture. In order to fully understand both these concepts, we have implemented our own `curry` and `partial` functions! We have made a lot of progress, we're not done yet!

Functional programming is all about composing functions – composing several small functions to build a new function! Composing and Pipelines will be the topic of the next chapter. Stay tuned!

CHAPTER 7

Composition and Pipelines

■ **Note** The chapter examples and library source code are in branch `chap07`. The repo's URL is: `https://github.com/antoaravinth/functional-es6.git`

Once checkout the code, please checkout branch `chap07`:

```
...
git checkout -b chap07 origin/chap07
...
```

For running the codes, as before run:

```
...
npm run playground
...
```

In the previous chapter we saw two important techniques for functional programming: Currying and Partial application. We discussed how these two techniques work! We also discussed that as a JavaScript programmer we will be choosing either Currying or Partial application in our code base. In this chapter we are going to see what *Functional Composition* means and its practical use cases.

Functional Composition is simply referred as *composition* in the functional programming world. We are going to see a bit of theory on the idea of composition and quite a few examples of it. Then we will be writing our own `compose` function. Again, it's fun! Understanding how `compose` function works under the hood is really a fun task.

Composition in General Terms

Before we see what a functional composition is all about, let's step back and understand the idea behind composition. In this section we are going to drive a philosophy that will help us to get the benefit out of composition in general.

A. Aravinth, *Beginning Functional JavaScript*, DOI 10.1007/978-1-4842-2656-8_7

Unix Philosophy

Unix philosophy is a set of ideas that originated by Ken Thompson. One part of the Unix philosophy is this:

> Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new "features."

This is exactly what we are doing as part of creating our functions. Functions as we have seen until now in the books are supposed to take an argument and return a data. Yes, functional programming does follow Unix philoshopy. Kudos!

The second part of the philosophy is this:

> Expect the output of every program to become the input to another, as yet unknown, program.

Hmmm, that's an interesting quote. What does it mean by *Expect the output of every program to become the input to another*? To make the point clear, let's look at a few commands on Unix platform that were built by following these philosophies.

For example, `cat` is a command (or you can think of it as a function) that is used to display the contents of a text file to a console. Here the `cat` command takes an argument (as similar to a function), that is, the file location, etc., and returns the output (again as similar to a function) to the console. So we can do the following:

```
cat test.txt
```

Which will print to the console?

```
Hello world
```

■ **Note** Here the content of the `test.txt` will be `Hello world.`

That's so simple. Another command called `grep` allows us to search for content in a given text. An important point to note is that the `grep` function takes an input and gives the output (again very similar to a function).

We can do the following with the `grep` command:

```
grep 'world' test.txt
```

which will return us the matching content, in this case:

```
Hello world
```

We have seen two very simple functions: `grep` and `cat` that are is built by following the Unix philosophy. Now we can take some time to understand this quote:

> Expect the output of every program to become the input to another, as yet unknown, program.

Imagine you to want to send the data from the `cat` command as an input to the `grep` command in order to do a search. We know that the `cat` command will return the data; and also we know that the `grep` command takes the data for processing the search operation. Thus, using the Unix | (pipe symbol), we can achieve our task:

```
cat test.txt | grep 'world'
```

which will return the data as expected:

```
Hello world
```

■ **Note** The symbol | is called a pipe symbol. This allows us to combine several functions to create a new function that will help us to solve our problem! Basically | sends the output of a function on the left-hand side as an input to a function on the right-hand side! This process, technically, is called s pipeline!

The above example might be trivial, but it conveys the idea behind the quote:

> Expect the output of every program to become the input to another, as yet unknown, program.

As our example shows, the `grep` command or a function receives the output of a `cat` command or a function. What we have done here is that we have created a new function altogether without any effort! Of course the | pipe acts as a *bridge* to connect the given two commands.

Let's change our problem statement a bit; what if we want to count the number of occurrences of the word *`world`* in a given text file? How we can achieve it?

This is how we are going to solve it:

```
cat test.txt | grep 'world' | wc
```

■ **Note** The command `wc` is used to count the words in a given text. This command is available on all of the Unix and Linux platforms.

---

which is going to return the data as we expect! As the above examples show, we are creating a new function as per our need on the fly from our *base* functions! In other words, we are *composing* a new function from our base function(s). Note that the *base* function needs to obey this rule:

> Each base function needs to take an argument and return data!

We would be able to compose a new function with the help of |. As this chapter shows, we will be building our own `compose` function in JavaScript, which does the same job of | in the Unix/Linux world.

Now we have the idea of composing functions from base functions! The real advantage of composing functions is that we can combine our base function to solve the problem at hand, without re-creating a new function!

Functional Composition

In this section we are going to discuss a use case where functional composition will be useful in the JavaScript world. Stay with me – you're going to love the idea of the `compose` function.

Revisiting map,filter

In Chapter 5, "Being Functional on Arrays, under the section "Chaining Operations," we saw how to chain the data from a `map` and `filter` to solve the problem in hand. Let's quickly revisit the problem and the solution we have taken.

We had an array of objects, whose structure looks like the following:

Listing 7-1. Apress book object structure

```
let apressBooks = [

    {
        "id": 111,
        "title": "C# 6.0",
        "author": "ANDREW TROELSEN",
        "rating": [4.7],
        "reviews": [{good : 4 , excellent : 12}]
    },
    {
        "id": 222,
        "title": "Efficient Learning Machines",
        "author": "Rahul Khanna",
        "rating": [4.5],
        "reviews": []
    },
    {
        "id": 333,
        "title": "Pro AngularJS",
        "author": "Adam Freeman",
        "rating": [4.0],
        "reviews": []
    },
    {
        "id": 444,
        "title": "Pro ASP.NET",
```

```
        "author": "Adam Freeman",
        "rating": [4.2],
        "reviews": [{good : 14 , excellent : 12}]
    }
];
```

The problem was to get the title and author object out of our `apressBooks` for which the review is greater than `4.5`. Our solution to that current problem looks like the following:

Listing 7-2. Getting author details using map

```
map(filter(apressBooks, (book) => book.rating[0] > 4.5),(book) => {
    return {title: book.title,author:book.author}
})
```

for which we have got the result as the following:

```
[
        {
                title: 'C# 6.0',
                author: 'ANDREW TROELSEN'
        }
]
```

The code to achieve the solution tells an important point. The data out of our `filter` function is passed into the `map` function as its input argument! Yeah, you have guessed it correctly. Does it sound like the exact same problem we solved in the previous section using | in the Unix world? Can we do the same thing in the JavaScript world? Can we create a function that will combine two functions by sending the output of one function as an input to another function?

Yes, we can! Meet `compose` function.

compose Function

In this section, let's create our first `compose` function. Creating a new `compose` function is easy and straightforward. `compose` function needs to take the output of one function and give it as input to another function. Let's put them up in in a function:

Listing 7-3. compose function definition

```
const compose = (a, b) =>
  (c) => a(b(c))
```

The `compose` function is simple and does what we need it to do. It takes two functions, `a` and `b`, and returns a function that takes the argument `c`. When we call the return function by supplying the value of `c`, it will call the function `b` with input of `c` and the output of the function `b` goes into as input of function `a`. And that's exactly what a `compose` function definition is!

Great, now let's quickly test our `compose` function with a simple example before we dive into our running example from the previous section.

■ **Note** `compose` function executes `b` first and pass the return value of `b` as an argument to the function `a`. The direction of function calling in `compose` is right to left (i.e `b` executes first, followed by `a`).

Playing with `compose` function

With our `compose` function in place, let's go and build some toy examples.

Imagine we want to round a given number. The given number will be a float, so we have to convert that number to a float and then call `Math.round`.

Without compose, we can do the following:

```
let data = parseFloat("3.56")
let number = Math.round(data)
```

the output will be 4 as we would expect. As you can see in the above example, the `data` (which is the output of `parseFloat` function) is passed as input to `Math.round` to get a solution; this is the right problem candidate which our `compose` function will solve.

Let's solve this via our `compose` function:

```
let number = compose(Math.round,parseFloat)
```

The above statement will return a new function that is stored as a `number` and looks like this:

```
number = (c) => Math.round(parseFloat(c))
```

Now if we pass the input `c` to our `number` function, we will get what we expect:

```
number("3.56")
=> 4
```

Wow, what we have done right above is functional composition! Yes, we have composed two functions in order to build a new function on the fly! An important point to note over here is that, the functions `Math.round` or `parseFloat` aren't executed/run until we call our `number` function.

Now imagine we have two functions namely:

```
let splitIntoSpaces = (str) => str.split(" ");
let count = (array) => array.length;
```

Now if you want to build a new function in order to count number of words in a string, we can easily do this:

```
const countWords = compose(count,splitIntoSpaces);
```

Now we can call that:

```
countWords("hello your reading about composition")
=> 5
```

The newly created function `countWords` using `compose` is an elegant and easy way to see exactly what it does exactly!

`curry` and `partial` to the Rescue

We know that we can `compose` two functions, only if this function takes one input argument! But that's not the case always, as there can be functions that have multiple arguments! How we are going to compose those functions? Is there something we can do about it?

Yes, we can do it using either `curry` or `partial` function that we have defined in the previous chapter! As you can recall from the section *"Revisiting map,filter."* From this chapter, we have the following code to solve our problem in hand (Listing 7-2):

```
map(filter(apressBooks, (book) => book.rating[0] > 4.5),(book) => {
    return {title: book.title,author:book.author}
})
```

Now can we use the `compose` function to compose both `map` and `filter` with specifics to our example? Remember that both `map` and `filter` functions do take two arguments: the first argument is the array, and the second argument being the function to operate on that array. So we can't `compose` these two functions directly.

However we can take help from `partial` functions. Remember that the above code snippet does work on the `apressBooks` object. Pulling it out here again for easy reference:

```
let apressBooks = [
    {
        "id": 111,
        "title": "C# 6.0",
        "author": "ANDREW TROELSEN",
        "rating": [4.7],
        "reviews": [{good : 4 , excellent : 12}]
    },
    {
        "id": 222,
        "title": "Efficient Learning Machines",
        "author": "Rahul Khanna",
        "rating": [4.5],
        "reviews": []
    },
    {
        "id": 333,
        "title": "Pro AngularJS",
        "author": "Adam Freeman",
```

```
        "rating": [4.0],
        "reviews": []
    },
    {
        "id": 444,
        "title": "Pro ASP.NET",
        "author": "Adam Freeman",
        "rating": [4.2],
        "reviews": [{good : 14 , excellent : 12}]
    }
];
```

Now let's say we have many small functions in our code base for filtering the books based out of different ratings like the following:

```
let filterOutStandingBooks = (book) => book.rating[0] === 5;
let filterGoodBooks = (book) => book.rating[0] > 4.5;
let filterBadBooks = (book) => book.rating[0] < 3.5;
```

and we do have many projection functions like:

```
let projectTitleAndAuthor = (book) => { return {title: book.
title,author:book.author} }
let projectAuthor = (book) => { return {author:book.author}  }
let projectTitle = (book) => { return {title: book.title} }
```

■ **Note** You might be wondering why we have small functions even for simple things. Remember that composition is all about small functions being composed into a larger function. Simple functions are easy to read, test, and maintain; and using `compose` we can build anything out of it, as we will see in this section.

Now to solve our problem – *To get Books titles and authors for 4.5 above rating,* we can use `compose` and `partial` as in the following:

```
let queryGoodBooks = partial(filter,undefined,filterGoodBooks);
let mapTitleAndAuthor = partial(map,undefined,projectTitleAndAuthor)

let titleAndAuthorForGoodBooks = compose(mapTitleAndAuthor,queryGoodBooks)
```

Let's take some time to understand the position of `partial` function in the current problem domain. As mentioned, the `compose` function can only compose a function that takes one argument! However both `filter` and `map` take two arguments, so we can't compose them directly.

That's the reason we have used `partial` function to partially apply the second argument for both `map` and `filter` as you can see here:

```
partial(filter,undefined,filterGoodBooks);
partial(map,undefined,projectTitleAndAuthor)
```

Here we have passed `filterGoodBooks` function to query the books that have ratings over `4.5` and passed `projectTitleAndAuthor` function to take the `title` and `author` property from the `apressBooks` object! Now the returned partial application will expect only one argument, which is nothing but the array itself! Now with these two partial functions in place, we can compose them via `compose` as we already have done:

Listing 7-4. Using compose function

```
let titleAndAuthorForGoodBooks = compose(mapTitleAndAuthor,queryGoodBooks)
```

Now the function `titleAndAuthorForGoodBooks` expects one argument in our case that is `apressBooks`; let's pass the object array to it:

```
titleAndAuthorForGoodBooks(apressBooks)
=> [
        {
                title: 'C# 6.0',
                author: 'ANDREW TROELSEN'
        }
]
```

Wow, we got back exactly what we wanted without `compose`. But the latest composed version `titleAndAuthorForGoodBooks` is much more readable and elegant in my opinion. You can sense the importance of creating small units of function that can be again rebuilt using `compose` as per our needs!

In the same example, what if we want to get only the `titles` of the books with those above a `4.5` rating? Ah ha, it's so simple:

```
let mapTitle = partial(map,undefined,projectTitle)
let titleForGoodBooks = compose(mapTitle,queryGoodBooks)

//call it
titleForGoodBooks(apressBooks)
=> [
        {
                title: 'C# 6.0'
        }
]
```

How about getting only `author` names for books with ratings that equal `5`? That should be easy, right? I leave you to solve this using the above defined functions and the `compose` function!

■ **Note** In this section, we have used `partial` to fill the arguments of a function. However you can use `curry` to do the same thing. It's just a matter of choice. But can you come up with a solution for using `curry` in our example above? (Hint: Reverse the order of argument for `map`, `filter`).

`compose` many function

Currently our version of `compose` function does compose only two given functions. How about composing three, four, or n number of functions? Sadly, our current implementation doesn't handle this. Let's rewrite our `compose` function so that it can compose multiple functions on the fly.

Remember that we need to send the output of each function as an input to another function (by remembering the last executed function output recursively). We can use `reduce` function, which we have used in previous chapters to reduce the n of function calls one at a time. The rewritten `compose` function now looks like the following:

Listing 7-5. compose many function

```
const compose = (...fns) =>
  (value) =>
    reduce(fns.reverse(),(acc, fn) => fn(acc), value);
```

■ **Note** The above function is called `composeN` in source code repo.

The important line of the function is:

```
reduce(fns.reverse(),(acc, fn) => fn(acc), value);
```

■ **Note** If you recall from the previous chapter, we have used `reduce` function to reduce the array into a single value (along with an accumulator value; i.e., the third parameter of `reduce`), for example, to find the sum of the given array, using `reduce`:

```
reduce([1,2,3],(acc,it) => it + acc,0)
=> 6
```

Here the array `[1,2,3]` is reduced into `[6]`; the accumulator value here is `0`.

Here we are first reversing the function array via `fns.reverse()` and passing the function as `(acc, fn) => fn(acc)`, which is going to call each function one after the other by passing the `acc` value as its argument. And notably the initial accumulator value is nothing but a `value` variable, which will be the first input to our function!

With the new `compose` function in place, let's go and test it with our old example. In the previous section we composed a function to count words given in a string:

```
let splitIntoSpaces = (str) => str.split(" ");
let count = (array) => array.length;
const countWords = compose(count,splitIntoSpaces);

//count the words
countWords("hello your reading about composition")
=> 5
```

Now imagine we want to find out whether the word count in the given string is odd or even. We already have a function for it:

```
let oddOrEven = (ip) => ip % 2 == 0 ? "even" : "odd"
```

Now with our `compose` function in place, we can compose these three functions to get what we really want:

```
const oddOrEvenWords = compose(oddOrEven,count,splitIntoSpaces);
oddOrEvenWords("hello your reading about composition")
=> ["odd"]
```

We got back the expected result! Go and play around with our new `compose` function!

Now we have a solid understanding of how to use compose function to get what we need. In the next section, we are going to see the same concept of `compose` in a different way called *Pipelines.*

Pipelines / Sequence

In the previous section, we saw how the `compose` function data flow works. Yes, the data flow of compose is from left to right, as the functions on the left mostly get executed first, passing on the data to the next function, and so on . . . and the right-most function gets executed at last!

Certain people prefer the other way – where the right-most function gets executed first and the left-most function on the left most gets executed last. As you can remember, the data flow on Unix commands when we do | is from right to left. So in this section, we are going to implement a new function called `pipe` that does exactly the same thing as the `compose` function, but just swaps the data flow!

■ **Note** This process of flowing the data from right to left is called pipelines or even sequences! You can call them either PipeLine or Sequences as you prefer.

Implementing pipe

pipe function is just replica of our compose function, the only change is that the data flow:

Listing 7-6. pipe function definition

```
const pipe = (...fns) =>
  (value) =>
    reduce(fns,(acc, fn) => fn(acc), value);
```

That's it! Note that there is no more call on fns reverse functions as in compose, which means we are going to execute the function order as it is (from right to left).

Let's quickly check our implementation of pipe function by rerunning the same example in previous section:

```
const oddOrEvenWords = pipe(splitIntoSpaces,count,oddOrEven);
oddOrEvenWords("hello your reading about composition");
=> ["odd"]
```

The result is going to be the exact same; however, notice that we have changed the order of functions when we do piping! First we call splitIntoSpaces and then count and finally oddOrEven!

Some people (who have the knowledge of shell scripting) prefer pipes over compose. It's just a personal preference and nothing to do with the underlying implementation. The takeaway is that both pipe and compose do the same thing, but in different data flow! You can use either pipe or compose in your code base, but not both, as it can lead to confusion between your team members! Stick to one style of composing. :)

Odds on Composition

In this section, we discuss two topics. The first discussion is on one of the most important properties of compose - *Composition is associative*. The second discussion will be on how we debug when we compose many functions!

Let's tackle one after the other.

Composition is associative

The functional composition is always associative:

```
compose(f, compose(g, h)) == compose(compose(f, g), h);
```

let's quickly check our previous section example:

```
//compose(compose(f, g), h)

let oddOrEvenWords = compose(compose(oddOrEven,count),splitIntoSpaces);
let oddOrEvenWords("hello your reading about composition")
```

```
=> ['odd']

//compose(f, compose(g, h))

let oddOrEvenWords = compose(oddOrEven,compose(count,splitIntoSpaces));
let oddOrEvenWords("hello your reading about composition")
=> ['odd']
```

As you can see in the above examples, the result is going to be the same for both the cases! Thus it proves the functional composition is associative. You might be thinking, what is the benefit of `compose` being associative?

The real benefit is that it allows us to group functions into their own `compose`! That is:

```
let countWords = compose(count,splitIntoSpaces)
let oddOrEvenWords = compose(oddOrEven,countWords)

or
let countOddOrEven = compose(oddOrEven,count)
let oddOrEvenWords = compose(countOddOrEven,splitIntoSpaces)

or
...
```

The above code is possible just because the composition possesses the associative property! Earlier in the chapter we discussed that creating small functions is the key to compose! Since `compose` is associative we can create small functions by composition, without any worry, as the result is going to be the same!

Debugging Using `tap` Function

We have been using `compose` function quite a lot in this chapter. `compose` function can compose any number of functions. The data is going to flow from left to right in a chain until the full function list is evaluated! In this section, I'm going to teach you a trick that allows you to debug the failures on `compose`!

Let's create a simple function called `identity`. The aim of this function is to take the argument and return the same argument; hence the name *identity*:

```
const identity = (it) => {
        console.log(it);
        return it
}
```

Here we have added a simple `console.log` to print the value this function receives and also return it as it is! Now imagine we have the following call:

```
compose(oddOrEven,count,splitIntoSpaces)("Test string");
```

When you execute the above code, what if `count` function throws an error? How will you know what value does `count` function receive as its argument? And that's where our little `identity` function comes into picture. We can add

`identity` in the flow where we see the error like:

```
compose(oddOrEven,count,identity,splitIntoSpaces)("Test string");
```

which is going to print the input argument that the `count` function is going to receive. This little function can be very helpful in debugging what data a function does receive.

Summary

We started this chapter by taking Unix philosophy as an example. We have seen how, by following the Unix philosophy, Unix commands like `cat`,`grep`,`wc` could be able to compose as needed! Then we went ahead and created our own version of the `compose` function to achieve the same in JavaScript world! The little `compose` function brings heavy usage to developers as we can compose complex functions as needed from our well-defined small functions. We also saw an example of how *currying* helps in functional composition, by a `partial` function.

We also discussed another function called `pipe`, which does exactly the same thing but inverts the data flow when compared to the `compose` function. At the end of the chapter we discussed an important property of `compose` - composition is associative! We also presented a small function called `identity` that we can use as our debugging tool while facing problems with the `compose` function!

In the next chapter, we are going to see Functors. Functors are very simple, but at the same time very powerful. We will be seeing the use cases and lot more about Functors in the next chapter! Stay tuned!

CHAPTER 8

Fun with Functors

■ **Note** The chapter examples and library source code are in branch `chap08`. The repo's URL is: `https://github.com/antoaravinth/functional-es6.git`

Once checkout the code, please checkout branch `chap08`:

```
...
git checkout -b chap08 origin/chap08
...
```

For running the codes, as before run:

```
...
npm run playground
...
```

In our previous chapter, we dealt with many functional programming techniques. In this chapter we are going to see yet important concept in programming called *error handling*. Error handling is a common programming technique for handling errors in your application. But the functional programming way of error handling will be different, and that's exactly what we are going to see in the current chapter.

We will be looking at a new concept called *Functor*. This new friend will be introduced, and it is going to help us to handle errors in a purely functional way. Once we grasp the idea of *Functor*, we are going to implement two real-world functors: namely, `MayBe` and `Either`. So let's get started.

What Is a Functor?

In this section we are going to see what a *Functor* really is. Here is its definition:

> Functor is a plain object (or type class in other languages) that implements the function `map` that, while running over each value in the object to produce a new object.

A. Aravinth, *Beginning Functional JavaScript*, DOI 10.1007/978-1-4842-2656-8_8

Hmmm, that's the definition of *Functor*. But it is not as easy to understand the definition at first sight. We are going to break it down step by step section so that we clearly understand and see in action (via writing code) what a *Functor* is.

Functor Is a Container

Simply put, functor is a container that holds the value in it. We have seen this in the definition stating that *Functor is a plain object*. So let's go ahead and create a simple container that can hold any value we pass onto it, and we call it a `Container` :

Listing 8-1. Container Definition

```
const Container = function(val) {
        this.value = val;
}
```

You might be wondering why we didn't write the `Container` function using our arrow syntax:

```
const Container = (val) => {
        this.value = val;
}
```

The above code will be fine, but the moment we try to apply `new` keyword on our `Container`, we will be getting the back an error like this:

```
Container is not a constructor(...)(anonymous function)
```

Why is that? Well, technically, in order to create a new Object, the function should have the internal method `[[Construct]]` and the property prototype. Sadly, the Arrow function doesn't have both! So here we are falling back to our old school friend `function`, which has the internal method `[[Construct]]`, and it also has access to the prototype property.

Now with `Container` in place, we can go ahead and create a new object out of it:

Listing 8-2. Playing With Container

```
let testValue = new Container(3)
=> Container(value:3)

let testObj = new Container({a:1})
=> Container(value:{a:1})

let testArray = new Container([1,2])
=> Container(value:[1,2])
```

Nothing fancy here; Container is just holding the value inside it. We can pass any data type in JavaScript to it and Container will hold it. Before we move on, we can create a util method called of in the Container prototype, which will save us in writing the new keyword to create a new Container. The code looks like the following:

Listing 8-3. of method definition

```
Container.of = function(value) {
  return new Container(value);
}
```

With this of method in place, we can rewrite the above code (Listing 8-2) snippets like this:

Listing 8-4. Creating Container with of

```
testValue = Container.of(3)
=> Container(value:3)

testObj = Container.of({a:1})
=> Container(value:{a:1})

testArray = new Container([1,2])
=> Container(value:[1,2])
```

It is worth noting that Container can contain nested Containers too:

```
Container.of(Container.of(3));
```

is going to print:

```
Container {
        value: Container {
                value: 3
        }
}
```

Now that we have defined that the *Functor* is nothing but a Container that can hold the value, let's revisit the definition of Functor.

Functor is a plain object (or type class in other languages) that implements the function map while running over each value in the object to produce a new object.

It looks like *Functor* needs to implement a method called map. Let's implement that method in the next section.

Functor Implements Method Called map

Before we implement the map function, let's stop here and think why we need map function in the first place. Remember that the created Container just holds the value that we pass onto it. But holding the value hardly has any use cases. And that is where map

function comes into place. map function allows us to call any function on the value that is being currently held by the Container.

map function takes the value out of the Container and applies the passed function on that value and again put back the result in the Container. Let's visualize using the following image (Figure 8-1):

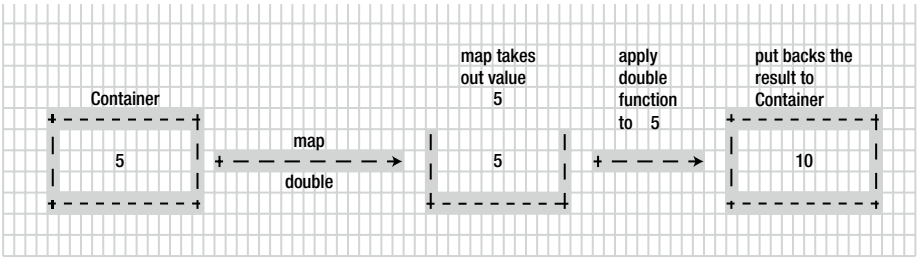

Figure 8-1. *Mechanism of Container And map Function*

The above image tells the way map function is going to work with our Container object. It takes the value in the Container; in this case the value is 5, and pass on that value to the passed function double (this function just doubles the given number), and the result is being put back again to Container. With that understanding in place, we can implement the map function:

Listing 8-5. map function definition

```
Container.prototype.map = function(fn){
  return Container.of(fn(this.value));
}
```

As shown above, the above map function just simply does what we have discussed in our image! It's simple and elegant! Now to make the point concrete, let's put our image piece into code action:

```
let double = (x) => x + x;
Container.of(3).map(double)
=> Container { value: 6 }
```

Note that the map returns the result of the passed function again in the container, which allows us to chain the operation:

```
Container.of(3).map(double)
                            .map(double)
                            .map(double)

=> Container {value: 24}
```

Now implementing `Container` with our `map` function, we can make complete sense of Functor definition:

> Functor is a plain object (or type class in other languages) that implements the function `map` that, while running over each value in the object to produce a new object.

Or in other words:

> Functor is an object, which implement a `map` contract!

Well that's Functor for you. But you might be wondering what Functor is useful for? We are going to answer that in the upcoming section.

■ **Note** that Functor is a *concept* that looks for a contract. The contract as we have seen is simple, implement `map`! The way in which we implement `map` function provides different types of Functor like `MayBe`, `Either`, which we are going to discuss later in this chapter.

MayBe

We started the chapter with the argument of how we handle errors/exception using functional programming techniques. In the previous section we learned about the fundamental concept of Functor. In this section, we are going to see a *type* of Functor called as `MayBe`. The `MayBe` functor allows us to handle errors in our code in a more functional way.

Implementing MayBe

`MayBe` is a type of Functor, which means it's going to implement a `map` function but in a *different way*. Let's start with a simple `MayBe`, which can hold the data (very similar to `Container` implementation):

Listing 8-6. `MayBe` Function Definition

```
const MayBe = function(val) {
  this.value = val;
}

MayBe.of = function(val) {
  return new MayBe(val);
}
```

We just created `MayBe`, which resembles the `Container` implementation. Now as stated earlier, we have to implement a `map` contract for the `MayBe`, which looks like this:

Listing 8-7. MayBe's map function definition

```
MayBe.prototype.isNothing = function() {
  return (this.value === null || this.value === undefined);
};
MayBe.prototype.map = function(fn) {
  return this.isNothing() ? MayBe.of(null) : MayBe.of(fn(this.value));
};
```

The map function does very similar things to the Container (simple Functor) map function. MayBe's map first checks whether the value in the container is null or undefined before applying the passed function using the isNothing function, which takes care of null/undefined checks:

```
(this.value === null || this.value === undefined);
```

Note that map puts back the result of applying the function back in the container:

```
return this.isNothing() ? Maybe.of(null) : Maybe.of(f(this.__value));
```

Now it's time to see MayBe in action.

Simple Use Cases

As we discussed in the previous section, MayBe does checks the null, undefined before applying the passed function in map. This is a very powerful abstraction that takes care of error handling! To make this concrete, here's a simple example:

Listing 8-8. Creating our first MayBe

```
MayBe.of("string").map((x) => x.toUpperCase())
```

which returns

```
MayBe { value: 'STRING' }
```

The most important and interesting point to note here:

```
(x) => x.toUpperCase()
```

doesn't care if x is null or undefined or that it has been abstracted by the MayBe functor! What if the value of the string is null? Then the code looks like the following:

```
MayBe.of(null).map((x) => x.toUpperCase())
```

we will be getting back:

```
MayBe { value: null }
```

Wow, our code now doesn't explode in `null` or `undefined` values as we have wrapped our value in the *type safety* container `MayBe`! We are now handling the null values in a declarative way.

■ **Note** On `MayBe.of(null)` case, if we call `map` function, from our implementation we know that `map` first checks if the value is `null` or `undefined` by calling `isNothing`:

```
//implementation of map
MayBe.prototype.map = function(fn) {
  return this.isNothing() ? MayBe.of(null) : MayBe.of(fn(this.value));
};
```

if `isNothing` returns `true`. We return back `MayBe.of(null)` instead of calling the passed function!

In a normal imperative way, we would have done this:

```
let value = "string"
if(value != null || value != undefined)
        return value.toUpperCase();
```

The above code does exactly the same thing, but look at the steps required to check if the `value` is `null` or `undefined`, even for a single call! And also using `MayBe`, we don't care about those sneaky variables to hold the resulting value! Remember that we can chain our `map` function as desired:

Listing 8-9. Chaining with `map`

```
MayBe.of("George")
     .map((x) => x.toUpperCase())
     .map((x) => "Mr. " + x)
```

gives back:

```
MayBe { value: 'Mr. GEORGE' }
```

It's so pleasant to watch! Before we close this section, we need to talk about two more important properties of `MayBe`.

The first one is that even if your passed function to `map` returns `null`/`undefined`, `MayBe` can take care of it! In other words, in the whole chain of `map` calls, it is fine if a function returns `null` or `undefined`. To illustrate the point, let's tweak the last example:

```
MayBe.of("George")
     .map(() => undefined)
     .map((x) => "Mr. " + x)
```

Note that our second map function returns undefined; however, running the above code will give this result:

```
MayBe { value: null }
```

as expected!

The second important point is that all map functions will be called regardless if it receives null/undefined. We'll pull out the same code snippet (Listing 8-9) that we have used in the previous example:

```
MayBe.of("George")
     .map(() => undefined)
     .map((x) => "Mr. " + x)
```

The point here is that even though the first map does return undefined:

```
map(() => undefined)
```

the second map will be called *always* (i.e., the chained maps to any level will be called always); it is just that the next map function in the chain returns undefined (as the previous map returns undefined/null), without applying the passed function! This process is repeated until the last map function call is evaluated in the chain.

Real-World Use Cases

Since MayBe is a type of container that can hold any values, it can also hold values of type Array. Imagine you have written an API to get the top 10 subreddit data based on types like top, new, hot:

Listing 8-10. Getting Top 10 SubReddit Posts

```
let getTopTenSubRedditPosts = (type) => {
    let response
    try{
       response = JSON.parse(request('GET',"https://www.reddit.com/r/
       subreddits/" + type + ".json?limit=10").getBody('utf8'))
    }catch(err) {
        response = { message: "Something went wrong" , errorCode:
err['statusCode'] }
    }
    return response
}
```

---

request comes from the package sync-request. This will allow us to fire a request and get the response in synchronous fashion. This is just for illustration, and I don't recommend using synchronous calls in production.

`getTopTenSubRedditPosts` function just hits the URL and gets back the response. If there are any issues in hitting the reddit API, it sends back a custom response of the format:

```
. . .
response = { message: "Something went wrong" , errorCode: err['statusCode']
}
. . .
```

and if we call our API like this:

```
getTopTenSubRedditPosts('new')
```

we will be getting back the response in this format:

```
{"kind": "Listing", "data": {"modhash": "", "children": [], "after": null,
"before": null}}
```

where `children` property will have an array of JSON objects. It will look something like this:

```
"{
  "kind": "Listing",
  "data": {
    "modhash": "",
    "children": [
      {
        "kind": "t3",
        "data": {
          . . .
          "url": "https://twitter.com/malyw/status/780453672153124864",
          "title": "ES7 async/await landed in Chrome",
          . . .
        }
      }
    ],
    "after": "t3_54lnrd",
    "before": null
  }
}"
```

From the response we need to return the array of JSON object that has the `URL` and `title` in it. Remember that if we pass an invalid subreddit type such as `test` to our `getTopTenSubRedditPosts`, it will return an error response that does not have a `data` or `children` property.

With MayBe in place, we can go ahead and implement the logic as such:

Listing 8-11. Getting Top 10 SubReddit Posts using MayBe

```
//arrayUtils from our library
import {arrayUtils} from '../lib/es6-functional.js'

let getTopTenSubRedditData = (type) => {
    let response = getTopTenSubRedditPosts(type);
    return MayBe.of(response).map((arr) => arr['data'])
                             .map((arr) => arr['children'])
                             .map((arr) => arrayUtils.map(arr,
                                (x) => {
                                    return {
                                        title : x['data'].title,
                                        url   : x['data'].url
                                    }
                                }
                            ))
}
```

Let's break down how getTopTenSubRedditData works. First we are wrapping the result of the reddit API call within the MayBe context using MayBe.of(response). Then we are running a series of functions using MayBe's map:

```
. . .
.map((arr) => arr['data'])
.map((arr) => arr['children'])
. . .
```

This will return the children array object from the response structure:

```
{"kind": "Listing", "data": {"modhash": "", "children": [ . . . .], "after":
null, "before": null}}
```

And in the last map, we are using our own ArrayUtils's map to iterate over the children property and return only the title and URL as needed:

```
. . .
.map((arr) =>
        arrayUtils.map(arr,
    (x) => {
        return {
            title : x['data'].title,
            url   : x['data'].url
        }
    }
. . .
```

Now if we call our function with a valid reddit name like new:

```
getTopTenSubRedditData('new')
```

we get back the response:

```
MayBe {
  value:
   [ { title: '/r/UpliftingKhabre - The subreddit for uplifting and positive
   stories from India!',
       url: 'https://www.reddit.com/r/upliftingkhabre' },
     { title: '/R/JerkOffToCelebs - The Best Place To Jerk Off To Your Fave
     Celebs',
       url: 'https://www.reddit.com/r/JerkOffToCelebs' },
     { title: 'Angel Vivaldi channel',
       url: 'https://qa1web-portal.immerss.com/angel-vivaldi/angel-vivaldi' },
     { title: 'r/SuckingCock - Come check us out for INSANE Blowjobs!
     (NSFW)',
       url: 'https://www.reddit.com/r/suckingcock/' },
     { title: 'r/Just_Tits - Come check us out for GREAT BIG TITS! (NSFW)',
       url: 'https://www.reddit.com/r/just_tits/' },
     { title: 'r/Just_Tits - Come check us out for GREAT BIG TITS! (NSFW)',
       url: 'https://www.reddit.com/r/just_tits/' },
     { title: 'How to Get Verified Facebook',
       url: 'http://imgur.com/VffRnGb' },
     { title: '/r/TrollyChromosomes - A support group for those of us whose
     trollies or streetcars suffer from chronic genetic disorders',
       url: 'https://www.reddit.com/r/trollychromosomes' },
     { title: 'Yemek Tarifleri Eskimeyen Tadlarımız',
       url: 'http://otantiktad.com/' },
     { title: '/r/gettoknowyou is the ultimate socializing subreddit!',
       url: 'https://www.reddit.com/r/subreddits/comments/50wcju/
       rgettoknowyou_is_the_ultimate_socializing/' } ] }
```

The above response might not be the same for the readers, as the response will change from time to time.

The beauty of the getTopTenSubRedditData method is how it handles unexpected input that can cause null/undefined errors in our logic flow. What if someone calls your getTopTenSubRedditData with a wrong reddit type? Remember that it will return the JSON response from Reddit:

```
{ message: "Something went wrong" , errorCode: 404 }
```

That is, the `data` – `children` property – will be empty! Let's try this by passing the wrong reddit type and see how it responds:

```
getTopTenSubRedditData('new')
```

which returns:

```
MayBe { value: null }
```

without throwing any error! Even though our `map` function tries to get the `data` from the response (which is not present in this case), it returns `MayBe.of(null)`, so the corresponding `maps` would not apply the passed function, as we have discussed earlier.

We can clearly sense how `MayBe` handled all the `undefined`/`null` errors with ease! Our `getTopTenSubRedditData` looks so declarative!

That's all about the `MayBe` Functor. We are going to meet another functor in the next section called `Either.`

`Either` Functor

In this section we are going to create a new functor called `Either`. `Either` will allow us to solve the branching-out problem. To give a context, let's see an example from the previous section (Listing 8-9):

```
MayBe.of("George")
     .map(() => undefined)
     .map((x) => "Mr. " + x)
```

The above code will return the result as:

```
MayBe {value: null}
```

as we would expect. But the question is, which branching (i.e., out of two `map` calls above) failed with `undefined` or `null` values. We can't answer this question easily with `MayBe`. The only way is to manually dig into the branching of `MayBe` and discover the culprit! This doesn't mean that `MayBe` has flaws, but just that in certain use cases, we need a better Functor than `MayBe` (mostly where you have many nested `maps`). This is where `Either` comes into the picture.

Implementing `Either`

We have seen the problem `Either` is going to solve for us; now let's see its implementation:

Listing 8-12. Either Functor Parts Definition

```
const Nothing = function(val) {
  this.value = val;
};

Nothing.of = function(val) {
  return new Nothing(val);
};

Nothing.prototype.map = function(f) {
  return this;
};

const Some = function(val) {
  this.value = val;
};

Some.of = function(val) {
  return new Some(val);
};

Some.prototype.map = function(fn) {
  return Some.of(fn(this.value));
}
```

The implementation has two functions, namely, `Some` and `Nothing`. You can see that `Some` is just a copy of a `Container` with a name change. The interesting part is with `Nothing`. `Nothing` is also a `Container`, but its `map` doesn't run over a given function but rather just returns:

```
Nothing.prototype.map = function(f) {
  return this;
};
```

In other words, you can run your functions on `Some` but not on `Nothing` (not a technical statement right? :)

Here's a quick example:

```
Some.of("test").map((x) => x.toUpperCase())
=> Some {value: "TEST"}

Nothing.of("test").map((x) => x.toUpperCase())
=> Nothing {value: "test"}
```

As shown in the above code snippet, calling `map` on `Some` runs over the passed function. However, in `Nothing`, it just returns the same value back `test`. And we will wrap these two objects into `Either` object like this:

Listing 8-13. Either Definition

```
const Either = {
  Some : Some,
  Nothing: Nothing
}
```

You might be wondering, what's the usefulness of `Some` or `Nothing`. To understand this, let's revisit our reddit example version of `MayBe`.

Reddit Example `Either` Version

The `MayBe` version of reddit example looks like (Listing 8-11):

```
let getTopTenSubRedditData = (type) => {
    let response = getTopTenSubRedditPosts(type);
    return MayBe.of(response).map((arr) => arr['data'])
                             .map((arr) => arr['children'])
                             .map((arr) => arrayUtils.map(arr,
                                (x) => {
                                    return {
                                        title : x['data'].title,
                                        url   : x['data'].url
                                    }
                                }
                            ))
}
```

on passing a wrong reddit type, say, for example, `unknown`:

```
getTopTenSubRedditData('unknown')
=> MayBe {value : null}
```

we get back `MayBe` of `null` value. But we didn't know the reason why `null` was returned! We know that `getTopTenSubRedditData` uses `getTopTenSubRedditPosts` to get the response. Now that `Either` in place, we can create a new version of `getTopTenSubRedditPosts` using `Either`:

Listing 8-14. Get Top Ten Subreddit Using Either

```
let getTopTenSubRedditPostsEither = (type) => {

    let response
    try{
       response = Some.of(JSON.parse(request('GET',"https://www.reddit.
       com/r/subreddits/" + type + ".json?limit=10").getBody('utf8')))
    }catch(err) {
        response = Nothing.of({ message: "Something went wrong" , errorCode:
        err['statusCode'] })
    }
    return response
}
```

Note that we have wrapped the proper response with `Some` and error response with `Nothing`! Now with that in place, we can modify our reddit API to:

Listing 8-15. Get Top Ten Subreddit Using Either

```
let getTopTenSubRedditDataEither = (type) => {
    let response = getTopTenSubRedditPostsEither(type);
    return response.map((arr) => arr['data'])
                           .map((arr) => arr['children'])
                           .map((arr) => arrayUtils.map(arr,
                              (x) => {
                                  return {
                                      title : x['data'].title,
                                      url   : x['data'].url
                                  }
                              }
                          ))
}
```

The above code is just literally the `MayBe` version, but it's just not using `MayBe`, rather it's using `Either`'s type.

Now let's call our new API with the wrong reddit data type:

```
getTopTenSubRedditDataEither('new2')
```

which will return:

```
Nothing { value: { message: 'Something went wrong', errorCode: 404 } }
```

This is so brilliant! Now with `Either` types in place, we get back the exact reason why our branching failed! As you can guess, `getTopTenSubRedditPostsEither` returns `Nothing` in case of an error (i.e., unknown reddit type); hence the mappings on `getTopTenSubRedditDataEither` will never happen since it is of type `Nothing`! You can

sense how Nothing helped us in preserving the error message and also blocking the functions to map over!

On a closing note, we can try our new version with a valid reddit type:

```
getTopTenSubRedditDataEither('new')
```

It will give back the expected response in Some:

```
Some {
  value:
   [ { title: '/r/UpliftingKhabre - The subreddit for uplifting and positive
   stories from India!',
       url: 'https://www.reddit.com/r/upliftingkhabre' },
     { title: '/R/JerkOffToCelebs - The Best Place To Jerk Off To Your Fave
     Celebs',
       url: 'https://www.reddit.com/r/JerkOffToCelebs' },
     { title: 'Angel Vivaldi channel',
       url: 'https://qa1web-portal.immerss.com/angel-vivaldi/angel-vivaldi' },
     { title: 'r/SuckingCock - Come check us out for INSANE Blowjobs!
     (NSFW)',
       url: 'https://www.reddit.com/r/suckingcock/' },
     { title: 'r/Just_Tits - Come check us out for GREAT BIG TITS! (NSFW)',
       url: 'https://www.reddit.com/r/just_tits/' },
     { title: 'r/Just_Tits - Come check us out for GREAT BIG TITS! (NSFW)',
       url: 'https://www.reddit.com/r/just_tits/' },
     { title: 'How to Get Verified Facebook',
       url: 'http://imgur.com/VffRnGb' },
     { title: '/r/TrollyChromosomes - A support group for those of us whose
     trollies or streetcars suffer from chronic genetic disorders',
       url: 'https://www.reddit.com/r/trollychromosomes' },
     { title: 'Yemek Tarifleri Eskimeyen Tadlarımız',
       url: 'http://otantiktad.com/' },
     { title: '/r/gettoknowyou is the ultimate socializing subreddit!',
       url: 'https://www.reddit.com/r/subreddits/comments/50wcju/
       rgettoknowyou_is_the_ultimate_socializing/' } ] }
```

That's all about Either.

If you are from a Java background, you can sense that Either is very similar to Optional in Java 8. In fact, Optional is a functor!

Word of Caution - Pointed Functor

Before we close the chapter, I need to make a point clear. In the beginning of the chapter we started saying that we created the `of` method just to escape the `new` keyword in place for creating `Container`. We did the same for `MayBe` and `Either` as well. To recall, Functor is just an interface that has a `map` contract. *Pointed Functor* is a subset of Functor, which has an interface that has a `of` contracts!

So what we have designed until now is called a *Pointed Functor*! This is just to make the terminologies right in the book! But you got to see what he problem does *Functor* or *Pointed Functor* solves for us in the real world, which is much important.

E6 adds `Array.of` making arrays a pointed functor!

Array.of("You are a pointed functor, too?")

Summary

We started our chapter with asking questions about how we will be handling exceptions in the functional programming world. We began with creating a simple Functor. We defined a Functor as being nothing but a container with a `map` function implemented. Then we went ahead and implemented a functor called `MayBe`. We saw how `MayBe` helps us in avoiding those pesky `null/undefined` checks. `MayBe` allowed us to write code in functional and declarative ways. Then we saw how `Either` helped us to preserve the error message while branching out. `Either` is just a supertype of `Some` and `Nothing`. Now we have seen Functors in action!

CHAPTER 9

Monads in Depth

■ **Note** The chapter examples and library source code are in branch `chap09`. The repo's URL is: `https://github.com/antoaravinth/functional-es6.git`

Once checkout the code, please checkout branch `chap09`:

```
...
git checkout -b chap09 origin/chap09
...
```

For running the codes, as before run:

```
...
npm run playground
...
```

In the previous chapter we have seen what Functors are and how they are useful to us. In this chapter we are going to continue with Functors. We will learn about a new functor called Monads. Don't be afraid with the terms: the concepts are easy.

We are going to start with a problem of retrieving and displaying the reddit comments for our search query. Initially we are going to use Functors, especially the `MayBe` functor, to solve this problem. But while we solve the problem, we are going to encounter a few issues with the `MayBe` functor. Then we will be moving ahead to create a special type of functor called *Monad*.

Getting Reddit Comments for Our Search Query

We have been using reddit API starting from the previous chapter. In this section, too, we will be using the reddit API for searching the posts with our query and getting the list of comments for each of the search results. We are going to use `MayBe` for this problem; as we have seen in the previous chapter, `MayBe` allows us to focus on the problem, without worrying about those pesky `null`/`undefined` values.

A. Aravinth, *Beginning Functional JavaScript*, DOI 10.1007/978-1-4842-2656-8_9

You might be wondering why not the `Either` functor for the current problem, as `MayBe` has a few drawbacks of not capturing the error when branching out as we have seen in the previous chapter. That's true, but the reason I have chosen `MayBe` is mainly to keep things simple. As you see, we will be extending the same idea to `Either` as well!

The Problem

Before we begin implementing the solution, let's look at the problem and its associated Reddit API endpoints. The problem contains two steps:

1. For searching a specific posts/comments/ we need to hit the Reddit API endpoint:

   https://www.reddit.com/search.json?q=<SEARCH_STRING>

and pass along the `SEARCH_STRING`. For example, if we search for the string `functional programming` like this:

https://www.reddit.com/search.json?q=functional%20programming

we get back:

Listing 9-1. Structure of Reddit Response

```
{ kind: 'Listing',
  data:
   { facets: {},
     modhash: '',
     children:
      [ [Object],
        [Object],
        [Object],
        [Object],
        [Object],
        [Object],
        . . .
        [Object],
        [Object] ],
     after: 't3_terth',
     before: null } }
```

and each `children` object looks like this:

```
{ kind: 't3',
  data:
   { contest_mode: false,
     banned_by: null,
```

```
domain: 'self.compsci',
. . .
downs: 0,
mod_reports: [],
archived: true,
media_embed: {},
is_self: true,
hide_score: false,
permalink: '/r/compsci/comments/3mecup/eli5_what_is_functional_
programming_and_how_is_it/?ref=search_posts',
locked: false,
stickied: false,
. . .
visited: false,
num_reports: null,
ups: 134 } }
```

where these objects specify the results that are matching our search query.

2. Once we have the search result, we need to get each search result's comments. How do we do it? As mentioned in the previous point, each `children` object is our search result. These objects have a field called `permalink`, which looks like :

   ```
   permalink: '/r/compsci/comments/3mecup/eli5_what_is_functional_
   programming_and_how_is_it/?ref=search_posts',
   ```

we need to navigate to the above URL:

```
GET: https://www.reddit.com//r/compsci/comments/3mecup/eli5_what_is_
functional_programming_and_how_is_it/.json
```

which is going to return the array of comments like the following:

```
[Object,Object,..,Object]
```

where each `Object` gives the information about comments.

Once we get the comments object, we need to merge the result with `title` and return as a new Object:

```
{
        title : Functional programming in plain English,
        comments : [Object,Object,..,Object]
}
```

where `title` is the title we get from the first step. Now with our understanding of the problem, let's go and implement the logic.

Implementation of the First Step

Let's implement the solution step by step. In this section, we'll implement the solution for the first step. The first step involves hitting the Reddit search API endpoint along with our search query. Since we need to fire the HTTP GET call, we will be requiring the `sync-request` module that we used in the previous chapter.

Let's pull out the module by and hold it in a variable for future use:

```
let request = require('sync-request');
```

Now with the `request` function, we could fire HTTP GET call to our Reddit Search API endpoint. Let's wrap the search steps in a specific function, which we call `searchReddit`:

Listing 9-2. `searchReddit` function definition

```
let searchReddit = (search) => {
    let response
    try{
       response = JSON.parse(request('GET',"https://www.reddit.com/search.
       json?q=" + encodeURI(search)).getBody('utf8'))
    }catch(err) {
        response = { message: "Something went wrong" , errorCode:
        err['statusCode'] }
    }
    return response
}
```

Now we'll walk down the code in steps:

1. We are firing the search request to the URL endpoint `https://www.reddit.com/search.json?q=` as shown here:

   ```
   response = JSON.parse(request('GET',"https://www.reddit.com/
   search.json?q=" + encodeURI(search)).getBody('utf8'))
   ```

Note that we are using the `encodeURI` method for escaping special characters in our search string.

2. Once the response is a success, we are returning back the value.
3. In case of error, we are catching it in a `catch` block and getting the error code and returning the error response like this:

```
. . .
catch(err) {
        response = { message: "Something went wrong" , errorCode:
        err['statusCode'] }
    }
. . .
```

With our little function in place, we go ahead and test it:

```
searchReddit("Functional Programming")
```

which will return the result:

```
{ kind: 'Listing',
  data:
   { facets: {},
     modhash: '',
     children:
      [ [Object],
        [Object],
        [Object],
        [Object],
        [Object],
        [Object],
        [Object],
        [Object],
        . . .
     after: 't3_terth',
     before: null } }
```

That's perfect! We are done with step 1! Let's go and implement step 2.

Implementing the decond Step For each search `children` object, we need to get its `permalink` value to get the list of comments. We can write a separate method for getting s list of comments for the given URL. We call this method `getComments`. The implementation of `getComments` is simple, which looks like the following:

Listing 9-3. `getComments` function definition

```
let getComments = (link) => {
    let response
    try {
        response = JSON.parse(request('GET',"https://www.reddit.com/" +
        link).getBody('utf8'))
    } catch(err) {
        response = { message: "Something went wrong" , errorCode:
        err['statusCode'] }
    }

    return response
}
```

getComments implementation is very similar to our searchReddit. Let's walk down in steps and see what getComments does:

1. It fires the HTTP GET call for the given link value. For example, if the link value is passed as:

   ```
   r/IAmA/comments/3wyb3m/we_are_the_team_working_on_react_native_
   ask_us/.json
   ```

getComments then will fire a HTTP GET call to the URL:

```
https://www.reddit.com/r/IAmA/comments/3wyb3m/we_
are_the_team_working_on_react_native_ask_us/.json
```

which is going to return the array of comments. As before, we are a bit defensive here and catching any errors within the getComments method in our favorite catch block. And finally we are returning back the response.

Quickly we'll test our getComments, by passing the below link value:

```
r/IAmA/comments/3wyb3m/we_are_the_team_working_on_react_native_ask_us/.json

getComments('r/IAmA/comments/3wyb3m/we_are_the_team_working_on_react_native_
ask_us/.json')
```

for the above call we get back:

```
[ { kind: 'Listing',
    data: { modhash: '', children: [Object], after: null, before: null } },
  { kind: 'Listing',
    data: { modhash: '', children: [Object], after: null, before: null } } ]
```

the result. Now with both APIs ready, it's time to go and merge these results.

Merging Reddit Calls. Now we have defined two functions, namely, searchReddit and getComments (Listing 9-2 and Listing 9-3 respectively), which does it tasks and return the response as seen in the previous sections. In this section, let's write a higher-level function, which takes up the search text and use these two functions to achieve our end goal.

We'll call the function we create as mergeViaMayBe and its implementation looks like the following:

Listing 9-4. mergeViaMayBe function definition

```
let mergeViaMayBe = (searchText) => {

    let redditMayBe = MayBe.of(searchReddit(searchText))
    let ans = redditMayBe
                .map((arr) => arr['data'])
                .map((arr) => arr['children'])
```

```
                .map((arr) => arrayUtils.map(arr,(x) => {
                        return {
                            title : x['data'].title,
                            permalink : x['data'].permalink
                        }
                    }
                ))
                .map((obj) => arrayUtils.map(obj, (x) => {
                    return {
                        title: x.title,
                       comments: MayBe.of(getComments(x.permalink.
                       replace("?ref=search_posts",".json")))
                    }
                }));

   return ans;
}
```

Let's quickly check our function by passing the search text `functional programming`:

```
mergeViaMayBe('functional programming')
```

the above call will give the result:

```
MayBe {
  value:
   [ { title: 'ELI5: what is functional programming and how is it different
   from OOP',
       comments: [Object] },
     { title: 'ELI5 why functional programming seems to be "on the rise" and
     how it differs from OOP',
       comments: [Object] } ] }
```

For better clarity I have reduced the number of results in the output of the above call. The default call will give back 25 results, which will take a couple of pages to put in the output of `mergeViaMayBe`. From here on, I will be displaying only minimal output in the book. Kindly note that the source code example does call and print all of the 25 results.

Great! Now let's step back and understand in detail what `mergeViaMayBe` function does.

The function first calls the `searchReddit` with `searchText` value. The result of the call is wrapped in `MayBe`:

```
let redditMayBe = MayBe.of(searchReddit(searchText))
```

and once the result is wrapped inside a `MayBe` type, we are free to `map` over it as you can see in the code.

To remind us of the search query (which our `searchReddit` will call), it will send back the result in the following structure:

```
{ kind: 'Listing',
  data:
   { facets: {},
     modhash: '',
     children:
      [ [Object],
        [Object],
        [Object],
        [Object],
        [Object],
        [Object],
        . . .
        [Object],
        [Object] ],
     after: 't3_terth',
     before: null } }
```

In order to get the `permalink` (which is in our `children` object), we need to navigate to `data.children`. This is exactly demonstrated in the code:

```
redditMayBe
         .map((arr) => arr['data'])
         .map((arr) => arr['children'])
```

Now we got the handle to `children` array. Remember that each `children` has an object with the following structure:

```
{ kind: 't3',
  data:
   { contest_mode: false,
     banned_by: null,
     domain: 'self.compsci',
     . . .
     permalink: '/r/compsci/comments/3mecup/eli5_what_is_functional_
     programming_and_how_is_it/?ref=search_posts',
     locked: false,
     stickied: false,
     . . .
     visited: false,
     num_reports: null,
     ups: 134 } }
```

We need to get only `title` and `permalink` out of it; since it's an array, we run `Array`'s `map` function over it:

```
.map((arr) => arrayUtils.map(arr,(x) => {
        return {
            title : x['data'].title,
            permalink : x['data'].permalink
        }
    }
))
```

Now we have both `title` and `permalink`, our last step is to take `permalink` and pass it to our `getComments` function, which will fetch the list of comments for the passed value. This is seen here in the code:

```
.map((obj) => arrayUtils.map(obj, (x) => {
        return {
            title: x.title,
           comments: MayBe.of(getComments(x.permalink.replace("?ref=search_
           posts",".json")))
        }
}));
```

Since the call of `getComments` can get an error value, we are wrapping it again inside a `MayBe`:

```
. . .
      comments: MayBe.of(getComments(x.permalink.replace("?ref=search_
      posts",".json")))
. . .
```

■ **Note** That we are replacing the `permalink` value `?ref=search_posts` with `.json` as search results append the value `?ref=search_posts`, which is not the correct format for the `getComments` API call.

That's it!

Throughout the full process we haven't come outside our `MayBe` type. We run our all `map` function happily on our `MayBe` type without worrying about it too much! We have solved our problem so elegantly with `MayBe`, didn't we? There is a slight problem with our `MayBe` functor that is used this way. Let's talk about it in next section.

Problem of So Many maps

If you count the number of map calls on our MayBe in our mergeViaMayBe function, its 4! You might be thinking what is the big deal about it? Who cares about the number of map calls! Do we?

Let's try to understand the problem of many *chained* map calls like in mergeViaMayBe. Now imagine we want to get a comments array that is returned from mergeViaMayBe.

We'll pass our search text functional programming in our mergeViaMayBe function:

```
let answer = mergeViaMayBe("functional programming")
```

after the call answer:

```
MayBe {
  value:
   [ { title: 'ELI5: what is functional programming and how is it different
   from OOP',
       comments: [Object] },
     { title: 'ELI5 why functional programming seems to be "on the rise" and
     how it differs from OOP',
       comments: [Object] } ] }
```

Now let's get the comments object for processing. Since the return value is MayBe, we can map over it:

```
answer.map((result) => {
      //process result.
})
```

The result (which is the value of MayBe) is an array that has title and comments, so let's map over it using our Array's map:

```
answer.map((result) => {
    arrayUtils.map(result,(mergeResults) => {
        //mergeResults
    })
})
```

Each mergeResults is an object, which has title and comments. Remember that comments are also a MayBe! So in order to get comments, we need to map over our comments:

```
answer.map((result) => {
    arrayUtils.map(result,(mergeResults) => {
        mergeResults.comments.map(comment => {
            //finally got the comment object!
        })
    })
})
```

It looks like we have done more work to get the list of comments! Imagine someone is using our mergeViaMayBe API to get the comments list. They will be really irritated to get back the result using nested maps as you can see above. Can we make our mergeViaMayBe better? Yes we can – meet *Monads*!

Solving the Problem via join

We saw in previous sections how deep we have to go inside our MayBe to get back our desired results! Writing such API's is not going to help us definitely but rather irritate other developers working on it! In order to solve these deep-nested issues, let's add join to the MayBe functor.

join Implementation

Let's start implementing the join function. The join function is simple and looks like the following:

Listing 9-5. join function definition

```
MayBe.prototype.join = function() {
  return this.isNothing() ? MayBe.of(null) : this.value;
}
```

join is very simple and it simply returns the value inside our container (if there are values); if not, returning MayBe.of(null). join is simple, but it helps us to unwrap the nested MayBe's:

```
let joinExample = MayBe.of(MayBe.of(5))

=> MayBe { value: MayBe { value: 5 } }

joinExample.join()
=> MayBe { value: 5 }
```

As shown in the above example, it unwraps the nested structure into a single level! Imagine we want to add 4 to our value in joinExample MayBe. Let's give it a try:

```
joinExample.map((outsideMayBe) => {
    return outsideMayBe.map((value) => value + 4)
})
```

The above code gives back:

```
MayBe { value: MayBe { value: 9 } }
```

Even though the value is correct, we have mapped twice to get the result. Again the result that we got ends up in a nested structure! Now let's do the same via `join`:

```
joinExample.join().map((v) => v + 4)

=> MayBe { value: 9 }
```

Wow, the above code is just elegant! The call to `join` returns the inside `MayBe`, which has the value of 5; once we have that, we are running over it via `map` and thenadd the value 4. Now the resulting value is in a *flatten* structure `MayBe { value: 9 }`.

Now with `join` in place, let's go and try to level our nested structure returned by `mergeViaMayBe`. We'll change the code to the following:

Listing 9-6. `mergeViaMayBe` using `join`

```
let mergeViaJoin = (searchText) => {
    let redditMayBe = MayBe.of(searchReddit(searchText))
    let ans = redditMayBe.map((arr) => arr['data'])
                .map((arr) => arr['children'])
                .map((arr) => arrayUtils.map(arr,(x) => {
                         return {
                             title : x['data'].title,
                             permalink : x['data'].permalink
                         }
                     }
                 ))
                .map((obj) => arrayUtils.map(obj, (x) => {
                     return {
                         title: x.title,
                        comments: MayBe.of(getComments(x.permalink.
                        replace("?ref=search_posts",".json"))).join()
                     }
                }))
                .join()

   return ans;
}
```

As you can see, we have just added two `joins` in our code. One is on the `comments` section, where we create a nested `MayBe;` and another one is right after our all `map` operation.

Now with `mergeViaJoin` in place, let's go and implement the same logic of getting the `comments` array out of the result.

First let's quickly see the response returned by `mergeViaJoin`:

```
mergeViaJoin("functional programming")
```

which is going to return:

```
[ { title: 'ELI5: what is functional programming and how is it different
from OOP',
    comments: [ [Object], [Object] ] },
  { title: 'ELI5 why functional programming seems to be "on the rise" and
  how it differs from OOP',
    comments: [ [Object], [Object] ] } ]
```

Compare the above result with our old `mergeViaMayBe`:

```
MayBe {
  value:
   [ { title: 'ELI5: what is functional programming and how is it different
   from OOP',
       comments: [Object] },
     { title: 'ELI5 why functional programming seems to be "on the rise" and
     how it differs from OOP',
       comments: [Object] } ] }
```

As you can see, `join` has taken out the `MayBe`'s value and sent it back. Now lets see how to use the `comments` array for our processing task. Since the value returned from `mergeViaJoin` is an array, we can `map` over it using our `Arrays` map:

```
arrayUtils.map(result, mergeResult => {
    //mergeResult
})
```

Now each `mergeResult` variable directly points to the object that has `title` and `comments`. Note that we have called `join` in our `MayBe` call of `getComments`, so the `comments` object is just a simple array! With that in mind, to get the list of `comments` from the iteration, we just need to call `mergeResult.comments`:

```
arrayUtils.map(result,mergeResult => {
    //mergeResult.comments has the comments array!
})
```

This looks promising, as we have gotten the full benefit of our `MayBe` and also a good data structure to return the result, which are easy for processing!

`chain` Implementation

Have a look at the code in listing 9-6. As you can guess, we need to call `join` always after `map`. Let's wrap this logic inside a method called `chain`:

Listing 9-7. chain function definition

```
MayBe.prototype.chain = function(f){
  return this.map(f).join()
}
```

Once chain is in place, we can make our merge function logic to looks like this:

Listing 9-8. mergeViaMayBe using chain

```
let mergeViaChain = (searchText) => {
    let redditMayBe = MayBe.of(searchReddit(searchText))
    let ans = redditMayBe.map((arr) => arr['data'])
                .map((arr) => arr['children'])
                .map((arr) => arrayUtils.map(arr,(x) => {
                         return {
                             title : x['data'].title,
                             permalink : x['data'].permalink
                         }
                     }
                 ))
                .chain((obj) => arrayUtils.map(obj, (x) => {
                     return {
                        title: x.title,
                        comments: MayBe.of(getComments(x.permalink.
                        replace("?ref=search_posts",".json"))).join()
                     }
                }))

   return ans;
}
```

The output is going to be exactly the same via chain too! Go and play around with above function! In fact, with chain in place, we can move the logic of counting the number of comments to an in-place operation:

Listing 9-9. Making improvements on mergeViaChain

```
let mergeViaChain = (searchText) => {
    let redditMayBe = MayBe.of(searchReddit(searchText))
    let ans = redditMayBe.map((arr) => arr['data'])
                .map((arr) => arr['children'])
                .map((arr) => arrayUtils.map(arr,(x) => {
                         return {
                             title : x['data'].title,
                             permalink : x['data'].permalink
                         }
                     }
                 ))
```

```
                .chain((obj) => arrayUtils.map(obj, (x) => {
                     return {
                        title: x.title,
                        comments: MayBe.of(getComments(x.permalink.
                        replace("?ref=search_posts",".json"))).chain(x => {
                             return x.length
                        })
                     }
                }))

   return ans;
}
```

Now calling the above code:

```
mergeViaChain("functional programming")
```

will return the following:

```
[ { title: 'ELI5: what is functional programming and how is it different
from OOP',
    comments: 2 },
  { title: 'ELI5 why functional programming seems to be "on the rise" and
  how it differs from OOP',
    comments: 2 } ]
```

Bingo! We won! The solution looks so elegant! But still we haven't seen a Monad, have we?

So What Is a Monad?

You might be wondering why we started the chapter with a promise of teaching you a Monad! But still now we haven't defined what a Monad is. I'm sorry for not defining the Monad, but you have already *seen it in action*. (What?!!)

Yes, Monad is a functor that has a `chain` method! Yeah, that's it, that's what a Monad is! As you have already seen, we have extended our favorite `MayBe` functor to add a `chain` (and of course a `join` function) to make it a Monad!

We started with an example of a functor to solve an ongoing problem and ended up solving the problem using a Monad without even being aware of using it! That's intentional from my side as I wanted to see the intuition behind Monad (the problem it solves in hand with a functor)! I could have started with a simple definition of Monad, but that directly shows *what* a Monad is, but it won't show *why* a Monad!

You might be in confused thinking about whether `MayBe` is a Monad or a Functor! Don't get confused: `MayBe` with only `of` and `map` is a Functor. A functor with `chain` is a Monad!

Summary

In this chapter we have seen a new Functor type called Monad. We discussed the problem of how repetitive `map`s will cause nested values, which become tough to handle later! We introduced a new function called `chain`, which helps to flatten the `MayBe` data. We saw that a pointed functor with a `chain` is called a Monad. In the current chapter, we were using a third- party library to create AJAX calls. In the next chapter, we will be seeing a new way to think of Asynchronous calls.

CHAPTER 10

Pause, Resume with Generators

■ **Note** The chapter examples and library source code are in branch `chap10`. The repo's URL is: `https://github.com/antoaravinth/functional-es6.git`

Once checkout the code, please checkout branch `chap10`:

```
...
git checkout -b chap10 origin/chap10
...
```

For running the codes, as before run:

```
...
npm run playground
...
```

We started the book with a simple definition of functions. Then we constantly saw how to use functions to do great things using functional programming technique. We have seen how to handle arrays, objects, and error handling, in pure functional terms. It has been quite a long journey for us. But still we have not talked about yet another important technique that every JavaScript developer should be aware of – asynchronous code.

You have dealt with a whole lot of asynchronous codes in your project. You might be wondering whether functional programming can help developers in asynchronous code? The answer is yes and no. The technique that I'm going to showcase here is using ES6 **Generators**. Generators are new specs for functions in ES6. Generators are not really a functional programming technique; however, it's part of a function (functional programming is about function, right?); for that reason we have dedicated a chapter for it in this functional programming book!

A. Aravinth, *Beginning Functional JavaScript*, DOI 10.1007/978-1-4842-2656-8_10

Even if you are a big fan of promises (which is a technique for solving the callback problem), I would still advise you to have a look at this chapter. I will bet you, that you are going to love generators and the way they solve the async code problems!

Async Code and Its Problem

Before we really see what generators are, let's discuss the problem of handling async code in JavaScript in this section. We are going to talk about a *Callback Hell* problem. If you already knew what it is, you are free to move to the next section. For others, please read on.

Callback Hell

Imagine you have a function like the following:

Listing 10-1. Synchronous functions

```
let sync = () => {
        //some operation
        //return data
}

let sync2 = () => {
        //some operation
        //return data
}

let sync3 = () => {
        //some operation
        //return data
}
```

The above functions `sync`, `sync1`, and `sync2` do some operations synchronously and return the results. As a result, one can call these functions like this:

```
result = sync()
result2 = sync2()
result3 = sync3()
```

What if the operation is asynchronous? Let's see it in action:

Listing 10-2. Asynchronous functions

```
let async = (fn) => {
        //some async operation
        //call the callback with async operation
        fn(/*  result data */)
}

let async2 = (fn) => {
        //some async operation
        //call the callback with async operation
        fn(/*  result data */)
}

let async3 = (fn) => {
        //some async operation
        //call the callback with async operation
        fn(/*  result data */)
}
```

Synchronous vs. Asynchronous

Synchronous is when the function blocks the caller when it is executing and returns the result once it's available.

Asynchronous is when the function doesn't block the caller when it's executing the function but returns the result once available.

We deal with Asynchronous heavily when we deal with an AJAX request in our project.

Now if someone wants to process these functions at once, how they do it? The only way to do it is like this:

Listing 10-3. Async functions calling example

```
async(function(x){
    async2(function(y){
        async3(function(z){
            ...
        });
    });
});
```

Oops! You can see in the above code (Listing 10-3), we are passing many callback functions to our `async` functions! This little piece of code showcases what Callback Hell is! Callback Hell makes the program harder to understand. Handling errors and bubbling the errors out of callback are tricky and always error prone.

Before ES6 arrived, JavaScript developers used **Promises** to solve the above problem. Promises are great, but given the fact that ES6 has generators at language level, we don't need Promises anymore!

Generators 101

As mentioned, generators are part of ES6 specifications and it's bundled up at language level. We talked about using generators helping in handling async code. But before we get there, we are going to talk about the fundamentals of generators. This section will focus on explaining the core concepts behind generators. Once we learn the basics, we will be creating a generic function using generators to handle async code in our library. That's the plan of this chapter. Let's begin.

Creating Generators

Let's start our journey by seeing how to create generators in the first place. Generators are nothing but a function that comes up with its own syntax. A simple generator looks like the following:

Listing 10-4. First Simple Generator

```
function* gen() {
    return 'first generator';
}
```

The function gen in the above code is a generator. As you might notice, we have used an asterisk before our function name (in this case gen) to denote that it's a generator function! Al right, we have seen how to create a generator; now let's see how to invoke a generator:

```
let generatorResult = gen()
```

What will be the result of `generatorResult`? Is it going to be `first generator` value? Let's print it on the console and inspect it:

```
console.log(generatorResult)
```

The result will be:

```
gen {[[GeneratorStatus]]: "suspended", [[GeneratorReceiver]]: Window}
```

The above result shows that the `generatorResult` is not a normal function, but an instance of Generator primitive type! So the question is how to get the value from this generator instance? The answer is to call the function `next`, which is available on the generator instance. So to get the value you need to do this:

```
generator.next()
```

The above code returns:

```
Object {value: "hello world", done: true}
```

As you can see the returned object from `next` has `value` and is `done`. So we need to call `next` along with fetching a `value` from the object:

```
generator.next().value
=> 'first generator'
```

That's great!

Caveats of Generators

The above examples show how to create a generator, how to create an instance for it, and how it gets value. But there are a few important things we need to take care of while we are working with generators.

The first thing is that we can't call `next` as many times as we want to get the value from the generator. To make it clearer, let's try to fetch a `value` from our first generator (Refer Listing 10-4 for first generator definition):

```
let generatorResult = gen()

//for the first time
generatorResult.next().value
=> 'first generator'

//for the second time
generatorResult.next().value
=> undefined
```

As you can see in the above code, calling `next` for the second time will return an `undefined` rather than `first generator`. The reason is that generators are like sequences: once the values of the sequence are consumed, you can't consume it again. In our case `generatorResult` is a sequence that has value as a `first generator`. With our first call to `next`, we (as the caller of the generator) have consumed the value from the sequence. Since the sequence is empty by now, calling it a second time will return you `undefined`!

In order to consume the sequence again, you need to create another generator instance:

```
let generatorResult = gen()
let generatorResult2 = gen()

//first sequence
generatorResult.next().value
=> 'first generator'

//second sequence
generatorResult2.next().value
=> 'first generator'
```

The above code also shows that different instances of Generators can be in different states. The key takeaway here is that each generator's state depends on how we are calling the next function on it.

yield New Keyword

With generator functions, there is a new keyword that we can use called yield. In this section, we are going to see how to use yield within a generator function.

Let's start with the below code:

Listing 10-5. Simple Generator Sequence

```
function* generatorSequence() {
    yield 'first';
    yield 'second';
    yield 'third';
}
```

As usual we can create a generator instance for the above code:

```
let generatorSequence = generatorSequence();
```

Now if we call next for the first time we get back the value first:

```
generatorSequence.next().value
=> first
```

What happens if we call next again? Do we get first? Or second? Or third? Or Error? Let's find that out:

```
generatorSequence.next().value
=> second
```

What? We got back the value second, how come? yield makes the generator function pause the execution and send back the result to the caller. So when we call generatorSequence for the first time, the function sees the yield with value first, so it put the function to pause mode and returned back the value (and it remembers where it exactly paused, too). The next time, we call the generatorSequence (using the same instance variable), the generator function resumes from where it left off. Since it paused at the line:

```
yield 'first';
```

for the first time, when we call it for second time (using the same instance variable), we get back the value second. What happens when we call it for the third time? Yeah, we will get back the value third!

This is better explained by looking at Figure 10-1:

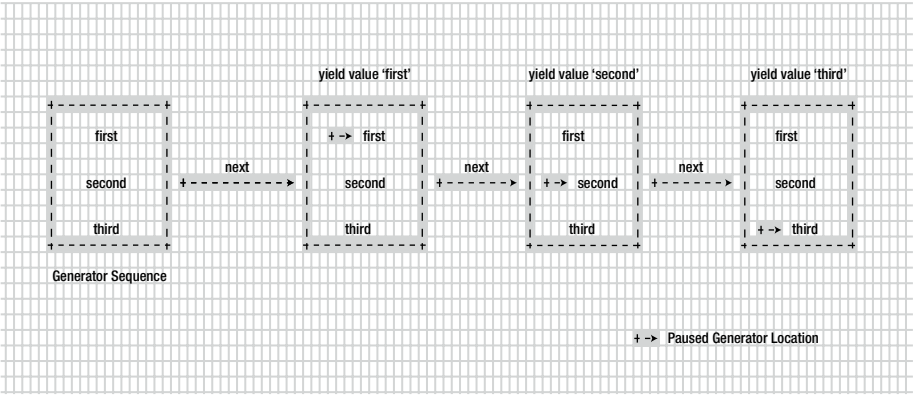

Figure 10-1. *Visual View Of Generator Listed In 10-4*

This sequence is explained via the code here:

Listing 10-6. Calling our generator sequence

```
//get generator instance variable
let generatorSequenceResult = generatorSequence();

console.log('First time sequence value',generatorSequenceResult.next().
value)
console.log('Second time sequence value',generatorSequenceResult.next().
value)
console.log('thrid time sequence value',generatorSequenceResult.next().
value)
```

prints back to the console:

```
First time sequence value first
Second time sequence value second
third time sequence value third
```

With that understanding in place, you can see why we call a generator as a sequence of values! One more important point to keep in mind is that all generators with `yield` will execute in lazy evaluation order.

Lazy evaluation

What is lazy evaluation? To put it in simple terms, lazy evaluation means the code won't run until we ask it to run. As you can guess, the example of `generatorSequence` function shows that Generators are lazy evaluated. The values are being executed and returned only when we ask for them. That's so lazy about Generators, isn't it?

done Property of Generator

Now we have seen how a generator can produce a sequence of values lazily with the `yield` keyword. But a generator can produce `n` numbers of sequence; as a user of the generator function, how will you know when to stop calling `next`? Because calling `next` on your already consumed generator sequence will give back the `undefined` value. How to handle this situation? This is where the `done` property comes into the picture.

Remember that every call to the `next` function is going to return back an object that looks like the following:

```
{value: 'value', done: false}
```

We are aware that the `value` is the value from our generator. But what about `done`? `done` is a property that is going to tell whether the generator sequence has been fully consumed or not.

We will rerun the code from previous sections here (Listing 10-4), just to print the object being returned from the `next` call:

Listing 10-7. Code For Understanding `done` property

```
//get generator instance variable
let generatorSequenceResult = generatorSequence();

console.log('done value for the first time',generatorSequenceResult.next())
console.log('done value for the second time',generatorSequenceResult.next())
console.log('done value for the third time',generatorSequenceResult.next())
```

Running the above code will print the following:

```
done value for the first time { value: 'first', done: false }
done value for the second time { value: 'second', done: false }
done value for the third time { value: 'third', done: false }
```

As you can see we have consumed all the values from the generator sequence, so calling `next` again will return the following object:

```
console.log(generatorSequenceResult.next())
=> { value: undefined, done: true }
```

Now the done property clearly tells us that the generator sequence is already *fully* consumed! When the done is `true`, it's time for us to stop calling `next` on that particular generator instance! Again it can be better visualized with Figure 10-2:

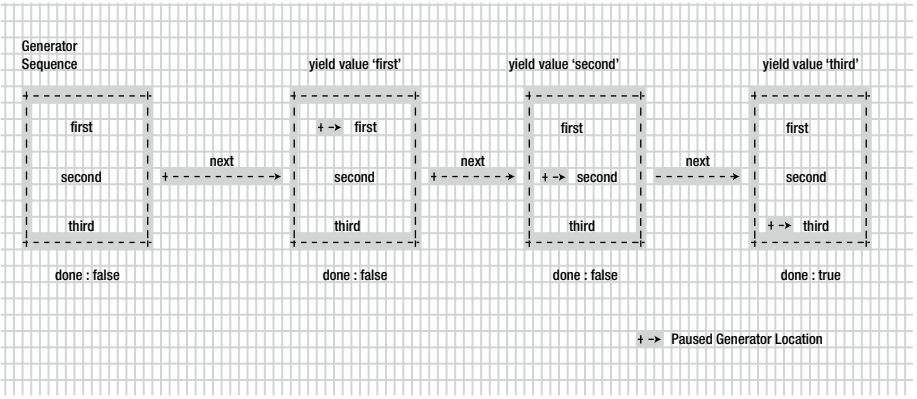

Figure 10-2. *Visual View Of Generators done property for `generatorSequence`*

Since Generator become the core part of ES6, we have a `for` loop that will allow us to iterate a generator (after all it's a sequence. :)

```
for(let value of generatorSequence())
        console.log("for of value of generatorSequence is",value)
```

which is going to print:

```
for of value of generatorSequence is first
for of value of generatorSequence is second
for of value of generatorSequence is third
```

notably `for` using the generator's done property to iterate through it!

Passing Data to Generators

In this section, let's discuss how we pass data to generators. Passing data to generators might feel confusing at first, but as you will see in this chapter, it makes async programming easy!

Let's take a look at the following code snippet:

Listing 10-8. Passing Data Generator Example

```
function* sayFullName() {
    var firstName = yield;
    var secondName = yield;
    console.log(firstName + secondName);
}
```

The code snippet now might be not a surprise for you. Let's use this code to explain the concept of passing data to the generator. As always, we create a generator instance first:

```
let fullName = sayFullName()
```

Once the generator instance is created, let's call `next` on it:

```
fullName.next()
fullName.next('anto')
fullName.next('aravinth')
=> anto aravinth
```

In the above code snippet the last call will print `anto aravinth` to the console! You might be confused with this result, so let's walk over the code slowly. When we call the `next` for the first time:

```
fullName.next()
```

the code will return and pause at the line:

```
var firstName = yield;
```

Since here we are not sending any value back via `yield`, `next` will return a value `undefined`. The second call to `next` is where an interesting thing happens:

```
fullName.next('anto')
```

Here we are passing value `anto` to the `next` call! Now the generator will be resumed from its previous paused state, if you remember the previous paused state is on the line:

```
var firstName = yield;
```

Since we have passed value `anto` on this call, `yield` will be replaced by `anto` and thus `firstName` holds the value `anto`. After the value is being set to `firstName`, the execution will be resumed (from the previous paused state) and again sees the `yield` and stops the execution at:

```
var secondName = yield;
```

Now for the third time, if we call `next`:

```
fullName.next('aravinth')
```

When this line gets executed, our generator will resume from where it paused. The previous paused state is:

```
var secondName = yield;
```

As before, the passed value `aravinth` of our `next` call will be replaced by `yield` and `aravinth` is set to `secondName`. Then the generator happily resumes the execution and sees the statement:

```
console.log(firstName + secondName);
```

By now, `firstName` is `anto` and `secondName` is `aravinth`, so the `console` will print `anto aravinth`.

This full process is explained in Figure 10-3:

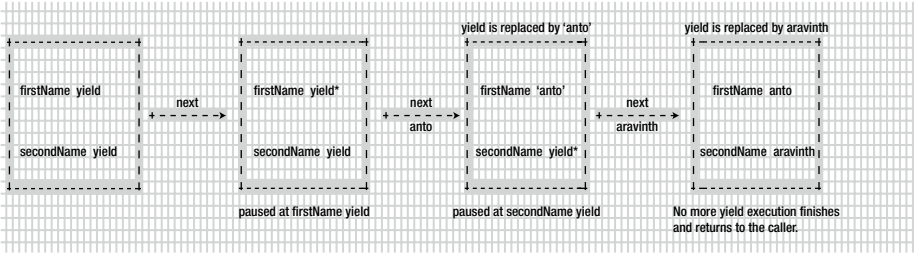

Figure 10-3. *Explaining How Data Is Passed To sayFullName Generator*

You might be wondering why we need such an approach. It turns out that using generators by passing data to them makes it very powerful. We will be using the same technique in the next section to handle async calls!

Using Generators to Handle Async Calls

In this section, we are going to use generators in real-world stuff. We are going to see how passing data to generators make them very powerful to handle async calls. We are going to have quite a lot of fun in this section!

Generators for Async - A Simple Case

In this section, we are going to see how to use generators for handling async code. Since we are getting started with a different mindset of using generators to solve the async problem, I want it to keep the section simple. So we will be mimicking the async calls with `setTimeout` calls!

Imagine you have two functions (which are async in nature):

Listing 10-9. Simple Asynchronous Functions

```
let getDataOne = (cb) => {
        setTimeout(function(){
        //calling the callback
        cb('dummy data one')
    }, 1000);
}
```

```
let getDataTwo = (cb) => {
        setTimeout(function(){
        //calling the callback
        cb('dummy data two')
    }, 1000);
}
```

Both of the above functions mimics the async code with `setTimeout`. Once the desired time has elapsed, `setTimeout` will call the passed callback `cb` with value `dummy data one` and `dummy data two` respectively. Let's see how we will be calling these two functions without generators in the first place:

```
getDataOne((data) => console.log("data received",data))
getDataTwo((data) => console.log("data received",data))
```

The above code will print:

```
data received dummy data one
data received dummy data two
```

after `1000ms`.

Now as you notice, we are passing the callbacks to get back the response. We have talked about how bad the Callback Hell can be in async code. Let's use our generator knowledge to solve the current problem. We now change both the functions `getDataOne` and `getDataTwo` to use generator instances rather than callbacks for passing the data.

First let's change the function `getDataOne` (Listing 10-8) to:

Listing 10-10. Changing `getDataOne` to use generator

```
let generator;
let getDataOne = () => {
        setTimeout(function(){
        //call the generator and
        //pass data via next
        generator.next('dummy data one')
    }, 1000);
}
```

We have changed the callback line from:

```
. . .
cb('dummy data one')
. . .
```

to

```
generator.next('dummy data one')
```

That's a simple change. Note that we have also removed the `cb`, which is not required in this case. We will do the same for `getDataTwo` (Listing 10-8) too:

Listing 10-11. Changing `getDataTwo` to use generator

```
let getDataTwo = () => {
        setTimeout(function(){
        //call the generator and
        //pass data via next
        generator.next('dummy data two')
    }, 1000);
}
```

Now with that change in place, let's go and test our new code. We'll wrap our call to `getDataOne` and `getDataTwo` inside a separate generator function:

Listing 10-12. `main` generator function

```
function* main() {
    let dataOne = yield getDataOne();
    let dataTwo = yield getDataTwo();
    console.log("data one",dataOne)
    console.log("data two",dataTwo)
}
```

Now the `main` code looks exactly like `sayFullName` function from our previous section. Let's create a generator instance for `main` and trigger the `next` call and see what happens:

```
generator = main()
generator.next();
```

which will print the following to the console:

```
data one dummy data one
data two dummy data two
```

which is what exactly we wanted. Look at our `main` code; the code looks like synchronous calls to the function `getDataOne` and `getDataTwo`. However both these calls are asynchronous. Remember that these calls never block and they work in async fashion! Let's distill how this whole process works.

First we are creating a generator instance for `main` using the `generator` variable that we declared earlier. Remember that this `generator` is used by both `getDataOne` and `getDataTwo` to push the data to its call, which we will see soon. After creating the instance, we are firing the whole process with the line:

```
generator.next()
```

This calls the `main` function. `main` function is put into execution and we see the first line with `yield`:

```
. . .
let dataOne = yield getDataOne();
. . .
```

Now the generator will be put into pause mode as it has seen a `yield` statement. But before it's been put into pause mode, it calls the function `getDataOne`.

An important point to note here is that even though the `yield` makes the statement pause, it won't make the caller wait (i.e., caller is not blocked). To make the point more concrete, let's see the below code:

```
generator.next() //even though the generator pause for Async codes

console.log("will be printed")
=> will be printed
=> Generator data result is printed
```

The above code shows that even though our `generator.next` causes the Generator function to wait on the `next` call, the caller (the one who is calling the generator) won't be blocked! As you can see above, `console.log` will be printed (showcasing `generator.next` isn't blocked), and then we get the data from the generator once the async operation is done!

Now interestingly `getDataOne` function has the following line in its body:

```
. . .
    generator.next('dummy data one')
. . .
```

As we discussed in a previous section, calling `next` by passing parameter will resume the paused `yield`! And that's exactly what happens here in this case too! Remember that this piece of line is inside `setTimeout`. So this line will get executed only when `1000ms` have elapsed. Still then, the code will be paused at the line:

```
let dataOne = yield getDataOne();
```

One more important point to note here is that while this line is paused, the timeout will be running down from `1000` to `0`. Once it reaches `0`, it is going to execute the line:

```
. . .
    generator.next('dummy data one')
. . .
```

which is going to send back `dummy data one` to our `yield` statement. So the `dataOne` variable becomes `dummy data one`:

```
//after 1000ms dataOne becomes
//`dummy data one`
let dataOne = yield getDataOne();
=> dataOne = `dummy data one`
```

That's a lot of interesting stuff going around! And once `dataOne` is set to `dummy data one` value, the execution will continue to the next line:

```
. . .
let dataTwo = yield getDataTwo();
. . .
```

This line is going to run the same way as the line before! So after the execution of this line, we have `dataOne` and `dataTwo`:

```
dataOne = dummy data one
dataTwo = dummy data two
```

which is what is getting printed to the console at the final statements of the `main` function:

```
. . .
    console.log("data one",dataOne)
    console.log("data two",dataTwo)
. . .
```

The full process is shown in Figure 10-4:

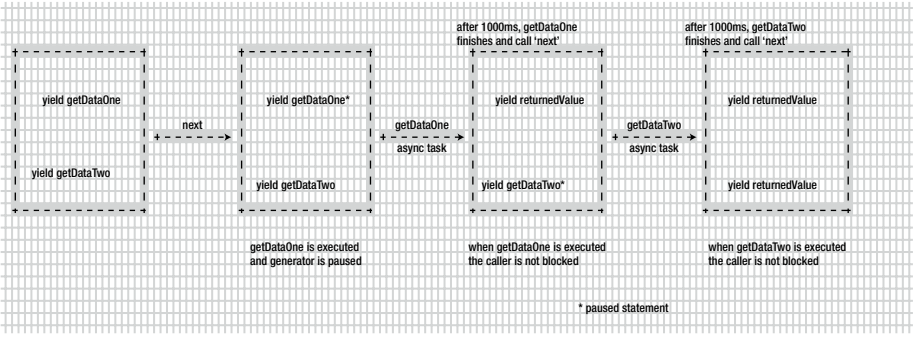

Figure 10-4. *Image Explainging How `main` generators works internally*

Phew! Now you have made an Asynchronous call look like a Synchronous call, but it works in an Asynchronous way!

Generators for Async - A Real-World Case

In the previous section, we saw how to handle asynchronous code using generators effectively. To mimic the async workflow we have used `setTimeout`. In this section, we are going to use a function to fire a real AJAX call to Reddit APIs to showcase the power of generators in the real world!

To make a async call, let's create a function called `httpGetAsync`:

Listing 10-13. `httpGetAsync` function definition

```
let https = require('https');
function httpGetAsync(url,callback) {

    return https.get(url,
        function(response) {
            var body = '';
            response.on('data', function(d) {
                body += d;
            });
            response.on('end', function() {
                let parsed = JSON.parse(body)
                callback(parsed)
            })
        }
    );

}
```

This is a simple function that uses a `https` module from node to fire an AJAX call to get the response back.

Here we are not going to see in detail how `httpGetAsync` function works. The problem we are trying to solve is how to convert functions like `httpGetAsync`, which works the async way but expects a callback to get the response from AJAX calls.

Let's check the `httpGetAsync` by passing a reddit URL:

```
httpGetAsync('https://www.reddit.com/r/pics/.json',(data)=> {
        console.log(data)
})
```

It works by printing the data to the console. The URL https://www.reddit.com/r/pics/.json prints the list of json about Picture Reddit page. The returned json has a data key whose structure looks like the following:

```
{ modhash: '',
  children:
   [ { kind: 't3', data: [Object] },
     { kind: 't3', data: [Object] },
     { kind: 't3', data: [Object] },
     . . .
     { kind: 't3', data: [Object] } ],
  after: 't3_5bzyli',
  before: null }
```

Imagine we want to get the URL of the first children of the array; we need to navigate to data.children[0].data.url. This will give us a URL like https://www.reddit.com/r/pics/comments/5bqai9/introducing_new_rpics_title_guidelines/. Since we need to get the json format of the given URL, we need to append .json to the URL, so that it becomes https://www.reddit.com/r/pics/comments/5bqai9/introducing_new_rpics_title_guidelines/.json.

Now let's see that in action:

```
httpGetAsync('https://www.reddit.com/r/pics/.json',(picJson)=> {
    httpGetAsync(picJson.data.children[0].data.url+".
json",(firstPicRedditData) => {
        console.log(firstPicRedditData)
    })
})
```

The above code will print the data as required. We are least worried about the data being printed. But we are worried about our code structure. As we have seen in the beginning of the chapter, code that looks like this suffers from Callback Hell. Here there are two levels of callbacks that might not be a real problem. But what if it goes to 4-5 nested levels? Can you read such codes easily? Definitely not. Now let's find out how to solve the problem via generator!

Let's wrap httpGetAsync inside a separate method called request:

Listing 10-14. request function

```
function request(url) {
    httpGetAsync( url, function(response){
        generator.next( response );
    } );
}
```

We have removed the callback with the generators' `next` call, very similar to our previous section. Now let's wrap our requirement inside a generator function; again we call it `main`:

Listing 10-15. `main` generator function

```
function *main() {
    let picturesJson = yield request( "https://www.reddit.com/r/pics/.json" );
    let firstPictureData = yield request(picturesJson.data.children[0].data.
    url+".json")
    console.log(firstPictureData)
}
```

The above `main` function looks very similar to the `main` function that we have defined in Listing 10-11 (only change is method call details). In the code we are `yield`ing on two calls to `request`. As you have seen in the `setTimeout` example, calling `yield` on `request` will make it pause until `request` calls the generator `next` by sending the AJAX response back! The first `yield` will get the json of pictures, and the second `yield` gets the first picture data by calling `request` respectively! Now we have made the code look like synchronous code, but in reality it works in asynchronous fashion!

We have also escaped from Callback Hell using generators. Now the code looks clean and clearly tells what it's doing! That's so much more power for us!

Try running it:

```
generator = main()
generator.next()
```

It's going to print the data as required! We have clearly seen how to use generators to convert any function that expects a callback mechanism into a generator based one. In turn, we get back clean code for handling an asynchronous operation.

Summary

The world is full of AJAX calls. There was a time when handling AJAX calls, we needed to pass a callback to process the result. Callbacks have their own limitations. Too many callbacks create Callback Hell problems. We have seen in this chapter a new type in JavaScript called Generator. Generators are functions that can be paused and resumed using the `next` method. The `next` method is available on all generator instances. We have seen how to pass data to generator instances using the `next` method. The technique of sending data to generators helps us to solve the asynchronous code problem. We have seen how to use Generator to make Asynchronous code look synchronous, which is a very powerful technique for any JavaScript developer!

Appendix A

How to Install Node In Your System

1. Navigate to https://nodejs.org/en/download/.
2. Select your operating system and download the installer.
3. Run the installer.
4. Finish the setup.

Installing Depedencies

In node, we will install the dependencies via npm. npm will be part of your installation.

Run the following command to install babel:

```
npm install -g babel
```

Run the following command to install babel-node:

```
npm install -g babel-cli
```

Note that -g will install the scripts globally.

A. Aravinth, *Beginning Functional JavaScript*, DOI 10.1007/978-1-4842-2656-8

Index

A. Aravinth, *Beginning Functional JavaScript*, DOI 10.1007/978-1-4842-2656-8

D

E

F

G

H

I

J, K

L

M

N

O

P, Q

R

S

T

U

V, W, X

Y

Z