



Beginning Functional JavaScript

Uncover the Concepts of Functional
Programming with EcmaScript 8

—
Second Edition
—

Anto Aravinth
Srikanth Machiraju

Apress®

Beginning Functional JavaScript

**Uncover the Concepts of
Functional Programming
with EcmaScript 8**

Second Edition

**Anto Aravinth
Srikanth Machiraju**

Apress®

Beginning Functional JavaScript

Anto Aravinth
Chennai, Tamil Nadu, India

Srikanth Machiraju
Hyderabad, Andhra Pradesh, India

ISBN-13 (pbk): 978-1-4842-4086-1
<https://doi.org/10.1007/978-1-4842-4087-8>

ISBN-13 (electronic): 978-1-4842-4087-8

Library of Congress Control Number: 2018964615

Copyright © Anto Aravinth, Srikanth Machiraju 2018, corrected publication 2020

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, log os, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Nikhil Karakal
Development Editor: Matthew Moodie
Coordinating Editor: Divya Modi

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

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/. For more detailed information, please visit <http://www.apress.com/source-code>.

Printed on acid-free paper

Table of Contents

About the Authors	xi
About the Technical Reviewer	xiii
Acknowledgments	xv
Introduction	xvii
Chapter 1: Functional Programming in Simple Terms	1
What Is Functional Programming? Why Does It Matter?.....	2
Referential Transparency	5
Imperative, Declarative, Abstraction.....	7
Functional Programming Benefits	8
Pure Functions	8
Pure Functions Lead to Testable Code.....	9
Reasonable Code.....	11
Parallel Code	12
Cachable.....	14
Pipelines and Composable	16
A Pure Function Is a Mathematical Function	17
What We Are Going to Build	18
Is JavaScript a Functional Programming Language?.....	19
Summary.....	20

TABLE OF CONTENTS

Chapter 2: Fundamentals of JavaScript Functions.....21

- ECMAScript: A Bit of History.....22
- Creating and Executing Functions23
 - First Function.....24
 - Strict Mode26
 - Return Statement Is Optional28
 - Multiple Statement Functions28
 - Function Arguments30
 - ES5 Functions Are Valid in ES6 and Above30
- Setting Up Our Project.....30
 - Initial Setup31
 - Our First Functional Approach to the Loop Problem.....33
 - Gist on Exports36
 - Gist on Imports36
 - Running the Code Using Babel-Node37
 - Creating Script in Npm38
 - Running the Source Code from Git39
- Summary.....40

Chapter 3: Higher Order Functions41

- Understanding Data42
 - Understanding JavaScript Data Types42
 - Storing a Function43
 - Passing a Function44
 - Returning a Function45

Abstraction and Higher Order Functions	47
Abstraction Definitions	48
Abstraction via Higher Order Functions.....	48
Higher Order Functions in the Real World	53
every Function.....	53
some Function.....	55
sort Function	56
Summary.....	61
Chapter 4: Closures and Higher Order Functions	63
Understanding Closures.....	64
What Are Closures?	64
Remembering Where It Is Born.....	67
Revisiting sortBy Function.....	69
Higher Order Functions in the Real World (Continued).....	70
tap Function.....	70
unary Function.....	72
once Function	74
memoize Function	75
assign function.....	78
Summary.....	80
Chapter 5: Being Functional on Arrays.....	81
Working Functionally on Arrays	82
map	82
filter	87
Chaining Operations.....	88
concatAll.....	90

TABLE OF CONTENTS

Reducing Function	95
reduce Function.....	95
Zipping Arrays	102
zip Function	105
Summary.....	108
Chapter 6: Currying and Partial Application	109
A Few Notes on Terminology.....	110
Unary Function	110
Binary Function	110
Variadic Functions	110
Currying	112
Currying Use Cases	114
A logger Function: Using Currying	116
Revisit Curry	117
Back to logger Function.....	122
Currying in Action.....	123
Finding a Number in Array Contents.....	124
Squaring an Array	124
Data Flow	125
Partial Application.....	125
Implementing partial Function	127
Currying vs. Partial Application	130
Summary.....	130

Chapter 7: Composition and Pipelines.....	133
Composition in General Terms	134
Unix Philosophy	134
Functional Composition	137
Revisiting map,filter	137
compose Function	139
Playing with the compose Function	140
curry and partial to the Rescue	142
compose Many Functions.....	146
Pipelines and Sequence.....	148
Implementing pipe.....	149
Odds on Composition.....	150
The Pipeline Operator	151
Debugging Using the tap Function	154
Summary.....	155
Chapter 8: Fun with Functors	157
What Is a Functor?	158
Functor Is a Container	158
Implementing map	161
Maybe	163
Implementing Maybe.....	163
Simple Use Cases.....	165
Real-World Use Cases	168

TABLE OF CONTENTS

- Either Functor 173
 - Implementing Either 174
 - Reddit Example Either Version..... 176
- Word of Caution: Pointed Functor 179
- Summary..... 180
- Chapter 9: Monads in Depth 181**
 - Getting Reddit Comments for Our Search Query..... 182
 - The Problem..... 183
 - Implementation of the First Step 185
 - Merging Reddit Calls 189
 - Problem of Nested/Many maps 194
 - Solving the Problem via join 196
 - join Implementation..... 196
 - chain Implementation..... 200
 - Summary..... 203
- Chapter 10: Pause, Resume, and Async with Generators 205**
 - Async Code and Its Problem 206
 - Callback Hell..... 206
 - Generators 101 209
 - Creating Generators..... 209
 - Caveats of Generators 210
 - yield Keyword 211
 - done Property of Generator 214
 - Passing Data to Generators 216

Using Generators to Handle Async Calls	219
Generators for Async: A Simple Case	219
Generators for Async: A Real-World Case.....	226
Async Functions in ECMAScript 2017	230
Promise	230
Await.....	231
Async.....	231
Chaining Callbacks	233
Error Handling in Async Calls	236
Async Functions Transpiled to Generators	237
Summary.....	239
Chapter 11: Building a React-Like Library.....	241
Immutability	242
Building a Simple Redux Library.....	245
Building a Framework Like HyperApp.....	251
Virtual DOM.....	252
JSX	254
JS Fiddle.....	255
CreateActions	261
Render	262
Patch	263
Update	264
Merge	265
Remove.....	266
Summary.....	268

TABLE OF CONTENTS

Chapter 12: Testing and Closing Thoughts269

- Introduction..... 270
- Types of Testing 272
- BDD and TDD..... 273
- JavaScript Test Frameworks..... 274
 - Testing Using Mocha 275
 - Mocking Using Sinon..... 283
 - Testing with Jasmine..... 287
- Code Coverage 290
- Linting 291
- Unit Testing Library Code 294
- Closing Thoughts..... 296
- Summary..... 297

Correction to: Fun with Functors C1

Index.....299

About the Authors



Anto Aravinth has been in the software industry for more than six years. He has developed many systems that are written in the latest technologies. Anto has knowledge of the fundamentals of JavaScript and how it works and has trained many people. Anto is also does OSS in his free time and loves to play table tennis.



Srikanth Machiraju has over ten years of experience as a developer, architect, technical trainer, and community speaker. He is currently working as Senior Consultant with Microsoft Hyderabad, leading a team of 100 developers and quality analysts developing an advanced cloud-based platform for a tech giant in the oil industry. With an aim to be an enterprise architect who can design hyperscale modern applications with intelligence, he constantly learns and shares modern application development tactics using cutting-edge platforms and technologies. Prior to Microsoft, he worked with BrainScale as Corporate Trainer and Senior Technical Analyst on application design, development,

ABOUT THE AUTHORS

and migrations using Azure. He is a tech-savvy developer who is passionate about embracing new technologies and sharing his learning via blog and community engagements. He has also authored the “Learning Windows Server Containers” and “Developing Bots with Microsoft Bot Framework,” blogs at <https://vishwanathsrikanth.wordpress.com>. He runs his own YouTube channel called “Tech Talk with Sriks” and is active on LinkedIn at <https://www.linkedin.com/in/vishsrik/>.

About the Technical Reviewer



Sakib Shaikh has been working as a Tech Lead with a large scientific publisher, with more than ten years of experience as a full stack developer with JavaScript technologies on front-end and back-end systems. He has been reviewing technical books and articles for the past few years and contributes to the developer community as a trainer, blogger, and mentor.

Acknowledgments

I remember the first code that I wrote for Juspay Systems in my first job as an intern. Coding was fun for me; at times it is challenging, too. Now with six years of software experience, I want to make sure I pass on all the knowledge I have to the community. I love teaching people. I love to share my thoughts with the community to get feedback. That's exactly the reason I'm writing a second edition of this book.

I have to acknowledge few people who have been standing right next to me in all phases of my life: my late father Belgin Rayen, mother Susila, Kishore (brother-in-law), Ramya (sibling), and Joshuwa (my new little nephew). They have been supportive and pushed me harder to achieve my goals. I want to say thanks to Divya and the technical reviewer of this book, as they did a wonderful job. Luckily, I have a wonderful coauthor in Srikanth, who did an amazing job as well.

Finally, I want to give special thanks to Bianaca, Deepak, Vishal, Arun, Vishwapriya, and Shabala, who have added joy to my life.

Please reach out to me at anto.aravinth.cse@gmail.com with any feedback.

—Anto Aravinth

I would like to thank Apress for providing me a second opportunity to author. I would also like to thank my family, especially my dear wife Sonia Madan and my four-month-old son Reyansh for supporting me throughout this stint. I'm always reachable at Vishwanath.srikanth@gmail.com for any feedback or questions.

—Srikanth Machiraju

Introduction

The second edition of a book is always special. When I wrote the first edition, I had about two years of IT experience. The book received positive as well as negative responses. I always wanted to work on the negative responses to make the content better and make the book worth the price. In the meantime, JavaScript evolved a great deal. Many ground-breaking changes were added into the language. The Web is full of JavaScript, and imagine a world without the Web. Hard, right?

This second edition is a much improved version that teaches the fundamentals of functional programming in JavaScript. We have added much new content in this second edition; for example, we will be building a library for building web applications using functional concepts, and we have added sections on testing as well. We have rewritten the book to match the latest ES8 syntax with many samples of `async`, `await` patterns, and a lot more!

We assure you that you will gain a lot of knowledge from this book and at the same time you will have fun while running the examples. Start reading.

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, a world that has only functions, living happily without any outside world dependencies, without states, and without mutations—forever. Functional programming is a buzzword these days. You might have heard about this term within your team or in a local group meeting. If you're already aware of what that means, great. For those who don't know the term, don't worry. This chapter is designed 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? Later, 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 provides to developers.

What Is Functional Programming? Why Does It Matter?

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

$$f(X) = Y$$

The statement can be read as “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. There are key takeaways in the definition, though:

- 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 presented the definition of function in mathematics rather than in JavaScript. Did you? That’s a great question. The answer is pretty simple: Functional programming techniques are heavily based on mathematical functions and their ideas. Hold your breath, though; we are not going to teach you functional programming in mathematics, but rather use JavaScript. Throughout the book, however, we will be seeing the ideas of mathematical functions and how they are used to help understand functional programming.

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 tax calculations. How are you going to do this in JavaScript? We can implement such a function as shown in Listing 1-1.

Listing 1-1. Calculate Tax Function

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

The `calculateTax` function 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 this 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 on the outside world. In our `calculateTax` function, we have made the function depend on the *global* variable `percentValue`. Thus this function we have created can't be called as a real function in a mathematical sense. Let's fix that.

The fix is very straightforward: We have to just move the `percentValue` as our function argument, as shown in Listing 1-2.

Listing 1-2. Rewritten `calculateTax` Function

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

Now our `calculateTax` function can be called as a real function. What have we gained, though? We have just eliminated 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 later in this chapter.)

Now we have shown the relationship between the math function and 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 their logic by depending only on their 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 a cachable and testable code base.

FUNCTIONS VS. METHODS IN JAVASCRIPT

We have talked about the word *function* a lot in this text. Before we move on, we 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 used to pass arguments that it can operate on and return values optionally.

A *method* is a piece of code that must be called by its name that is associated with an object.

Listing 1-3 and Listing 1-4 provide quick examples of a function and a method.

Listing 1-3. A Simple Function

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

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 programming that are missing in the definition. We discuss them in detail in the upcoming sections before we dive into the benefits of functional programming.

Referential Transparency

With our definition of function, we have made a statement that all the functions are going to return the same value for the same input. This property of a function is called a *referential transparency*. A simple example is shown in Listing 1-5.

Listing 1-5. Referential Transparency Example

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

In Listing 1-5, 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 the value 5 (i.e., the function just acts as a mirror or identity). Note that our function operates 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 a 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 that 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 given function with multiple threads without even the need to synchronize. Why? The reason for synchronizing comes from the fact that threads shouldn't act on global data when running parallel.

Functions that obey referential transparency are going to depend only on inputs from their argument; hence threads are free to run without any locking mechanism.

Because 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 know the factorial of 5 is 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 `factorial` is called for a second time with the input as 5, we can return the cached value instead of calculating it 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 PHILOSOPHY

Referential transparency is a term 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 the thing to which the expression refers. A context in a sentence is referentially transparent if replacing a term in that context with another term that refers to the same entity doesn't alter the meaning.

That's exactly how we have been defining referential transparency here. We have replaced the value of the function without affecting the context.

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 an 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 might look like Listing 1-6.

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. In this approach to solve our problem, though, 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? It looks like we are telling the compiler what to do, however. In this case, we are telling the compiler, “Get array length, loop our array, get each element of the array using the index, and so on.” We call it an imperative solution. *Imperative* programming is all about telling the compiler how to do 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 how. The “how” parts are abstracted into common functions (these functions are called higher order functions, which we cover in the upcoming chapters). Now we can use the built-in `forEach` function to iterate the array and print it, as shown in Listing 1-7.

Listing 1-7. Iterating over the Array Declarative Approach

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

Listing 1-7 does print exactly the same output as Listing 1-5. Here, though, we have removed the “how” parts like “Get array length, loop our array, get each element of an array using an index, and so on.” We have used an abstracted function, which takes care of the “how” part, leaving us, the developers, to worry about our problem at hand (the “what” part). We will be creating these built-in functions throughout the book.

Functional programming is about creating functions in an abstracted way that can be reused by other parts of the code. Now we have a solid understanding of what functional programming is; with this in mind, we can 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 JavaScript. We now have to answer a simple question: What are the benefits of functional programming? This section helps you see the huge benefits that functional programming offers us. Most of the benefits of functional programming come from writing pure functions. So before we see the benefits of functional programming, we need to know 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 the example in Listing 1-8.

Listing 1-8. A Simple Pure Function

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

This function `double` is a pure function because given an input, it is always going to return the same output. You can try it yourself. Calling the `double` function with input 5 always gives 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? They provide many benefits, which we discuss next.

Pure Functions Lead to Testable Code

Functions that are not pure have side effects. Take our previous tax calculation example from 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. The function works, but it is very difficult 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 passed. However, because our original `calculateTax` function depends on the external environment variable `percentValue`, things can go wrong. Imagine the external environment is changing the `percentValue` variable 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. We can easily fix the issue, though, 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 this function without any pain. Before we close this section, we need to mention an important property about pure functions: Pure functions also shouldn't mutate any external environment variables. In other words, the pure function shouldn't depend on any external variables (as shown in the example) and also change any external variables. We'll now take a quick look what we mean by changing any external variables. For example, consider the code in Listing 1-9.

Listing 1-9. `badFunction` Example

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

When the `badFunction` function is called it changes the global variable `global` to the value changed. Is it something to worry about? Yes. Imagine another function that depends on the `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 the case of debugging.

So we have seen with a simple example how a pure function can help us in easily 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
```

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 reading others' code. Having a function with side effects in your code base makes it hard for other developers in your team to read. Code bases with pure functions are easy to read, understand, and test. Remember that a function (regardless of whether it is a pure function) must always have a meaningful name. For example, you can't name the function `double` as `dd` given what it does.

SMALL MIND GAME

We are just replacing the function with a 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 a pure function. Now have a cup of coffee; you have done a great job!

Parallel Code

Pure functions allow us to run the code in parallel. As a pure function is not going to change any of its environments, this means we do not need to worry about *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 a node environment that runs the function in parallel?

For example, imagine we have the code 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 and thread two (T-2) picks `function2` to run. Now both threads are ready to run and here comes the problem. What if T-1 runs before T-2? Because both `function1` and `function2` depend on the 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 the `global` variable as arguments for both the functions, making them pure. Now we can run both functions in parallel without any issues. Because 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 functions help our code to run in parallel without any problems.

Cachable

Because 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 provide a simple example. Imagine we have a function that does time-consuming calculations. We name this function `longRunningFunction`:

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

If the `longRunningFunction` function is a pure function, then we know that for the given input, it is going to return the same output. 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