Mastering JavaScript Design Patterns

Third Edition

Create scalable and reliable applications with advanced JavaScript design patterns using reliable code

By Tomás Corral Casas

Packt
www.packt.com

What this book covers

`Chapter 1`, *Building Your Foundations*, this chapter is the introduction to the book. It will cover, in general, basic history of Javascript, programming paradigms and design patterns and how the book is arranged. By the end of this chapter, readers should be able to explain where JavaScript is used and the history and applicability of design patterns. They should also have an understanding of how to read the book.

`Chapter 2`, *Object Oriented Programming*, we'll talk about how to create classes and modules in JavaScript. Most of the patterns presented are related to classes so it is important to have some grasp of how to create classes using ES2015/2017/2018.

`Chapter 3`, *Functional Programming*, in this chapter, we'll take a look at how some functional paradigms can be used in JavaScript using ES2015/2017/2018.

`Chapter 4`, *Reactive Programming*, reactive and Data Flow applications view the world as a stream of events. Working with these streams instead of discrete interactions presents a different worldview and one which may be very useful for large-scale JavaScript applications using ES2015/2017/2018.

`Chapter 5`, *Creational Patterns*, in this chapter, we'll look at the classic creational design patterns that were popularized by the GoF book using ES2015/2017/2018.

`Chapter 6`, *Structural Patterns*, in this chapter we'll look at some of the ways code can be structured. Again we'll pay attention to the classic design patterns as described in the GoF book. Structural Patterns using ES2015/2017/2018.

`Chapter 7`, *Behavioural Patterns*, in this chapter, we'll pay attention to the classic behavioural design patterns as described in the GoF book using ES2015/2017/2018.

`Chapter 8`, *Performance Patterns*, in this chapter, we'll look at those patterns that will improve the performance of highly demanding applications using ES2015/2017/2018.

`Chapter 9`, *Asynchronous Patterns*, in this chapter, we'll explore the concept of asynchronous programming patterns and how to use them to do not block our application using ES2015/2017/2018.

`Chapter 10`, *Patterns for Testing*, JavaScript is a real language now so we need to make sure that we're doing proper testing of it. In this chapter, we'll discuss some of the ways our code can be structured to perform testing.

`Chapter 11`, *Advanced Patterns*, this chapter will cover some advanced patterns in JavaScript. Most of them are implemented as wild language hacks.

`Chapter 12`, *Application Patterns*, in this chapter, we'll examine some of the patterns for building applications in JavaScript using ES2015/2017/2018.

`Chapter 13`, *Web Patterns*, in this chapter, we'll look at patterns that are specifically useful for building web applications.

`Chapter 14`, *Messaging Patterns*, in this chapter, we'll explore the concept of messaging and how it can be used in JavaScript applications using ES2015/2017/2018.

`Chapter 15`, *Micro-services*, monolithic applications have become larger and more complicated. There are many possible remedies but the one which has been making a lot of noise recently is micro-services. In this chapter, we'll discuss what micro-services are, where to use them where not to use them.

`Chapter 16`, *ES2015/2017/2018 Solutions Today and the Road ahead*, in this chapter, we'll look at how we can get some of the features of the upcoming versions of JavaScript today.

`Chapter 17`, *ES2019 What is ESNEXT*, in this chapter, we'll look at what are the features that will be introduced in ES2019 a.k.an ESNext.

Table of Contents

Prototype

- When to use it?
- How to implement it?
- Example
- When not to use it?
- Pros
- Cons

Constructor

- When to use it?
- Example
- When not to use it?

Factory method

- When to use it?
- How to implement it?
- Example
- When not to use it?
- Pros
- Cons

Abstract Factory

- When to use it?

Adapter

- When to use it?
- How to implement it?
- Example
- When not to use it?
- Pros
- Cons

Composite

- When to use it?
- How to implement it?
- Example
- When not to use it?
- Pros
- Cons

Decorator

- When to use it?
- How to implement it?
- Example
- When not to use it?
- Pros
- Cons

Bridge

- When to use it?
- How to implement it?
- Example
- When not to use it?
- Pros
- Cons

Facade

- When to use it?
- How to implement it?
- Example
- When not to use it?
- Pros
- Cons

Flyweight

- When to use it?
- How to implement it?
- Example
- When not to use it?
- Pros
- Cons

Building your foundations

Introduction

JavaScript is an evolving language that has come a long way from its inception. Possibly more than any other programming language, it has grown and changed with the growth of the World Wide Web. The exploration of how JavaScript can be written using good design principles is the topic of this book. The preface of this book contains a detailed explanation of the sections of the book.

In the first half of this chapter, we'll explore the history of JavaScript and how it came to be the important language that it is today. As JavaScript has evolved and grown in importance, the need to apply rigorous methods to its construction has also grown. Design patterns can be a very useful tool to assist in developing maintainable code. The second half of the chapter will be dedicated to the theory of design patterns. Finally, we'll look briefly at anti-patterns.

The road to JavaScript

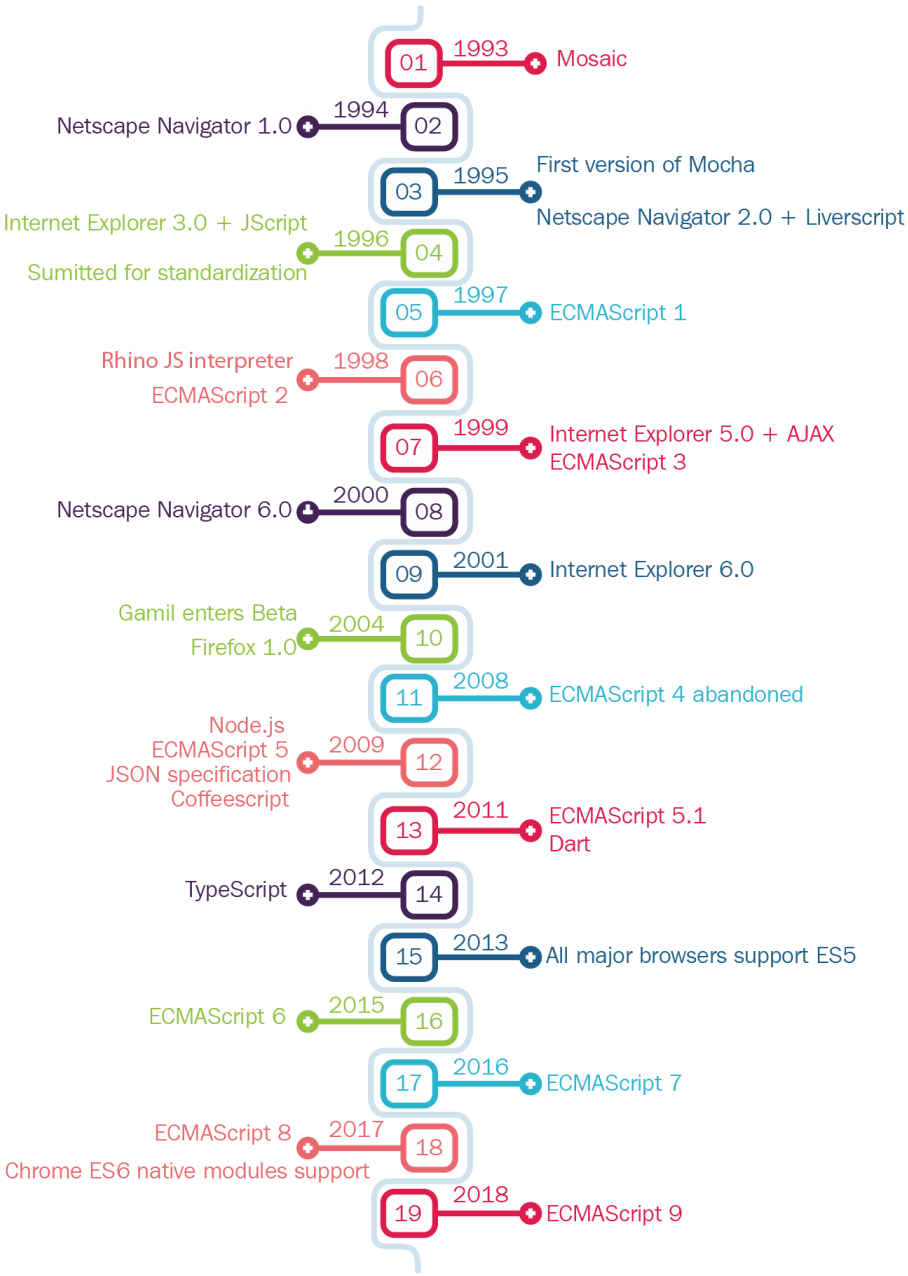
We'll never know how language first came into being. Did it slowly evolve from a series of grunts and guttural sounds made during grooming rituals? Perhaps it developed to allow mothers and their offspring to communicate. Both of these are theories, all but impossible to prove. Nobody was around to observe our ancestors during that important period. In fact, the general lack of empirical evidence led the Linguistic Society of Paris to ban further discussions on the topic, seeing it as unsuitable for serious study.

The early days

Fortunately, programming languages have developed in recent history and we've been able to watch them grow and change. JavaScript has one of the more interesting histories of modern programming languages. During what must have been an absolutely frantic 10 days in May of 1995, a programmer at Netscape wrote the foundation for what would grow up to be modern JavaScript.

At the time, Netscape was involved in the first of the browser wars with Microsoft. The vision for Netscape was far grander than simply developing a browser. They wanted to create an entire distributed operating system making use of Sun Microsystems' recently-released Java programming language. Java was a much more modern alternative to the C++ Microsoft was pushing. However, Netscape didn't have an answer to Visual Basic. Visual Basic was an easier to use programming language, which was targeted at developers with less experience. It avoided some of the difficulties around memory management that make C and C++ notoriously difficult to program. Visual Basic also avoided strict typing and overall allowed more leeway.

Here is an illustration of the timeline of JavaScript:



Brendan Eich was tasked with developing Netscape repartee to VB. The project was initially codenamed Mocha, but was renamed LiveScript before Netscape 2.0 beta was released. By the time the full release was available, Mocha/LiveScript had been renamed JavaScript to tie it into the Java applet integration. Java Applets were small applications which ran in the browser. They had a different security model from the browser itself and so were limited in how they could interact

with both the browser and the local system. It is quite rare to see applets these days, as much of their functionality has become part of the browser. Java was riding a popular wave at the time and any relationship to it was played up.

The name has caused much confusion over the years. JavaScript is a very different language from Java. JavaScript is an interpreted language with loose typing, which runs primarily on the browser. Java is a language that is compiled to bytecode, which is then executed on the Java Virtual Machine. It has applicability in numerous scenarios, from the browser (through the use of Java applets) to the server (Tomcat, JBoss, and so on), to full desktop applications (Eclipse, OpenOffice, and so on). In most laypersons' minds, the confusion remains.

JavaScript turned out to be really quite useful for interacting with the web browser. It was not long until Microsoft had also adopted JavaScript into their Internet Explorer to complement VBScript. The Microsoft implementation was known as JScript.

By late 1996, it was clear that JavaScript was going to be the winning web language for the near future. In order to limit the amount of language deviation between implementations, Sun and Netscape began working with the European Computer Manufacturers Association (ECMA) to develop a standard to which future versions of JavaScript would need to comply. The standard was released very quickly (very quickly in terms of how rapidly standards organizations move), in July of 1997. On the off chance that you have not seen enough names yet for JavaScript, the standard version was called ECMAScript, a name which still persists in some circles.

Unfortunately, the standard only specified the very core parts of JavaScript. With the browser wars raging, it was apparent that any vendor that stuck with only the basic implementation

of JavaScript would quickly be left behind. At the same time, there was much work going on to establish a standard Document Object Model (DOM) for browsers. The DOM was, in effect, an API for a web page that could be manipulated using JavaScript.

> *DHTML was a popular term in the late 1990s and early 2000s. It really referred to any web page that had some sort of dynamic content that was executed on the client side. It has fallen out of use, as the popularity of JavaScript has made almost every page a dynamic one.*

Fortunately, the efforts to standardize JavaScript continued behind the scenes. Versions 2 and 3 of ECMAScript were released in 1998 and 1999. It looked like there might finally be some agreement between the various parties interested in JavaScript. Work began in early 2000 on ECMAScript 4, which was to be a major new release.

A pause

Then, disaster struck. The various groups involved in the ECMAScript effort had major disagreements about the direction JavaScript was to take. Microsoft seemed to have lost interest in the standardization effort. It was somewhat understandable, as it was around that time that Netscape self-destructed and Internet Explorer became the de-facto standard. Microsoft implemented parts of ECMAScript 4 but not all of it. Others implemented more fully-featured support, but without the market leader on-board, developers didn't bother using them.

Years passed without consensus and without a new release of ECMAScript. However, as frequently happens, the evolution of the Internet could not be stopped by a lack of agreement between major players. Libraries such as jQuery, Prototype, Dojo, and Mootools, papered over the major differences in browsers, making cross-browser development far easier. At the same time, the amount of JavaScript used in applications increased dramatically.

The way of Gmail

The turning point was, perhaps, the release of Google's GMail application in 2004. Although XMLHttpRequest, the technology behind Asynchronous JavaScript and XML (AJAX), had been around for about five years when GMail was released, it had not been well-used. When GMail was released, I was totally knocked off my feet by how smooth it was. We've grown used to applications that avoid full reloads, but at the time, it was a revolution. To make applications like that work, a great deal of JavaScript is needed.

> *AJAX is a method by which small chunks of data are retrieved from the server by a client instead of refreshing the entire page. The technology allows for more interactive pages that avoid the jolt of full page reloads.*

The popularity of GMail was the trigger for a change that had been brewing for a while. Increasing JavaScript acceptance and standardization pushed us past the tipping point for the acceptance of JavaScript as a proper language. Up until that point, much of the use of JavaScript was for performing minor changes to the page and for validating form input. I joke with people that, in the early days of JavaScript, the only function name which was used was Validate().

Applications such as GMail that have a heavy reliance on AJAX and avoid full page reloads are known as Single Page Applications or SPAs. By minimizing the changes to the page contents, users have a more uid experience. By transferring only a JavaScript Object Notation (JSON) payload instead of HTML, the amount of bandwidth required is also minimized. This makes applications appear to be snappier. In recent years,

there have been great advances in frameworks that ease the creation of SPAs. AngularJS, Backbone.js, and ember are all Model View Controller style frameworks. They have gained great popularity in the past two to three years and provide some interesting use of patterns. These frameworks are the evolution of years of experimentation with JavaScript best practices by some very smart people.

> *JSON is a human-readable serialization format for JavaScript. It has become very popular in recent years, as it is easier and less cumbersome than previously popular formats such as XML. It lacks many of the companion technologies and strict grammatical rules of XML, but makes up for it in simplicity.*

At the same time as the frameworks using JavaScript are evolving, the language is too. 2015 saw the release of a much-vaunted new version of JavaScript that had been under development for some years. Initially called ECMAScript 6, the final name ended up being ECMAScript-2015. It brought with it some great improvements to the ecosystem. Browser vendors are rushing to adopt the standard. Because of the complexity of adding new language features to the code base, coupled with the fact that not everybody is on the cutting edge of browsers, a number of other languages that transcompile to JavaScript are gaining popularity. CoffeeScript is a Python-like language that strives to improve the readability and brevity of JavaScript. Developed by Google, Dart is being pushed by Google as an eventual replacement for JavaScript. Its construction addresses some of the optimizations that are impossible in traditional JavaScript. Until a Dart runtime is sufficiently popular, Google provides a Dart to the JavaScript transcompiler. TypeScript is a Microsoft project that adds some ECMAScript-2015 and even some ECMAScript-201X syntax, as well as an interesting typing system, to JavaScript. It aims to address some of the issues that large JavaScript projects present.

The point of this discussion about the history of JavaScript is twofold: first, it is important to remember that languages do not develop in a vacuum. Both human languages and computer programming languages mutate based on the environments in which they are used. It is a popularly held belief that the Inuit people have a great number of words for "snow", as it was so prevalent in their environment. This may or may not be true, depending on your definition for the word and exactly who makes up the Inuit people. There are, however, a great number of examples of domain-specific lexicons evolving to meet the requirements for exact definitions in narrow fields. One need look no further than a specialty cooking store to see the great number of variants of items which a layperson such as myself would call a pan.

The Sapir–Whorf hypothesis is a hypothesis within the linguistics domain, which suggests that not only is language influenced by the environment in which it is used, but also that language influences its environment. Also known as linguistic relativity, the theory is that one's cognitive processes differ based on how the language is constructed. Cognitive psychologist Keith Chen has proposed a fascinating example of this. In a very highly-viewed TED talk, Dr. Chen suggested that there is a strong positive correlation between languages that lack a future tense and those that have high savings rates (`https://www.ted.com/talks/keith_chen_could_your_language_affect_your_ability_to_save_money/transcript`). The hypothesis at which Dr. Chen arrived is that when your language does not have a strong sense of connection between the present and the future, this leads to more reckless behavior in the present.

Thus, understanding the history of JavaScript puts one in a better position to understand how and where to make use of JavaScript.

The second reason I explored the history of JavaScript because it is absolutely fascinating to see how quickly such a popular tool has evolved. At the time of writing, it has been about 20 years since JavaScript was first built and its rise to popularity has been explosive. What more exciting thing is there than to work in an ever-evolving language?

Javascript everywhere

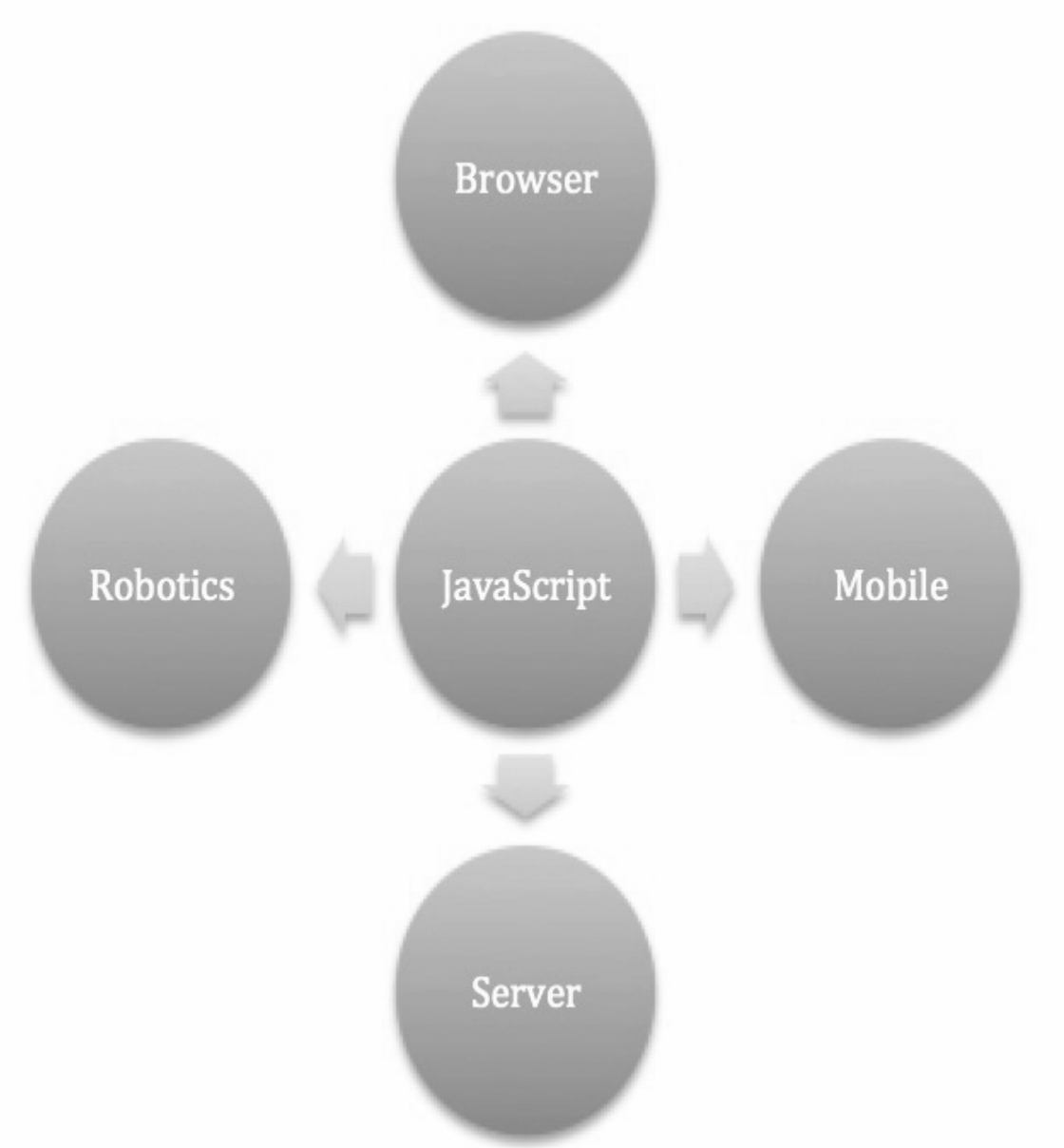
Since the GMail revolution, JavaScript has grown immensely. The renewed browser wars, which pit Internet Explorer and Edge against Chrome and against Firefox, have lead to building a number of very fast JavaScript interpreters. Brand new optimization techniques have been deployed and it is not unusual to see JavaScript compiled to machine-native code for the added performance it gains. However, as the speed of JavaScript has increased, so has the complexity of the applications built using it.

JavaScript is no longer simply a language for manipulating the browser, either. The JavaScript engine behind the popular Chrome browser has been extracted and is now at the heart of a number of interesting projects such as Rhino or Node.js. Rhino was the first Javascript interpreter created in Java. Node.js started off as a highly asynchronous method of writing server-side applications and it was built in C and C++. It has grown greatly and has a very active community supporting it. A wide variety of applications have been built using the Node.js runtime. Everything from build tools to editors has been built on the base of Node.js. Recently, the JavaScript engine for Microsoft Edge, ChakraCore, was also open sourced and can be embedded in Node.js as an alternative to Google's V8.

SpiderMonkey, the Firefox equivalent, is also open source and is making its way into more tools.

JavaScript can even be used to control microcontrollers. The Johnny-Five framework is a programming framework for the very popular Arduino. It brings a much simpler approach to programming devices than the traditional low-level languages used for programming these devices. Using JavaScript and Arduino opens up a world of possibilities, from building robots to interacting with real-world sensors.

All of the major smartphone platforms (iOS, Android, and Windows Phone) have an option to build applications using JavaScript. The tablet space is much the same, with tablets supporting programming using JavaScript. Even the latest version of Windows provides a mechanism for building applications using JavaScript. This illustration shows some of the things possible with JavaScript:



JavaScript is one of the most important languages in the world. Although language usage statistics are notoriously difficult to calculate we can get an idea about how many repositories use Javascript as the main language in services like

https://github.com/.

Ranking website	Ranking Position
Tiobe	6
Indeed Jobs Opening	2
Github	1

What is more interesting is that you can see how the usage of JavaScript is on the rise and it didn't stop growing in the last few years.

The long and short of it is that JavaScript is a major language already and more and more applications are being written in JavaScript and it is the lingua franca for any sort of web development. Developer of the popular Stack Over ow website Jeff Atwood created Atwood's Law regarding the wide adoption of JavaScript:

"Any application that can be written in JavaScript, will eventually be written in JavaScript"

–

Atwood's Law, Jeff Atwood

This insight has been proven to be correct time and time again. There are now compilers, spreadsheets, word processors—you name it—all written in JavaScript.

As the applications which make use of JavaScript increase in complexity, the developer may stumble upon many of the same issues as have been encountered in traditional programming languages: how can we write this application to be adaptable to change?

This brings us to the need for properly designing applications. No longer can we simply throw a bunch of JavaScript into ale and hope that it works properly. Nor can we rely on libraries such as jQuery to save ourselves. Libraries can only provide additional functionality and contribute nothing to the structure of an application. At least some attention must now be paid to how to construct the application to be extensible and adaptable. The real world is ever-changing and any application that is unable to change to suit the changing world is likely to be left in the dust. Design patterns provide some guidance in building adaptable applications, which can shift with changing business needs.

What is a design pattern?

For the most part, ideas are only applicable in one place. Adding peanut butter is really only a great idea in cooking and not in sewing. However, from time to time it is possible to find applicability for a great idea outside of its original purpose. This is the story behind design patterns.

In 1977, Christopher Alexander, Sara Ishikawa, and Murray Silverstein authored a seminal book on what they called design patterns in urban planning, called A Pattern Language: Towns, Buildings, Construction.

The book described a language for talking about the commonalities of design. In the book, a pattern is described thusly:

"The elements of this language are entities called patterns. Each pattern describes a problem that occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice." — Christopher Alexander

These design patterns were such things as how to layout cities to provide a mixture of city and country living, or how to build roads in loops as a traffic-calming measure in residential areas, as is shown in the following picture taken from
the book:
Even for those without a strong interest in urban planning, the book presents some fascinating ideas about how to structure

our world to promote healthy societies.Using the work of Christopher Alexander and the other authors as a source of inspiration, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides wrote a book called Design Patterns: Elements of Reusable Object-Oriented Software. When a book is very influential in a computer science curriculum, it is often given a pet name. For instance, most computer science graduates will know of which book you mean if you talk about The Dragon Book (Principles of Compiler Design, 1986). In enterprise software, The Blue Book is well known to be Eric Evan's book on the domain- driven design. The design patterns book has been so important that it is commonly referred to as the GoF book, or Gang of Four book, for its four authors.This book outlined 23 patterns for use in object-oriented design. It is divided the patterns into three major groups:

- **Creational:** These patterns outlined a number of ways in which objects could be created and their lifecycles managed
- **Behavioral:** These patterns describe how objects interact with each other
- **Structural:** These patterns describe a variety of different ways to add functionality to existing objects

The purpose of design patterns is not to instruct you on how to build software, but rather to give guidance on ways in which to solve common problems. For instance, many applications have a need to provide some sort of an undo function. The problem is common to text editors, drawing programs, and even e-mail clients. Solving this problem has been done many times before so it would be great to have a common solution. The command pattern provides just such a common solution. It suggests

keeping track of all the actions performed in an application as instances of a command. This command will have forward and reverse actions. Every time a command is processed it is placed onto a queue. When it comes time to undo a command it is as simple as popping the top command off of the command queue and executing the undo action on it.Design patterns provide some hints about how to solve common problems like the undo problem. They have been distilled from performing hundreds of iterations of solving the same problem. The design pattern may not be exactly the correct solution for the problem you have, but it should, at the very least, provide some guidance to implement a solution more easily.

> *A consultant friend of mine once told me a story about starting an assignment at a new company. The manager told them that he didn't think there would be a lot of work to do with the team because they had bought the GoF design pattern book for the developers early on and they'd implemented every last design pattern. My friend was delighted about hearing this because he charges by the hour.*
> *The misapplication of design patterns paid for much of his first-born's college education.*

Since the GoF book, there has been a great proliferation of literature dealing with enumerating and describing design patterns. There are books on design patterns which are specific to certain domains and books which deal with patterns for large enterprise systems. The Wikipedia category for software design patterns contains 130 entries for different design patterns. I would, however, argue that many of the entries are not true design patterns but rather programming paradigms.For the most part, design patterns are simple constructs that don't need complicated support from libraries. While there do exist pattern libraries for most languages, you need not go out and spend a lot of money to purchase the libraries.Implement the patterns as you find the need. Having an expensive library burning a hole in your pocket encourages blindly applying patterns just to justify having spent the money. Even if you did have the money, I'm not aware of any

libraries for JavaScript whose sole purpose is to provide support for patterns. Of course, GitHub is a wealth of interesting JavaScript projects, so there may well be a library on there of which I'm unaware.

Anti-patterns

If there are common patterns to be found in good software design, are there also patterns that can be found in bad software design? Absolutely! There are a number of ways to do things incorrectly, but most of them have been done before. It takes real creativity to screw up in a hitherto unknown way.The shame of it is that it is very difficult to remember all the ways in which people have gone wrong over the years. At the end of many major projects, the team will sit down and put together a document called Lessons Learned. This document contains a list of things that could have gone better on the project and may even outline some suggestions as to how these issues can be avoided in the future. That these documents are only constructed at the end of a project is unfortunate. By that time, many of the key players have moved on and those who are left must try to remember lessons from the early stages of the project, which could be years ago. It is far better to construct the document as the project progresses.Once complete, the document is led away ready for the next project to make use of. At least, that is the theory. For the most part, the document is led away and never used again. It is dif cult to create lessons that are globally applicable. The lessons learned tend to only be used for the current project or an exactly identical project, which almost never happens.However, by looking at a number of these documents from various projects, patterns start to emerge. It was by following such an approach that William Brown, Raphael Malveau, Skip McCormick, and Tom Mowbray, collectively known as the Upstart Gang of Four in reference to the original Gang of Four, wrote the initial book on anti-patterns. The book, AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis, outlined anti-patterns not

just for issues in the code, but also in the management process which surrounds code.Patterns outlined include such humorously named patterns as The Blob and Lava Flow. The Blob, also known as the God object, is the pattern where one object grows to take on the responsibility for vast swathes of the application logic. Lava Flow is a pattern that emerges as a project ages and nobody knows if a code is still used. Developers are nervous about deleting the code because it might be used somewhere or may become useful again. There are many other patterns described in the book that are worth exploring. Just as with patterns, anti-patterns are emergent from writing code, but in this case, code which gets out of hand.This book will not cover JavaScript anti-patterns, but it is useful to remember that one of the anti-patterns is an over-application of design patterns.

Summary

Design patterns have a rich and interesting history. From their origin as tools for helping to describe how to build the structures to allow people to live together, they have grown to be applicable to a number of domains.It has now been a decade since the seminal work on applying design patterns to programming. Since then, a vast number of new patterns have been developed. Some of these patterns are general-purpose patterns such as those outlined in the GoF book, but a larger number are very specific patterns which are designed for use in a narrow domain.JavaScript also has an interesting history and is really coming of age. With server-side JavaScript taking off and large JavaScript applications becoming common, there is a need for more diligence in building JavaScript applications. It is rare to see patterns being properly exploited in most modern JavaScript code.Leaning on the teachings provided by design patterns to build modern JavaScript patterns gives one the best of both worlds. As Isaac Newton famously wrote:*"If I have seen further it is by standing on ye shoulders of Giants."*Patterns give us easily-accessible shoulders on which to stand.In the next chapter, we will look at some techniques for building structure into JavaScript. The inheritance system in JavaScript is unlike that of most other object-oriented languages and that provides us both opportunities and limits. We'll see how to build classes and modules in the JavaScript world using ES6 and up.

Questions

1. What was the official name of Javascript the first time it was released with Netscape?
2. What was the name of the alternative to Javascript released with Internet Explorer 3?
3. How many ECMAScript versions of Javascript have been released so far?
4. What was the turning point in the evolution of Javascript?
5. Where can you make use of Javascript?
6. What is the definition of a design pattern?
7. What other ways of working can be considered a sort of design pattern?
8. What are the names of the major groups of the design patterns?
9. When a design pattern can be considered an anti-pattern?

Further Reading

- If after reading this chapter you still think that you need to learn more about JavaScript this is one of the most handy books you can read and practice with https://www.packtpub.com/web-development/javascript-example
- This link will help you get a in depth knowledge about history of JavaScript https://en.wikipedia.org/wiki/JavaScript
- As we have introduced in this chapter, JavaScript is used everywhere if you want to deep in how it is used for robotics that's your book *Internet of Things Programming with JavaScript* (https://www.packtpub.com/web-development/internet-things-programming-javascript)

Object Oriented Programming

Introduction

Object-oriented programming is one of the basic pillars of the creation of design patterns and therefore, it is important to know how to use it, in JavaScript, but it is as important or more important to know how to keep our code isolated in a hostile environment to avoid having to resolve conflicts at the time of integration.

In the first part of this chapter, we will explain how to isolate our code from the rest, for this we will explain the concept of namespaces as well as how to isolate our code in modules to reuse the code in different systems. Further, in this chapter, we will refresh how to use ES5 to emulate object-oriented programming. Finally, at the end of the chapter, we will go into detail on how to program object oriented using ES6 and how to use ES6 modules natively.

The following topics are covered in this chapter:

- Creating a robust application using Namespaces.
- Creating a code using modules that can be reused
- Emulating object-oriented programming in ES5
- Using object-oriented programming in ES6 (ES2015)
- Programming using modules with ES6 (ES2015)

Isolate your code

One of the main problems we encounter when we work in companies where different teams work incrementally on the same product is how to make it easier for all of them to work in parallel and avoid conflicts on integrating the code.

In this section, we will talk about different ways that are used in JavaScript programming to avoid conflicts with other teams work, with third-party libraries and also allows us to group our code by functionality while maintaining the single responsibility principle.

Why should we avoid global variables?

In such a dynamic world as frontend development where every week dozens of new contributions to the ecosystem can appear, each one more interesting, sometimes we find ourselves with the problem of having to use third-party libraries, either for analytics, plugins, libraries or frameworks in our projects. One of the main pitfalls we encounter is how to avoid problems or conflicting names by adding an external library without having to change anything in our base code.

Some libraries have methods to prevent or deal with this type of conflict, such as jQuery which has a"noconflict" function to give it a new name and avoid conflicts with other existing libraries.

In principle, you should bear in mind that to avoid this type of problem it is recommended to never exhibit a global one outside your own library.

In the upcoming sections we will see how to avoid polluting globals keeping our code isolated from the rest, moving our code to small pieces of code with a single responsibility and converting them to modules.

Namespaces

One way to encapsulate our code and keep it isolated from the rest has been to use spaces designated under a name also called namespaces. This system is based on the naming of server language packs as the one used in Java.

```
com.companyName.packageName.subpackageName
```

It is an effective system to only expose a keyword that represents us as the company that created it, avoiding conflicts with other third-party packages.

Examples of namespaces in libraries:

ga	Google Analytics
fb	Facebook
yui	Yahoo User Interface
$	jQuery

As you can see a short name is enough but it is also very easy to be overwritten.

> *My recommendation is to use your company name as the big ones do and reduce it to the minimum expression but keeping it readable so that it is easy to remember*

Let's see a basic example of how to create a namespace:

```
var pkt = pkt || {};

 pkt.parser = {
   parseHTML: function (html) {
     // do something with html
   },
   parseCSS: function (css) {
     // do something with css
   }
 };
```

If the namespace already exists, the existing object is used as a container.

Using namespaces, in the traditional way does not avoid the pollution of the global ones since it continues adding an element to the global object but it is very useful for the organization of our code in packages.

IIFE – Immediate Invoked Function Expression

Since in JavaScript the only way to keep something private or isolated from the outside is through the closures, using an IIFE is the best way to avoid contaminating the global ones, allowing a better organization of the code and being able to expose only what is part of the public API and keeping the implementation private.

Using an IIFE, to create the namespace, passing this object as a parameter is considered a good practice, this object will be the global execution object of our code, in the browser, it will be the window object and in the server, with Node.js it will be the global object.

Another good practice is always to add `'use strict'` to our main function, as it works in cascade mode, and will prevent us from inadvertently polluting the global environment.

Use an IIFE to maintain an isolated namespace by maintaining private code and only exposing the public API.

In the following example code, you can see how we have created `countAttributes` as a private function that is used in various methods that will be exposed as public API of the parser object:

```
(function (namespace) {
   'use strict';
   function countAttributes(data) {
```

```
    // Do something generic with data
  }

  namespace.parser = {
    parseHTML: function (html) {
      var attributesLen = countAttributes(html);
      // do something with the HTML string
    },
    parseCSS: function (css) {
      var attributesLen = countAttributes(css);
      // do something with the CSS string
    }
  };

}(this.pkt || (this.pkt = {})));
```

When executing the IIFE, the object assigned to our namespace is passed on as an argument or, if it does not exist, it is created and assigned to the global object.

Although very similar to the previous one because this method continues adding an element to the global object, the possibility of being able to separate our code in public and private brings us closer to the way in which the modules work and the definitions of objects or classes that we will see in the second part of this chapter.

Modular programming

Modular programming is a software design used to separate the functionality of a program in interchangeable and independent small pieces called modules.

In this section we will deep into the various methods to create modules used in JavaScript.

UMD

In any library that is worthwhile must implement a system that is responsible for understanding the environment in which it is running and try to keep the code isolated from the rest. This system used is known as UMD. A library that implements UMD is a library that if it runs in a modular environment will be available as a module, if it is running on a server it will behave as a server module and if it does not have a choice then add one and only one keyword to the corresponding global object.

> *The UMD pattern typically attempts to offer compatibility with the most popular script loaders of the day (e.g RequireJS amongst others). In many cases it uses AMD (https://github.com/amdjs/amdjs-api/wiki/AMD) as a base, with special-casing added to handle CommonJS (http://wiki.commonjs.org/wiki/CommonJS) compatibility.*
>
> https://github.com/umdjs/umd

Let's look at an example of UMD implementation.

```
(function (global, factory) {
   if (typeof define === "function" && define.amd) {
     define(["exports"], factory);
   } else if (typeof exports !== "undefined") {
     factory(exports);
   } else {
     var mod = {
       exports: {}
     };
     factory(mod.exports);
     global.actual = mod.exports;
   }
 })(this, function (exports) {
   'use strict';
```

```
  Object.defineProperty(exports, "__esModule", {
    value: true
  });

  exports.default = 22;
});
```

UMD uses *duck typing* to interfere in which environment, the module, is being executed.

It checks if is executed in an AMD friendly environment or if it's being executed in the server and use CommonJS, if none of the checks is fulfilled then the variable to the global object.

The function where the module is defined is called factory and it will be passed or executed depending of the environment. In AMD it will be passed to the define function as a second argument and in any other case it will be executed passing the object where our module should be attached to.

> *If you want to learn more about the different ways of creating UMD check this repository:* *https://github.com/umdjs/umd/tree/master/templates Various methods to create modules.*

Module types

In the previous section we have explained how it is important to keep our code isolated from the rest and we have presented some concepts such as UMD(Universal Module Definition)and how to abstract the way to create a module through our code depending on the execution environment thus facilitating compatibility between the modular systems currently existing.

Let's review the most common systems to create modules that we can use in JavaScript

- Module revealed pattern
- AMD
- CommonJS

Revealing Module pattern

This way of creating a module is can be combined with a namespace it can also be used to interact with other modules already existing in the system.
With this format, we can maintain private functions and only expose those we want to make public as well as through the global object access to any object.

It is used in the browser and is often used in high-performance environments where adding AMD models does not help the load but can slow it down because the code can run immediately:

```
(function (global) {
   var oContainer = null;
   function setContainer(oCont) {
     oContainer = oCont;
   }
   function addZero(nTime) {
     return nTime < 10? '0' + nTime : nTime;
   }
   function getFormattedTime(dTime) {
     return addZero(dTime.getHours()) + ':' +
addZero(dTime.getMinutes()) + ':' + addZero(dTime.getSeconds());
   }
   function render() {
     oContainer.innerHTML = 'Test Module: ' + getFormattedTime(new
Date());
   }
   function destroy() {
     oContainer.innerHTML = '';
   }
   global.pkt.time = {
     render: function (oContainer) {
       setContainer(oContainer);
       render();
```

```
    },
    destroy: function () {
      destroy();
    }
  };
}(this));
```

You can read more about Revealing Module Pattern visiting https://www.lo okthink.com/blog/revealing-module-pattern

AMD

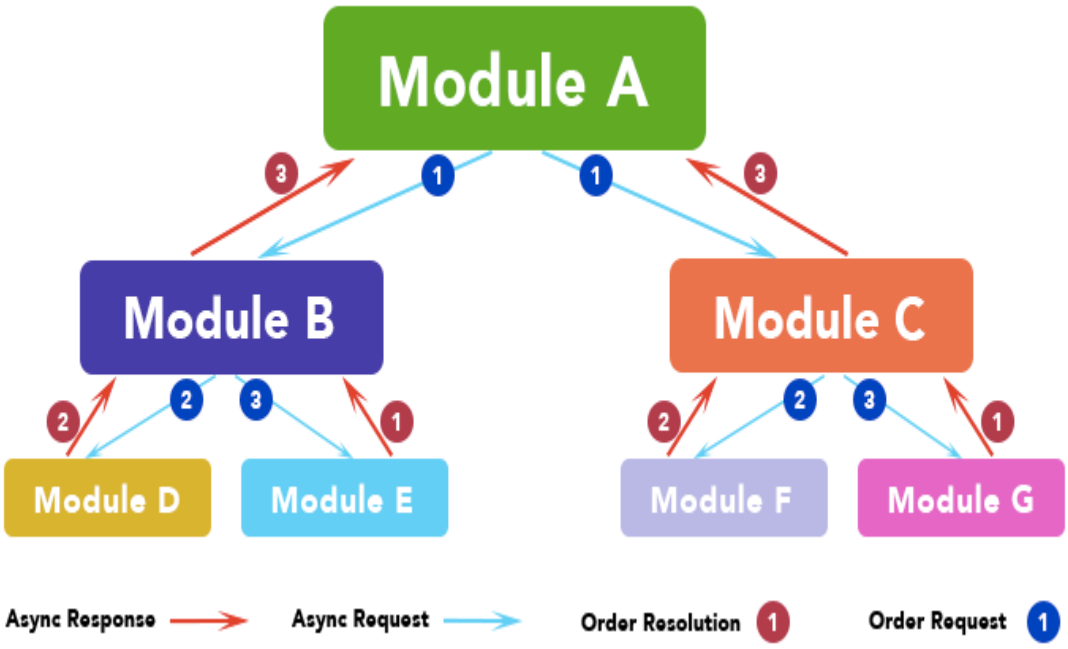
AMD stands for Asynchronous Module Definition. This system works in such a way that when the module is defined, usually at the moment of loading the file with the source code, it is not necessary when it is going to be used, but rather when it is available for later use in the system.

In JavaScript, nowadays, the most popular library for working with AMD is Require.js.

> *If you want to know more about how to use AMD modules using Require.js please visit this link:* http://requirejs.org/

AMD modules are loaded on demand and asynchronously and the code in the module will not be executed until all the dependencies are solved or in case of not being solved a timeout will return an error that the dependency could not be solved.

In the following diagram you can see how the dependencies are resolved:



In the following code, we will see how to create a module using AMD and how this code can be used later in another module where it will be loaded as a dependency.

```
// utilities.js
define('utilities', [], function () {
    return {
        countAttributes: function (string) {
            // implementation code
        }
    };
});

// parser.js
define('parser', ['utilities'], function (utilities) {
   return {
     parseHTML: function (html) {
       var attributesLen = utilities.countAttributes(html);
       // do something with the HTML string
     },
     parseCSS: function (css) {
       var attributesLen = utilities.countAttributes(css);
       // do something with the CSS string
     }
```

```
  }
});
```

The `define` function takes three arguments:

- Module name
- An array of strings with the dependencies if required
- A function that will receive as arguments the dependencies

> *You can get deep into AMD reading the following post:* *https://en.wikipedia.org/wiki/Asynchronous_module_definition*

Remember that it is good practice for every AMD module to return an object with the public API.Require.js is the most widely used system currently used in the browser although it has other competitors such as SystemJS.

> *SystemJS is a dynamic ES6 module loader used in Angular* *https://github.com/systemjs/systemjs*

CommonJS

CommonJS is a modular definition but not intended for the browser but for the server. It is the modular system used in Node.js.

> *In version 9.8.0 of Node.js, it has been added, experimentally, support ES6 modules* https://nodejs.org/api/esm.html

Unlike AMD modules that are loaded asynchronously on demand, CommonJS modules are loaded synchronously:

```
// utilities.js
var Utilities = {
  trim: function (string) {
     // implementation code should return a trimmed string.
  }
};

// parser.js
var Utilities = require('./utilities);

exports.parseHTML = function (html) {
   var trimmed = Utilities.trim(html);
   // do something with the trimmed HTML string
};
exports.parseCSS = function (css) {
   var trimmed = utilities.trim(css);
   // do something with the trimmed CSS string
};

// main.js
var parseHTML = require('parser').parseHTML;

var html = '<!DOCTYPE html><html><head><title>Mastering JavaScript
Design Patterns 3rd edition</title></head><body>The best book I ever
read</body></html>';
```

```
console.log(parseHTML(html));
```

Note that CommonJS uses its own way of loading dependencies and two different ways of exposing the public API to the other models that may require it as a dependency.

In the next sub-sections we are going to review the key points of CommonJS modules:

- require
- module.exports
- exports.<keyName>

> *If you want to learn more about CommonJS please visit: https://en.wikipedia.org/wiki/CommonJS*

require

Do not confuse with RequireJS function, this is the native mode in which dependencies are loaded asynchronously, it receives as argument the name if it is a module downloaded with NPM or the path if it is an internal module.

module.exports

module.exports is used to return an object, function or primitive as the default value of the module.

exports.<keyName>

exports.<keyName> is used to return an object, function , primitive as you can do with module.exports but in this case you will have a key to export it. It's used to set secondary values or to organize the code in small chunks of code instead of sending a big object.

Class-based languages versus JavaScript

In class-based languages, we can find four keywords (class, interface, extends and implements) which define how to work in object-oriented programming.

class

Class is like a blueprint where you define the properties and methods that will be used later and which we must instantiate to create objects. It was introduced in ES6.

interface

An interface is the definition of a specific behavior and does not have its own identity like classes because all the methods are not implemented in the interface. It is is not available in any JavaScript version, yet.

extends

It is used to extend an existing class with the intention of reusing all or part of the code. It was introduced in ES6.

implements

It is used to add the interfaces in order to add behaviors to the class. It is not available in JavaScript, yet.

PROTOTYPE EXTENSIONS

In the class-based languages the classes are the object definitions and when a class is instantiated using the keyword `new` it is when the object is obtained and this object contains its own and inherited attributes and methods. On the other hand in JavaScript everything is an object and each object has a linked prototype object. The parent class of each existing JavaScript object is Object and all objects have that prototype linked to. The methods and attributes of the prototype are not added to the object but there is only one instance of the prototype and the object does nothing but use them as its own.

Let's look at the Array prototype example:

```
var arr = [];
console.log(arr);
// []
console.log(arr.constructor);
// ƒ Array() { [native code] }
console.log(arr.__proto__.constructor);
// ƒ Array() { [native code] }
console.log(arr.__proto__.__proto__.constructor);
// ƒ Object() { [native code] }
console.log(arr.__proto__.__proto__.__proto__.constructor);
// Uncaught TypeError: Cannot read property 'constructor' of null
```

In the case of the array, we see that there is one more prototype and that we can navigate from prototype to prototype until we get an error. The first prototype constructor is of the Array type and the second one in the chain is Object.

You can also check that this string exists if you run the following:

```
console.log(arr instanceof Array);
// true
console.log(arr instanceof Object);
// true
```

In both cases "true" will be displayed indicating that it shares a prototype with that constructor. JavaScript uses this chaining of prototypes to be able to use methods and attributes of the parent classes.

Let's see how we can create a simple chain of prototypes ourselves:

```
function Mammal() {}
 Mammal.prototype.makeSound = function () {
   // do some sound
 };

 function Dog() {}
 Dog.prototype = new Mammal();
 Dog.prototype.constructor = Dog;

 var dog = new Dog();

 console.log(dog);
 // Dog {}
 console.log(dog.__proto__.constructor);
 // ƒ Dog() {}
 console.log(dog.__proto__.__proto__.constructor);
 // ƒ Mammal() {}
 console.log(dog.__proto__.__proto__.__proto__.constructor);
 //ƒ Object() { [native code] }
```

This example demonstrates how they work, the prototype extension of the object and the chaining of prototypes and how they can be used to create class inheritance.

The examples we have seen so far have been made in ES5 with the intention of showing how object orientation can be emulated in ES5 through the prototype and to serve as an introductory basis for object orientation with ES6.

OOP in ES6

One of the most important changes introduced in JavaScript in ES6, related to OOP, has been the new way of creating the object definition.

In this version, among many other things, the class, static and extended keywords that already existed in other languages have been added with the intention of facilitating communication between developers and separating normal functions from object definitions.

With the use of the new system we have the following benefits:

- A class cannot be overwritten in the same execution environment by loading a second class with the same name
- A class cannot be invoked without the use of the new keyword
- We can define static methods by adding the static keyword before defining our method
- The constructor has that name as such within the class definition
- Defining a method in the prototype is simple without having to touch the prototype ourselves

- We can extend of any kind just by using the keyword extends

Let's use as a base the example we made in the previous section to do it with ES6:

```
class Mammal {
   makeSound() {
     // do some sound
   }
 }

 class Dog extends Mammal{}

 var dog = new Dog();

 console.log(dog);  // Dog {}
 console.log(dog.__proto__.constructor);  // class Dog extends Mammal{}
 console.log(dog.__proto__.__proto__.constructor);
 /*
 class Mammal {
    makeSound() {
      // do some sound
    }
 }
 */
 console.log(dog.__proto__.__proto__.__proto__.constructor);
 //ƒ Object() { [native code] }
```

We see how the prototype chain is maintained but now this code is much cleaner and semantically correct.

If in OOP abstraction is the process by which we define which things are important and must be made public and which must remain private in order to maintain consistency or simply to avoid mistakes in execution, the encapsulation of data is the process of keeping private the sensitive data or that should be available to change from outside the class.

Encapsulating data in JavaScript is more complicated than in class-based languages because JavaScript does not have access modifiers that can be used to define which attribute or class should remain private. In JavaScript OOP you can keep something private if you don't expose it to the public API but then you will need to expose the methods needed to work with the private elements.

In JavaScript, there is a naming convention accepted worldwide by all the developers and it is to prefix the elements that we want to keep private with an underscore symbol `_speed`. This convention, although accepted, does not prevent that attribute or method from being modified from the outside, so from the point of view of encapsulation, it is not effective.

In the following piece of code we will see how we can use the constructor to encapsulate data that must remain isolated from the user:

```
class Bicycle {
   constructor() {
     let _speed = 0;
     let _speedStep = 10;
     this.getSpeed = () => {
       return _speed;
     };
     this.accelerate = () => {
       _speed += _speedStep;
     };
     this.slowDown = () => {
       _speed -= _speedStep;
     };
   }
 }

 var bicycle = new Bicycle();

 console.log(bicycle._speed);  // undefined
 console.log(bicycle.getSpeed());  // 0
 console.log(bicycle._speedStep);  // undefined
```

```
console.log(bicycle.accelerate());  // undefined
console.log(bicycle._speed);  // undefined
console.log(bicycle.getSpeed());  // 10

// Now you cannot change the values from outside
bicycle.speed = 100;

console.log(bicycle.accelerate());  // undefined
console.log(bicycle.getSpeed);  // 20
```

Although as we have seen this way of encapsulating the data works, this method has a performance problem in case there are many instances of the Bicycle object as the methods in the constructor are created in each new instance while in the prototype it does not.

Let's see what I mean by the following code example:

```
class Bicycle {
   constructor() {
     // This constructor must be the same as in the previous exercise.
   }
   start() {
     // do something to prepare the bicycle.
     console.log('Start riding the bicycle');
   }
 }

 var bicycle = new Bicycle();
 var bicycle2 = new Bicycle();

 console.log(bicycle.start === bicycle2.start);  // true

 console.log(bicycle.getSpeed === bicycle2.getSpeed);  // false
```

When we check if the start method of the instances is the same the result is true because that method is in the Bicycle prototype but when we check the same with the getSpeed method created in the constructor we can see how they are different because they have different references in memory.

To solve this problem we can use a new type of object introduced in ES6, the object is WeakMap (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WeakMap).

Let's see in the following code how we can keep our attributes private, without sacrificing performance, by adding our methods to the prototype:

```
const _speed = new WeakMap();
const _speedStep = new WeakMap();
class Bicycle {
  constructor() {
    _speed.set(this, 0);
    _speedStep.set(this, 10);
  }
  getSpeed() {
    return _speed.get(this);
  }
  accelerate() {
    _speed.set(this, this.getSpeed() + _speedStep.get(this));
  }
  slowDown() {
    _speed.set(this, this.getSpeed() - _speedStep.get(this));
  }
}

var bicycle = new Bicycle();

console.log(bicycle.getSpeed());  // 0
console.log(bicycle.accelerate());  // undefined
console.log(bicycle.getSpeed());  // 10

bicycle._speed = 100;

console.log(bicycle.getSpeed());  // 10
console.log(bicycle.accelerate());  // undefined
console.log(bicycle.getSpeed());  // 20

// Check that all the instances share the methods.
var bicycle2 = new Bicycle();

console.log(bicycle.getSpeed === bicycle2.getSpeed);  // true
```

Modular Programming with ES6

Another improvement in ES6 was the introduction of the native module concept so using ES6 you won't have to use the AMD or CommonJS modules.

> *Use SystemJS or similar to manage on-demand loading if you don't want to end up with a monolithic file in production.*

In the following code, we will see a simple example where one module is defined in one file and the other one is used as a dependency.

utilities.js

```
// Regular expression to get all the attributes.
 const regex = /(\S+)=["']?((?:.(?!["']?\s+(?:\S+)=|[>"']))+.)["']?/g;

 // In the next line we expose countAttributes as the default value
 // to be used as a dependency.
 export default function countAttributes (str) {
   return regex.match(str).length;
 };
```

parser.js

```
import countAttributes from './utilities';

 // This export method is used if you don't want to export anything by
default
 // or if you want the user to pick up only what he needs.

 export function parseHTML(html) {
```

```
  var attributesLen = countAttributes(html);
  // do something with the HTML string
};

export function parseCSS(css) {
  var attributesLen = countAttributes(css);
  // do something with the CSS string
};
```

main.js

```
import { parseHTML } from './parser';

const html = '<!DOCTYPE html><html><head><title>Mastering JavaScript
Design Patterns 3rd edition</title></head><body>The best book I ever
read</body></html>';

console.log(parseHTML(html));
```

Summary

Object-oriented programming is a very opinionated programming paradigm. In this chapter, we have started understanding how we can organize our code in modules using IIFE and also introduced the concept of UMD, AMD and CommonJS modules. We also learned about how the prototype chain works in JavaScript and how to use it in our benefit to emulate object-oriented programming in ES5 Finally we have used ES6 to do object-oriented programming and at the end of the chapter we have introduced the modular programming in ES6.

All this chapter has been related to object-oriented programming but also keeping in mind that a maintainable architecture should be modular.

In the next chapter we will talk about the functional programming paradigm that is getting trending and that solves much of the uncertainty which we have to deal with in object-oriented programming.

Questions

1. What type of language is JavaScript? Class-based or Prototype-based?
2. What is UMD?
3. What's the most important difference between AMD and CommonJS?
4. What is the last constructor in the prototype chain in JavaScript?
5. Does class or extends exist in ES5?
6. What is a constructor in JavaScript?
7. How does the prototype chaining work?
8. What can you use to keep private attributes in ES6?

Further reading

- This link will help you get an in-depth knowledge of Object-oriented programming: https://en.wikipedia.org/wiki/Object-oriented_programming

- If after reading this chapter you still think that you need to learn more about Object Oriented Programming in JavaScript you can use the most updated version of the book with the same name: https://www.packtpub.com/web-development/object-oriented-javascript-third-edition

- In this chapter, we have explained that JavaScript is a prototype-based language if you want to learn more about it you can use this link: https://en.wikipedia.org/wiki/Prototype-based_programmin

Functional Programming

Introduction

Even if you never worked with functional programming, you probably heard of it. And more every day since seems to be a hype with it. This programming paradigm, which seemed to be forgotten by the large mass of developers, has been resurfacing for a few years now. So it has not taken long for people to appear who think is easier to write a program with a functional approach, even saying you are much more productive and that you reduce the number of bugs and errors. Functional Programming has been widely adopted by JavaScript developers.
In the first part of the chapter, we will present the basic concepts of functional programming and what problems it aims to solve. Later we will glance at several of the patterns in functional programming and make correct use of them to make our programming more predictable and less prone to errors. For the examples we will use the latest versions of ECMAScript throughout the chapter.

After reading this chapter you will learn:

- Understand the basic concepts of functional programming

- Use first-class JavaScript functions for functional programming
- Understand how to use the diverse functional programming patterns
- Know what are the new features in ECMAScript useful for functional programming

Technical Requirements

As we begin with the core part of the chapter, we need to have knowledge about the below listed concepts

- Know or even using the Array functions introduced in the latest versions of ECMAScript.
- Browser to see some examples
- Use Runkit (`https://runkit.com/home`) and Babel.js (`https://babeljs.io/repl/`) for examples with ES6 Modules
- Exercises can be found in Github repo (`https://github.com/PacktPublishing/Mastering-JavaScript-Design-Patterns-Third-Edition/tree/master/Chapter03`)

What is Functional Programming?

The first thing you should know is that JavaScript is not a functional programming language. Haskell, Miranda, R, Lisp, Scala, Scheme, ML, Erlang are pure functional programming languages. Functional programming languages, as the name implies, are function-based.

In the following paragraph I will summarize in one simple sentence what functional programming is all about:

Functional programming is the declarative evaluation of pure functions intending to create immutable programs avoiding side-effects.

In this description you can notice words some new words I marked not only because they introduce new concepts but also because they are a great part of the functional programming we will examine in this chapter.

Functional programming helps us write code that is:

- More readable
- More reusable
- Easier to understand and debug
- More testable
- Less error-prone

Imperative programming versus declarative programming

Object-oriented programming is a programming paradigm in which we work using imperative programming while functional programming is a declarative form of programming.
Imperative programming is based on telling the computer HOW to perform the instructions. We tell the computer what it should do, step by step. Programming in this way is only intended to make the work easier for the computer but leaves aside the most important part of the process that is the developer.
The declarative programming focuses on WHAT we want to get and we do not care how the result has been reached as long as the result is the expected one.

Let's pay attention to the next example of imperative programming and how to improve it using functional programming:

```
const _drawer = new WeakMap();
export default class CashRegister {
  constructor() {
    _drawer.set(this, [500, 200, 100, 50, 20, 10, 5, 2, 1]);
  }
  get drawer() {
    return _drawer.get(this);
  }
  makeChange(bill, tendered) {
    let difference = tendered - bill;
    let i = 0;
```

```
    const change = [];
    let denomination = this.drawer[0];

    while(difference > 0) {
      if(difference < denomination) {
        i++;
        denomination = this.drawer[i];
        continue;
      }
      change.push(denomination);
      difference -= denomination;
    }
    return change;
  }
}
```

This code is using OOP but makeChange is a function that returns an array of bills to complete the amount given to the cashier.

> *The _drawer property is set using WeakMap to keep it private but also setting a setter so we can access it as usual but keeping it immutable because it cannot be set.*

If you want to discover what it does, use the next module code:

```
import CashRegister from './CashRegister';
const cashRegister = new CashRegister();
cashRegister.makeChange(400, 555);    // [100, 50, 5]
```

In the next code you will understand how we can transform an imperative method using functional programming:

```
import getChange from './change';

const _drawer = new WeakMap();
class CashRegister {
  constructor() {
    _drawer.set(this, [500, 200, 100, 50, 20, 10, 5, 2, 1]);
  }
  get drawer() {
    return _drawer.get(this);
```

```
  }
  makeChange(bill, tendered) {
    return getChange(tendered - bill, this.drawer);
  }
}
```

We moved the implementation of makeChange in the previous version to a new ES6 module but we didn't change the signature of our method in CashRegister. Now our method is more clear and the name of the functions is more semantic because just reading the name I know what I will get and, as an user of this function, I don't need to know about the implementation:

```
const getChange = (difference, denominations) => {
 if(difference === 0) {
   return [];
  }
  if(denominations.length === 0) {
    return false;
  }
  if(difference < denominations[0]) {
    return getChange(difference, denominations.slice(1));
  } else {
    return [denominations[0]].concat(getChange(difference -
denominations[0], denominations));
  }
};
```

The most important change in the implementation is that we have removed all the iterations and now we use recursion executing the function again when it's needed.

> *Note that in this function we are not iterating but we use recursion calling again the same function. We will examine it more in detail with other examples but one important thing in functional programming is that you should never iterate. On functional programming you have to forget about for, while, do-while...*

While the first bill in denominations is bigger than the difference, the function will continue being executed. If the

difference is lower or equals to the bill, it will concatenate this bill to the result of executing the function with a new difference.

Use this module to execute the method:

```
import CashRegister from './CashRegister';
const cashRegister = new CashRegister();
cashRegister.makeChange(400, 555);    // [100, 50, 5]
```

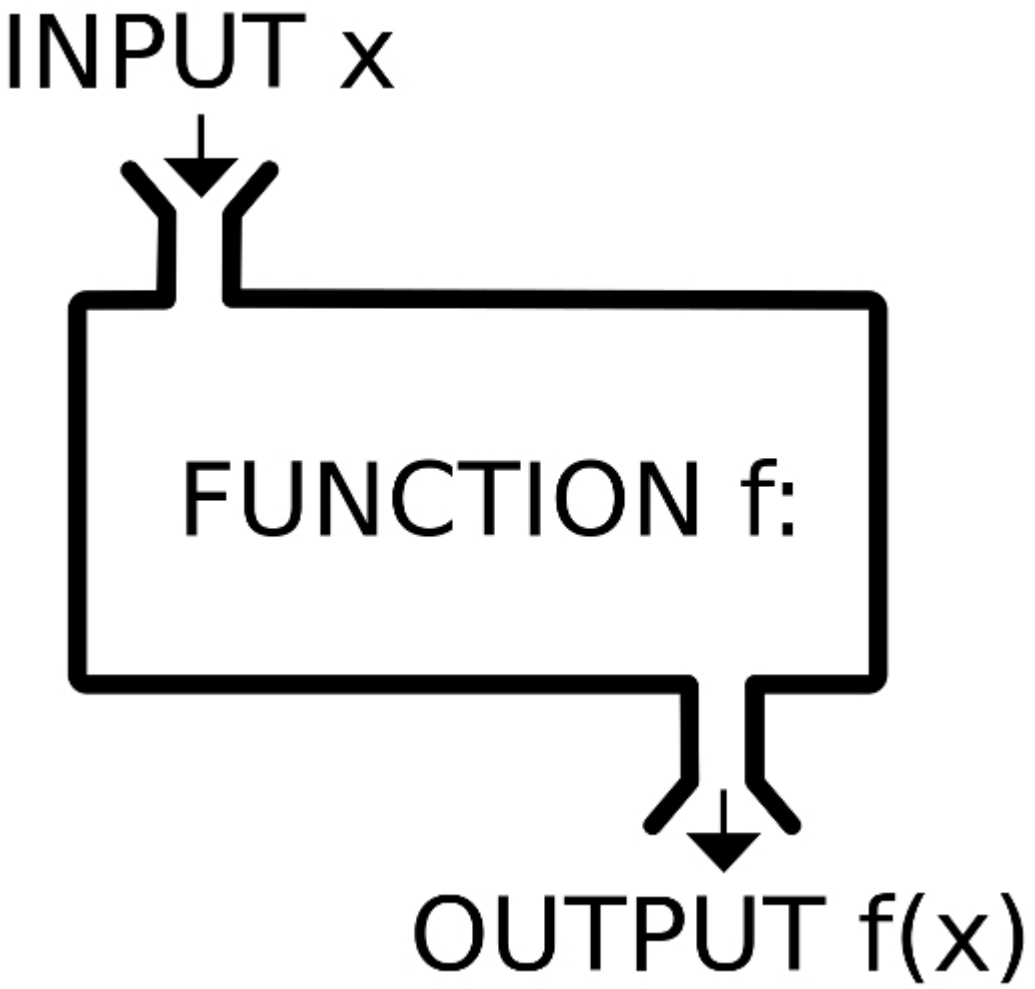
The change we have done to our code extracting the logic to a function that only works with the input and that returns something is the basic part of functional programming. We have also created a function in its own module that can be used everywhere because it doesn't depend on any state.

In this section apart from talking about imperative and declarative programming we also have introduced two important rules of functional programming.

- Never iterate, use recursion instead
- Only pure functions are allowed

Pure functions versus impure functions

In JavaScript we can do many things that in other languages like Java, Scala, C++, C# are not allowed, and this is why we will need to change our mindset to differentiate what is a pure function and what is an impure function. The pure functions are functions that get an input and return an output. The next diagram shows a pure function:



In order to create a pure function, you must follow a few rules:

- Must return something
- No matter how many times the function is executed the result should be the same if the input is also the same
- Input data must not be mutated during the execution of the function
- Not iterate over any array using for, while, do-while...
- Not sharing any state with other functions
- Implementing it should not have any side-effects in any other part of our program

Now we know what makes pure a function we will review the different ways we can do mistakes, creating impure functions, and how to avoid them.
Here we will use an example using OOP to illustrate how something valid in OOP is wrong in FP.
The exercise refers to a Cypriot bank branch office that has a method to generate new bank accounts.

> *For brevity, we will not include the all inheritance but you have all the files in the zip file with all the code examples.*

```
import CyprusBankBranch from './CyprusBankBranch';

const cyprusBankBranch0128 = new CyprusBankBranch('0128');

cyprusBankBranch0128.accounts; // []

cyprusBankBranch0128.generateAccount(); // undefined[{…}]

cyprusBankBranch0128.accounts
/**
```

```
 * [{
 * accountId: "0000000000000000",
 * ibanFormatted: "CY95 0020 0128 0000 0000 0000 0000"
 * .}]
 */
```

When we check the execution of the method, we can anticipate that this is not a pure function because we are not passing any input and because it's also not returning any value.
Other red flag is that our method is relying and changing the state of our object because as you can get before executing the function, accounts was empty but after executing it it contains an object.

> *Functional programming functions rules:*
>
> - *Should get an input and should return an output*
> - *Should not modify any state*

Now let's get deep into the method and determine what else we can find:

```
import CyprusBank from './CyprusBank';

class CyprusBankBranch extends CyprusBank {
  constructor(branchIdentifier) {
    super();
    this.branchIdentifier = branchIdentifier;
    this.accounts = [];
  }
  generateAccount() {
    let accountId;
    this.accounts.length > 0 ?
      accountId = this.accounts[this.accounts.length - 1].accountId + 1
:
      accountId = '0000000000000000';
    let ibanFormatted = (this.countryCode + Math.floor(Math.random() *
100) +
                         this.bankIdentifier + this.branchIdentifier +
                         accountId).match(/.{1,4}/g).join(' ');
```

```
    this.accounts.push({ accountId, ibanFormatted });
  }
}
```

When we study in detail what is happening inside the function, we can identify other things that are not allowed in functional programming. We are using other states of the object to fill the country code, the bank identifier and the branch identifier. Here we also realize how Math.random is used, and this is unpredictable so the value will be different every time this function is executed.

> *Functional programming functions rules:*
>
> - *Should not relay on any state*
> - *Should only work with its input to get an output*
> - *Should return always the same output on getting the same input*

Now we will improve our code moving all the code to functional programming.we can functions using functional programming. In the next code section you will realize how we improved the code to make it more functional:

```
export const getNextAccountId = (accounts) => {
    return accounts.length === 0 ? '0000000000000000' :
accounts.slice(-1).accountId + 1;
};

export const getIban = (data) => {
    return data.countryCode + data.checkDigits +
           data.bankIdentifier + data.branchIdentifier +
data.accountId;
}

export const formatIban = (iban) => iban.match(/.{1,4}/g).join(' ');

export const getNewAccount = (bankData, accounts) => {
    let accountId = getNextAccountId(accounts);
```

```
    let ibanFormatted = formatIban(getIban({
                            countryCode: bankData.countryCode,
                            checkDigits: bankData.checkDigits,
                            bankIdentifier: bankData.bankIdentifier,
                            branchIdentifier:
bankData.branchIdentifier,
                            accountId
                        }));

    return {
        accountId,
        ibanFormatted
    };
};
```

And here you have the new implementation:

```
import { getNewAccount } from './bank-utils';
import CyprusBank from './CyprusBank';

class CyprusBankBranch extends CyprusBank {
    constructor(branchIdentifier) {
        super();
        this.branchIdentifier = branchIdentifier;
        this.accounts = [];
    }
    setNewAccount() {
        let newAccount = getNewAccount({
            countryCode: this.countryCode,
            bankIdentifier: this.bankIdentifier,
            branchIdentifier: this.branchIdentifier,
            accountId
        });

        this.accounts.push(newAccount);
    }
}
```

We have changed the name of the function to be more semantic and to do what is expected to do. We couldn't make the method completely functional. In OOP if you have a method that doesn't use or relays in the context or in any property of the

object you will be compelled to set the function as a static function. This example is helpful to discover how we can implement both paradigms and get the best of both. All the functions we have moved now are reusable everywhere and also all of them follow the rules we have learn until now.

Other type of functions can be considered impure when:

- *A new Date object is generated during the execution instead of getting it through the arguments*
- *The filesystem is employed for any reason because the filesystem can change and then the result can be different*
- *The input is altered returning it or not as the output*

Higher-order functions

In JavaScript we work a lot with callbacks. Why we use so many callbacks in JavaScript is because of his single threaded nature. When we use a callback we are passing a function to another function. When we can use a function as any other variable it's because in this language functions are first-class citizens as in functional programming languages.

Example of passing a function as an argument:

```
document.getElementById('my-button').addEventListener('click', function
(event) {
  // Do something when the button is clicked.
});
```

Making calls to external resources in a browser is also dependent on a callback to notify the caller that some asynchronous operation has completed. In basic JavaScript this looks like the following:

```
let xmlhttp;
const requestData = () => {
    xmlhttp = new XMLHttpRequest()
    xmlhttp.onreadystatechange = processData;
    xmlhttp.open('GET', 'http://some.external.resource', true);
    xmlhttp.send();
}
const processData = () => {
    if (xmlhttp.readyState === 4 && xmlhttp.status === 200){
       //process returned data
    }
}
```

This is typically a cleaner approach and avoids performing complex processing in line with another function.

However, you might be more familiar with the jQuery version of this, which looks something like the following:

```
$.getJSON('http://some.external.resource', (json) => {
  //process returned data
});
```

Likely you've seen this pattern in use before without even realizing it. Passing functions into constructors as part of an options object is a commonly used approach to providing extension hooks in JavaScript libraries.

Another peculiarity of JavaScript and functional programming is that, like another variable, functions can also be returned because of the execution of another function.

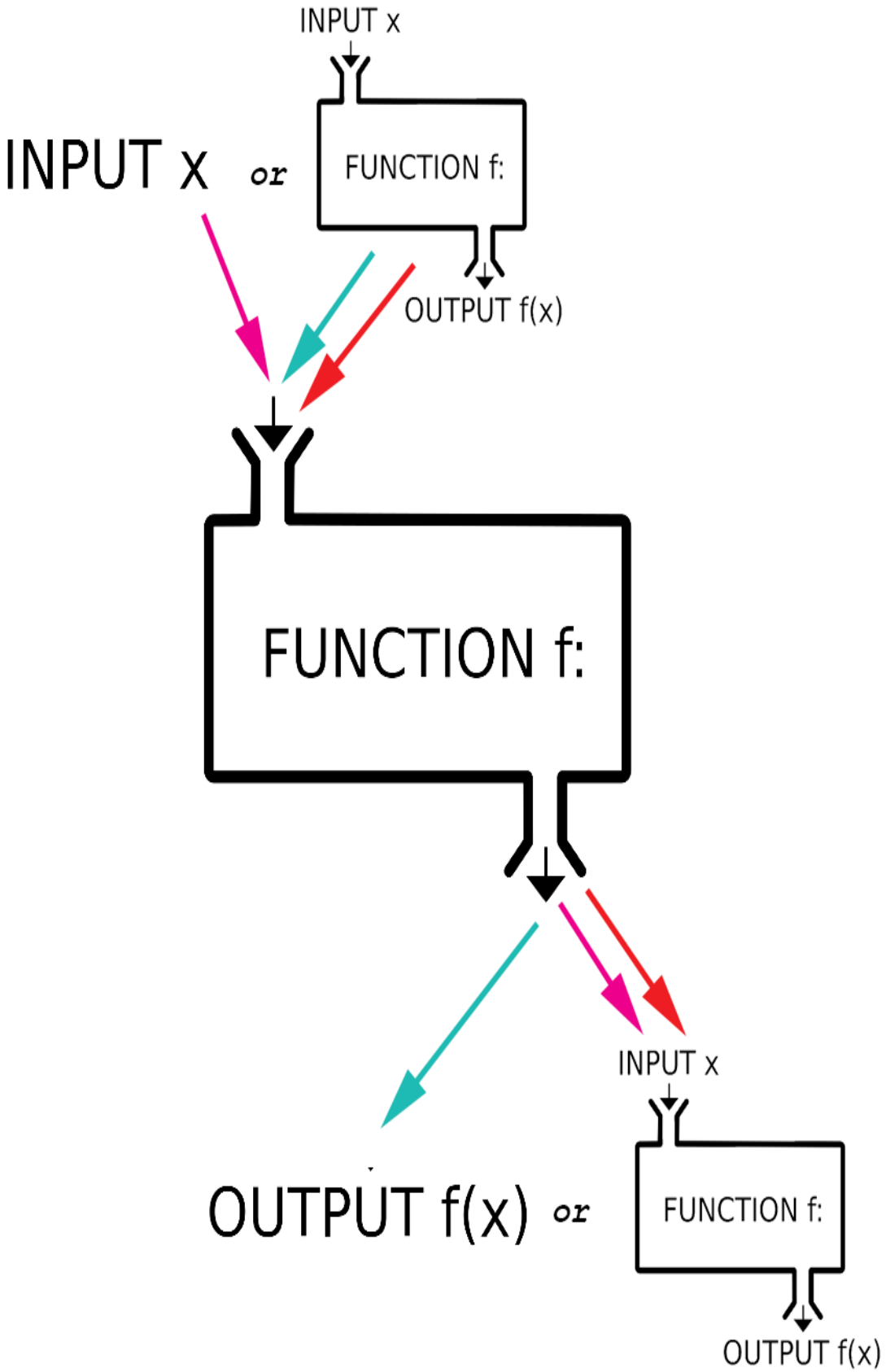
```
const greet = (greeting) => (name) => `${greeting} ${name}!`;

var sayHelloTo = greet('Hello');
sayHelloTo('Douglas');  // Hello Douglas!
```

As you can recognize when greet is executed it receives what type of greeting want to use it receives another function that expects a name to greet the person.

When a function either takes a function(s) as argument or returns a function it's denominated as a higher-order function.

$f_2(a)=f(f_1(a))$


INPUT x
INPUT x
or
FUNCTION f:
OUTPUT f(x)
FUNCTION f:
INPUT x
OUTPUT f(x)
or
FUNCTION f:
OUTPUT f(x)

The classical higher-order functions that are provided by JavaScript are:

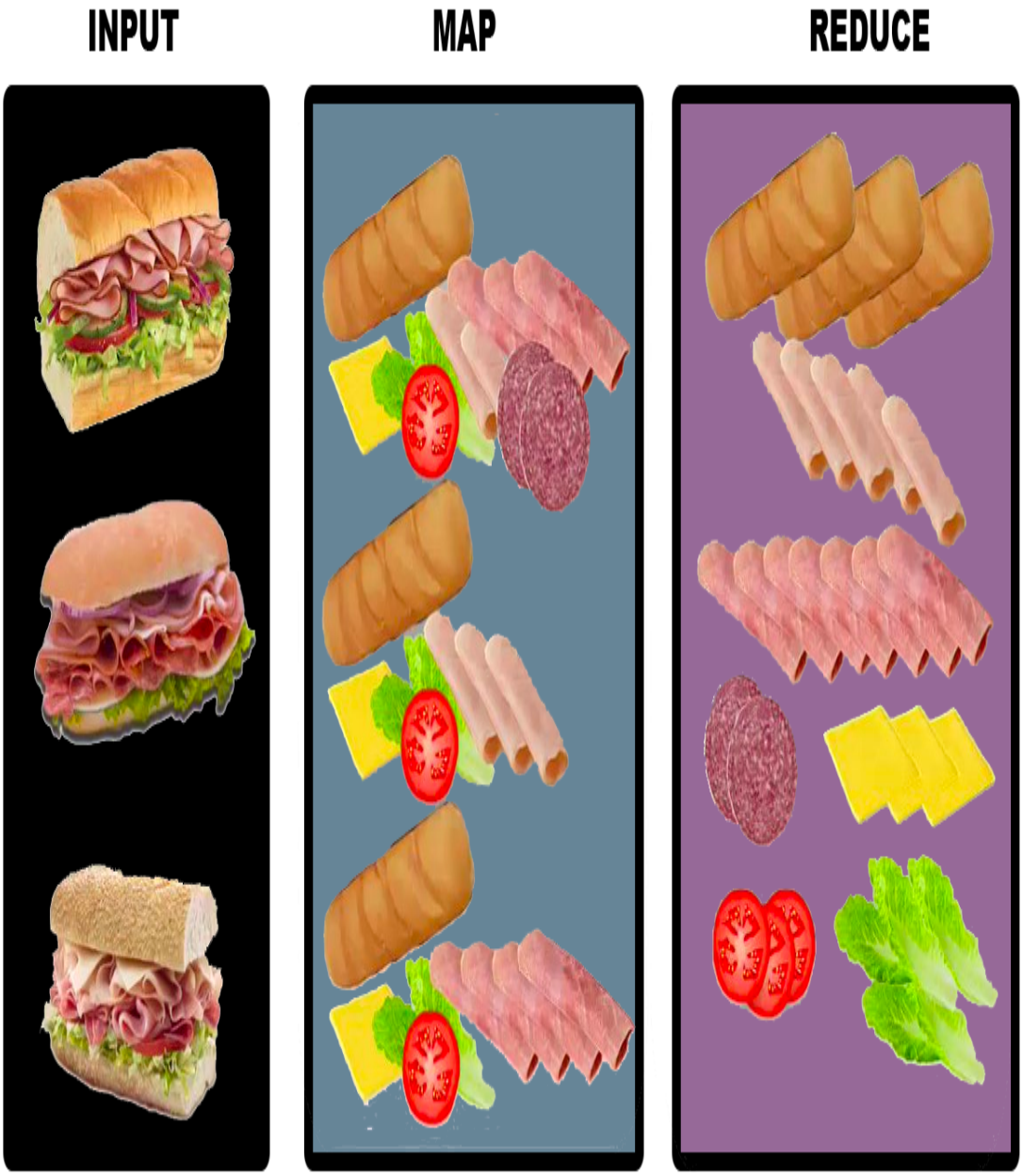
- Map
- Reduce
- Filter

In the next section we will get into these functions and how to use them to improve our productivity.

MapReduce pattern

The MapReduce pattern is used mostly for Big Data processing. In this section we will discover how to use map, reduce and filter to do not iterate any more over our arrays to reduce the complexity of our code and to use these higher-order functions to create more robust applications.

In the image below you can view a diagram that explains how the Map-Reduce pattern works:



The process of MapReduce pattern starts with some input that has a lot of information but that is too complex to get anything clear from it. The next step is to map each input to transform it in small pieces that have their own identity. The last step is to use reduce to group the data that are the same data so can be easily to use and to get information.

By the end of this section we will implement a MapReduce example so the concepts and this pattern can become more clear.

> *In this section we will explain how to use functions like map, reduce, filter but there are other functions like some, foreach... but those have a different purpose. Those functions and more where added in ECMAScript 5.1 . Note also that if you system needs to be more fast using this functions can add more time to process your arrays.*

Map

The function map is one of the prototype methods of Array object. Using map function on an array it iterates the array and per each iteration it executes the function we passed as an argument. The output of executing map over an array is a new array with the new data applying the transformations to the original data:

```
let x = [a,b,c,d];
let y = x.map(ƒ);

/* ƒ receives three parameters
 * 1. item
 * 2. index of the item in the array.
 * 3. the array that map is iterating.
 *
 * ƒ returns a new item that will be added to the new array.
*/

x === y; // false
```

For more information you can visit MDN webpage about this method at: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map

Below this line there is an example where we are traversing an array to get the value multiplied by two using a loop:

```
let numbers = [1, 2, 3, 4];

const duplicateValuesInArray = (numbersArray) => {
    for(let i = 0, len = numbersArray.length; i < len; i++) {
        numbersArray[i] = numbersArray[i] * 2;
    }
}

duplicateValuesInArray(numbers); // [2, 4, 6, 8]
```

This code cannot be functional because we are using a loop to iterate over the array and because we are mutating the data, we are getting in the input and last but not least because we are not returning any value.

Using map to do the same will benefit us with less code and with a more semantic and functional code.

```
const duplicate = (number) => number * 2;

const duplicateValuesInArray = (numbersArray) => {
 return numbersArray.map(duplicate);
};

let numbers = [1, 2, 3, 4];
duplicateValuesInArray(numbers);
```

As you can follow in this example, we have reduced the complexity in our code using map to execute or transform our array. Map returns a new array, and it never alters the input data.

Reduce and filter

Reduce and filter both are prototype methods of Array and what they do is to reduce the data to something easy to handle. Reduce is used to return a different type of input and filter is to return a subset of elements. Reduce and filter also do the same as map, they don't change the original data.

Reduce

One of the differences between reduce and any other function we will hear today is that it only will be executed if the array contains at least two elements and also the execution of the function will start at index 1. Reduce receives two arguments, the first is the function is going to be executed on iterate the array and the second one is the initial value for the accumulator. The accumulator can be anything we will use to simplify the data we are receiving, can be a primitive or an object. In the first execution the accumulator will contain the initial value plus the value of the first element in the array:

```
let x = [a,b,c,d];
let y = x.reduce(f, 0); //

/* f receives 4 parameters
 * 1. accumulator
 * 2. current value
 * 3. index
 * 4. the array that reduce is iterating.
 *
 * f returns the accumulated data
*/

x === y; // false
```

For more information you can visit MDN webpage about this method at: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reduce.

In the next snippet of code you can observe how to get the total amount of summing up all the elements of one array using imperative programming:

```
let numbers = [1, 2, 4, 6, 8, 9];

function sum(numbers) {
    var total = 0;
    for(let i = 0; i < numbers.length; i++) {
        total += numbers[i];
    }
    return numbers;
}

sum(numbers); // 30;
```

Now we will use reduce using functional programming to improve the readability and the reusability:

```
let numbers = [1, 2, 4, 6, 8, 9];

const add = (op1,op2) => op1 + op2;
const sum = (numbers) => numbers.reduce(add, 0);

sum(numbers); // 30
```

In the previous code you can realize how we created a new function called add. The add function can be reused by other functions.

Now we have seen a simple example of reduce we can study at also how it can be used to create more complex like a tag counter.

```
let tags = ['packt', 'structural', 'packt', 'behavioral', 'es6',
'patterns', 'es6', 'es6'];
let reducedTags = {};
const reducer = (accumulator,tag) => {
    var tagInAccumulatorValue = accumulator[tag];
    accumulator[tag] = tagInAccumulatorValue ? tagInAccumulatorValue +
1 : 1;
    return accumulator;
};
```

```
const tagProcessor = (tags) => tags.reduce(reducer, reducedTags);

tagProcessor(tags); // {packt: 2, structural: 1, behavioral: 1, es6: 3,
patterns: 1}
```

Filter

Filter function receives a function that will be executed on each iteration and this function should return a boolean value. It will generate a new array containing only the items that returned true. On filtering we are also doing a reduce because we are getting a subset of the original data:

```
let x = [a,b,c,d];
let y = x.filter(ƒ); //

/* ƒ receives 3 parameters
 * 1. item
 * 3. index
 * 4. the array that map is iterating.
 *
 * ƒ returns true or false.
*/

x === y; // false
```

For more information you can visit MDN webpage about this method at: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter

It's time to do a simple filtering to understand it a little better:

```
const greetings = ['hello', 'dog', 'goodbye'];

const filterGreetings = (greetings) => {
    const filteredGreetings = [];
    for(let i = 0; i < greetings.length; i++) {
        let greeting = greetings[i];
        if(greeting !== 'dog') {
           filteredGreetings.push(greeting);
        }
    }
```

```
    return filteredGreetings;
}

filterGreetings(greetings); // ['hello', 'goodbye'];
```

In the next code, we are going filter to reduce the amount of lines used in our code obtaining the same result:

```
const greetings = ['hello', 'dog', 'goodbye'];

const removeDog = (item) => {
    return item !== 'dog';
};
const filterGreetings = (greetings) => greetings.filter(removeDog);

filterGreetings(greetings); // ['hello', 'goodbye']
```

Map + Reduce + Filter - Pipes

We have gone through map, reduce and filter and it's time to use it in a real-world example.

In this example we will expose and explain a common problem that can be solved combining reduce, filter and map, in the same way it can be done for Big Data.

In this example we are the owners of a snack bar and we want to know a few details:

- How many customer took a muffin and a cappuccino
- How many bills there are
- Average amount of bills
- Who is the waiter that sells more cappuccinos together with a muffin

The next code is the main file where we are making use of map, reduce and filter using them as pipes, passing the result of one execution to the next and to the next and so on:

```
import bills from './bills';
import { flattenBill } from './mappers';
import { filterByTwoMatches } from './filters';
import { finalReport } from './reducers';

/**
 * getCappuccinoMuffinData
```

```
 * Gets an array of bills and returns an object with all the expected
data.
 * 1. It maps the object to flatten the items object in something more
manageable.
 * 2. It filters the array to get a subset of elements that have
consumed a muffin
 * and a cappuccino.
 * 3. It reduces all the information to a single object with all the
data.
 * @param {Array} bills
 * @returns Object
 */
const getCappuccinoMuffinData = (bills) => {
    return bills
            .map(flattenBill)
            .filter(filterByTwoMatches)
            .reduce(finalReport.reducer, finalReport.initialValue);
};

console.log(getCappuccinoMuffinData(bills));
// { waiters: { Sophia :1, Lorenzo :2}, totalAmount :16.5,
averageTicket :5.5, billsAmount :3 }
```

In the next piece of code you can see an example of the data we are going to use. You can find the whole file in the code zip file:

```
export const bills = [
    {
         date: 'Sat Mar 31 2018 09:08:00 GMT+0200 (CEST)',
         waiter: 'Lorenzo',
         items: [
             {
                 description: 'ristretto',
                 cost: 2.8
             }
         ]
     },
     /* Other data objects */
];
```

Here you have the filters applied in this exercise:

```
/*
 * filterByTwoMatches
 * Gets a bill as a parameter and returns true if it has two product
matches.
 * @param {Object} bill
 * @returns boolean
 */
export const filterByTwoMatches = (bill) => bill.items.productMatches
=== 2;
```

There are two reducers we will need in our exercise. The first to flatten the items in a single `productMatches` variable and the second one to get the final report object:

```
export const flattenItems = {
    /**
     * flattenItems.reducer
     * Gets two parameters the accumulator and the item and returns an
accumulator.
     * On each iteration it adds the cost of the product and if the
description matches
     * 'cappuccino' or 'muffin' it increments productMatches.
     * @param {Object} accumulator
     * @param {Object} item
     * @returns Object
     */
    reducer(accumulator, item) {
        if(['cappuccino', 'muffin'].includes(item.description)) {
            const productMatches = accumulator.productMatches;
            accumulator.productMatches = productMatches > 0 ?
productMatches + 1 : 1;
        }
        accumulator.cost += item.cost;
        return accumulator;
    },
    // Initial accumulator value
    initialValue: { productMatches: 0, cost: 0 }
};

export const finalReport = {
    /**
     * finalReport.reducer
     * Gets three parameters, the accumulator, the bill and the index
     * If it's the first time we find a waiter we add it to the waiters
```

```
     * object and we set the value to one, in any other case we just
add one
     * Increases the amount of bills and we increase the total amount
with
     * the value of the cost of the bill.
     * At the end it gets the average of the ticket.
     * @param {Object} accumulator
     * @param {Object} bill
     * @param {Number} index
     * @returns Object
     */
    reducer(accumulator, bill, index) {
        const waiter = accumulator.waiters[bill.waiter];
        accumulator.waiters[bill.waiter] = waiter > 0 ? waiter + 1 : 1;
        accumulator.billsAmount++;
        accumulator.totalAmount += bill.items.cost;
        accumulator.averageTicket = accumulator.totalAmount / (index +
1);
        return accumulator;
    },
    // Initial accumulator value
    initialValue: { waiters: {}, totalAmount: 0, averageTicket: 0,
billsAmount: 0 }
};
```

Here you have the code to flatten the bill used in the exercise:

```
import { flattenItems } from './reducers';

/**
 * flattenBill
 * Gets a bill as a parameter and returns a new object with only
 * the needed for the purpose.
 * The object contains two properties:
 * - waiter => Name of the waiter
 * - items => A new object with the cost of the whole bill and how many
product matches the expected types
 * @param {Object} bill
 * @returns Object
 */
export const flattenBill = (bill) => {
    return {
        waiter: bill.waiter,
        items: bill.items.reduce(flattenItems.reducer,
flattenItems.initialValue)
```

```
    };
};
```

To know more about MapReduce pattern and its usage in Big Data you can visit the following link:

https://en.wikipedia.org/wiki/MapReduce

Side-effects and immutability

One of the pillars of functional programming is the immutability of the data. As we have seen in methods like map, reduce or filter those methods get an array and return another array, a subset of the elements of the array or a different object or primitive but without changing the original value, these methods don't mutate the data.

When we don't mutate our data, we are avoiding the side-effects in our application because we only work with the data without changing. If we change our input data or any other global data inside our object, we can get in trouble because the side-effects.

Some of the most common side-effects are:

1. Setting a field on an object
2. Printing to the console or reading user input
3. Modifying a variable
4. Modifying a data structure in place
5. Reading from or writing to a file
6. Throwing an exception or halting with an error
7. Drawing on the screen

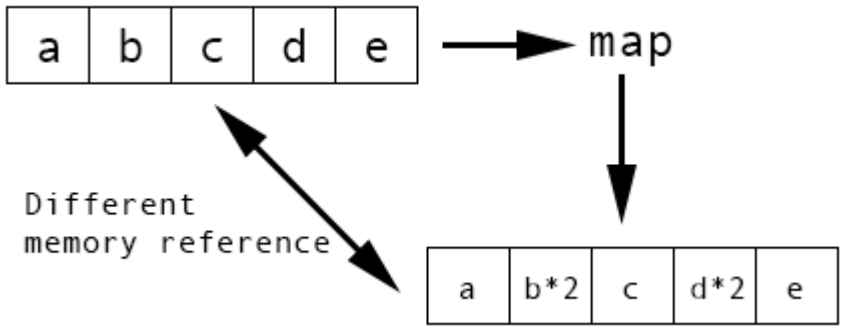
If you don't want to get any side-effect, on executing your code, you must use pure functions as you learned in the previous sections of this chapter. Only using pure functions don't keep

you, totally, save of changing your input data and using map, reduce or filter it's very costly.

When we use map, reduce or filter we are generating a completely new array what it means that the computer will consume more memory to keep those variables available.

Think about the workflow in a process where we are getting an array of number and we use map to duplicate only those that are even numbers.

In the next image you can see how map works and how it generates a complete new array.



If you see the result of mapping the array, you can see how only two elements of the array have changed but map is generating a complete new array with all the elements even with these that didn't change.

With only five items it doesn't sound harmful for you system but what could happen if instead of having an array of five items we have an array with millions of items... Yes, it will become a problem. To solve this problem in functional programming, we can use libraries, like Mori or Immutable.js, that work using tree nodes making it more efficient.

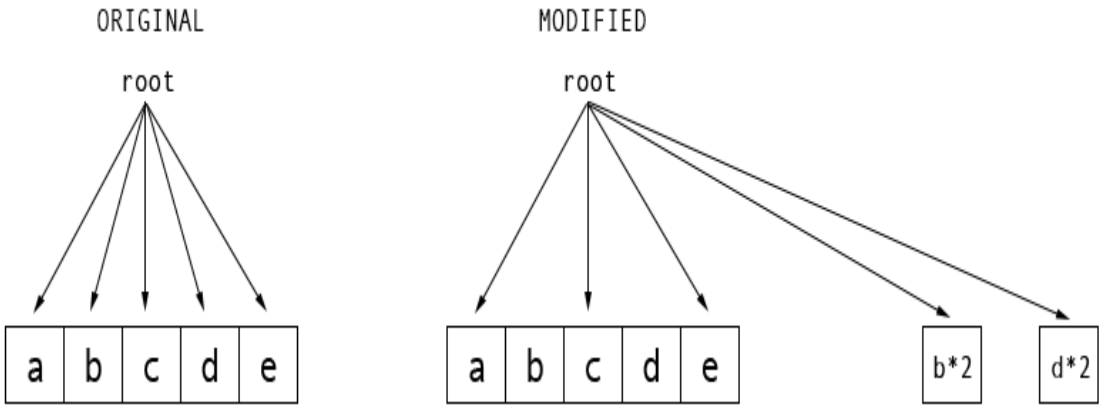
Using these libraries you can avoid the overhead of creating a new array every time. When a new array is created it reuses the items of the original links and links the new ones.

Immutable.js

To use functional programming more efficiently, you should use these libraries and in this section we will introduce you to Immutable.js.

Immutable.js uses a binary balanced tree to keep the track of the nodes, reimplements the array methods implements others to work with objects.

In the following diagram you can see how using Mori.js or Immutable.js libraries can help on reusing the data we don't want to change using binary balanced trees:




```
const { Map } = require('immutable');
const map = Map({ a: { fullname: 'John Doe' }, b: { fullname: 'Jane
Doe' } });
const set1 = map.set('a', { fullname: 'James Sullivan' });
const set2 = map.set('a', { fullname: 'Joe McCloud' });

set1.get('a') !== set2.get('a') && set1.get('b') === set2.get('b');
// true
```

On reviewing this code we can see how we can change the value of a and it generates two new maps but at the last line you can see how b remained the same.

> *If you want to learn more about Immutable.js and its different methods and objects visit the following link.*
>
> *https://facebook.github.io/immutable-js/*

ES6 tail call optimizitation

Using functional programming is very helpful in many cases but it can also become a problem when we need to execute a function several times because it very expensive for the computer.

The most simple and most expensive function that will help us to explain the problem is to get the factorial of a number:

```
const factorial = (num) => {
    if(num < 0) {
        return -1;
    } else if (num == 0) {
      return 1;
    } else {
      return (num * factorial(num - 1));
    }
};

factorial(10); // 3628800
factorial(11); // 39916800
factorial(1000000) // Maximum call stack size exceeded
```

Here we have two simple examples with small numbers but what will happen if we pass a very high number is that the JavaScript engine will not be able to handle it and it will get a Maximum call stack size exceeded error.

In the next screenshot you can see how it throws a Range Error on executing the factorial function with a very high value.

```
> const factorial = (num) => {
      if(num < 0) {
          return -1;
      } else if (num == 0) {
        return 1;
      } else {
        return (num * factorial(num - 1));
      }
  };
< undefined
> factorial(1000000)
✕ ▶Uncaught RangeError: Maximum call stack size exceeded
      at factorial (<anonymous>:1:19)
      at factorial (<anonymous>:7:21)
      at factorial (<anonymous>:7:21)
      at factorial (<anonymous>:7:21)
      at factorial (<anonymous>:7:21)
      at factorial (<anonymous>:7:21)
      at factorial (<anonymous>:7:21)
      at factorial (<anonymous>:7:21)
      at factorial (<anonymous>:7:21)
      at factorial (<anonymous>:7:21)
      at factorial (<anonymous>:7:21)
```

In ES6 they added a feature called **tail call optimization**. The tail call optimization improvement is that if the return line is a function, it will not deal with call stack issues. In the next code you can see how we use this new feature together with the default properties to set an accumulator:

```
const factorial = (number, accumulator = 1) => {
    if(number < 2) {
        return accumulator;
    }
    return factorial(number - 1, number * accumulator);
}
```

Here it will return Infinity because the number is too big to be handled by JavaScript but it will not fail because the stack size.

Memoization

Memoization is the process of storing the results of expensive function calls and returning the cached result when the same inputs occur again.

If we use the factorial example, we can avoid to repeat calls that have been already done and return the value stored from a previous call.

```
const privateFactorial = {};
const factorial = (num) => {
    if(privateFactorial[num]) {
        return privateFactorial[num];
    }
    if(num < 0) {
        return -1;
    } else if (num == 0) {
      return 1;
    } else {
      return (privateFactorial[num] = num * factorial(num - 1));
    }
};

console.time(); // API to track the time a function takes to execute.
factorial(150); // 8.159152832478977e+47
console.timeEnd(); // Returns the ms taken to end the execution.
// default: 0.072998046875ms

console.time(); // API to track the time a function takes to execute.
factorial(150); // 8.159152832478977e+47
console.timeEnd(); // Returns the ms taken to end the execution.
// default: 0.0048828125ms
```

If we compare both executions, we can also calculate what is the percentage of improvement in speed:

```
const op1 = 0.072998046875;
const op2 = 0.0048828125;
const improvement = (((op1 - op2) / op1) * 100).toFixed(2);  // 93.31%
```

Implementation

As you have seen before we have to create a private variable to cache all the different executions so that can be accessed easily but in this section we will create a function that can be used to memoize every function without adding any private property by ourselves:

```
const memoize = (fn) => {
    let cache = {};
    return (...args) => { // Use rest arguments to get an array
        let n = args[0]; // Get the first argument.
        if (n in cache) {
            return cache[n];  // Returns the cached value
        } else {
             let result = fn(n); // Executes the function
             cache[n] = result;  // Caches the result.
             return result;
        }
    }
}
```

In the next piece of code you can find the implementation of factorial memoized function using the memoize function we just created:

```
const factorial = (number, accumulator = 1) => {
    if(number < 2) {
        return accumulator;
    }
    return factorial(number - 1, number * accumulator);
};

const factorialMemo = memoize(factorial);

console.time();
```

```
factorialMemo(150);    // 5.7133839564458575e+262
console.timeEnd();    // default: 0.094970703125ms

console.time();
factorialMemo(150);    // 5.7133839564458575e+262
console.timeEnd();    // default: 0.008056640625ms
```

Lazy instantiation

If you go into a high-end coffee shop and place an order for some overly complex beverage (Grande Chai Tea Latte, Three Pump, Skim Milk, Lite Water, No Foam, Extra Hot anybody?), then that beverage is going to be made on the fly and not in advance. Even if the coffee shop knew what all the orders that were going to come in that day would be, they would still not make all the beverages up front. Firstly, because it would cause numerous ruined, cold beverages, and secondly, it would be a very long time for the first customer to get their order if they had to wait for all the orders of the day to be completed.

Instead coffee shops, follow a just-in-time approach to craft beverages. They make them when they're ordered. We can apply a similar approach to our code through the use of a technique known as lazy instantiation or lazy initialization.

Consider an object that is expensive to create that is to say it takes a great deal of time to create the object. If we are unsure if the object's value will be needed, we can defer its full creation until later.

Implementation

Let's jump into an example of this. Westeros isn't really big on expensive coffee shops but they do love a good bakery. This bakery takes requests for different bread types in advance and then bakes them all at once should they get an order. However, creating the bread object is an expensive operation, so we would like to defer that until somebody comes to pick up the bread:

```
const Bread = ((() => {
    return class Bread {
        constructor(breadType) {
 this.breadType = breadType;
 //some complex, time consuming operation
 console.log("Bread " + breadType + " created.")
 }
 };
})());
```

We start by creating a list of bread types to create as needed. This list is appended to by ordering a bread type:

```
const Bakery = ((() => {
    return class Bakery {
        constructor() {
            this.requiredBreads = [];
        }
        orderBreadType(breadType) {
            this.requiredBreads.push(breadType);
        }
    };
})());
```

This allows for breads to be rapidly added to the required breads list without paying the price for each bread to be created.

Now, when `pickUpBread` is called, we'll actually create the breads:

```
const Bakery = ((() => {
    return class Bakery {
        constructor() {
            this.requiredBreads = [];
        }
        orderBreadType(breadType) {
            this.requiredBreads.push(breadType);
        }
        pickUpBread(breadType) {
            console.log("Pickup of bread " + breadType + " requested");
            if (!this.breads) {
              this.createBreads();
            }
            return this.breads.filter((bread) => {
                return bread.breadType === breadType;
            })[0];
        }
        createBreads() {
            this.breads = this.requiredBreads.map((requiredBread) => {
                return new Bread(requiredBread);
            });
        }
  };
})());
```

Then we will call a series of operations:

```
var bakery = new Westeros.FoodSuppliers.Bakery();
bakery.orderBreadType("Brioche");
bakery.orderBreadType("Anadama bread");
bakery.orderBreadType("Chapati");
bakery.orderBreadType("Focaccia");

console.log(bakery.pickUpBread("Brioche").breadType + "picked up");
```

This will result in the following output:

```
Pickup of bread Brioche requested.
Bread Brioche created.
Bread Anadama bread created.
Bread Chapati created.
Bread Focaccia created.
Brioche picked up
```

You can see that the collection of actual breads is left until after the pickup has been requested.

Lazy instantiation can be used to simplify asynchronous programming. Promises are an approach to simplifying callbacks, which are common in JavaScript. Instead of building up complicated callbacks, a promise is an object that contains a state and a result. When first called, the promise is in an unresolved state; once the `async` operation completes, the state is updated to complete, and the result filled in. You can think of the result as being lazily instantiated. We'll watch at promises and promise libraries in more detail in Chapter 12, Web Patterns.

Being lazy can save you quite a bit of time in creating expensive objects that end up never being used.

Currying – Partial application

Currying is the technique of translating the evaluation of a function that takes multiple arguments into evaluating a sequence of functions, each with a single argument.

Currying makes use of high-order functions to store the arguments of the first execution and returns another function that will receive other parameters that will be used to do something with both.

Creating email accounts for different users and domains can be very time consuming let's see how to use currying to make it simpler.

```
const createEmailCreator = (domain) => {
    return (username) => `${username}@${domain}`;
};

const createGmailAccounts = createEmailCreator('gmail.com');
const createHotmailAccounts = createEmailCreator('hotmail.com');

createGmailAccounts('john_doe'); // john_doe@gmail.com
createGmailAccounts('jane_doe'); // jane_doe@gmail.com

createHotmailAccounts('jimmy_santillana'); //
jimmy_santillana@hotmail.com
createHotmailAccounts('gerard_picard'); // gerard_piccard@hotmail.com
```

The benefits of currying are that you can use this function generators to set the value at the beginning of the program and only execute the returned function when it's needed. Currying allows to delegate the execution during the execution of the program.

Composing

Composing is another benefit of functional programming allowing to generate super functions using existing functions.

In the next piece of code you can see a simple example that requires composing to reduce the complexity and imperative programming:

```
const hyphenate = (word) => word.match(new RegExp(`.
{1,${Math.floor(word.length/2)}}`, 'g')).join('-');

const reverse = (word) => word.split('').reverse().join('');

const toUpperCase = (word) => word.toUpperCase();

const words = ['web', 'design', 'patterns'];

let newWords = words.map(hyphenate);
newWords = newWords.map(reverse);
newWords = newWords.map(toUpperCase);


newWords; // ["B-E-W", "NGI-SED", "SNRE-TTAP"]
```

There is one way to reduce the iterations using composition:

```
const compose = (...fns) => fns.reverse().reduce((prevFn, nextFn) =>
    value => nextFn(prevFn(value)),
    value => value
);
const hyphenate = (word) => word.match(new RegExp(`.
{1,${Math.ceil(word.length/2)}}`, 'g')).join('-');

const reverse = (word) => word.split('').reverse().join('');
```

```
const toUpperCase = (word) => word.toUpperCase();

const composed = compose(hyphenate, reverse, toUpperCase);

const words = ['web', 'design', 'patterns'];

words.map(composed); // ["BE-W", "NGI-SED", "SNRE-TTAP"]
```

Functional libraries

In this chapter we have seen how to implement functional programming using vanilla JavaScript but there are libraries that provide with more functional programming features so it can make your work easier.

React.js

React.js is a library created by Facebook. The difference between React.js and other frameworks as Angular.js, Ember.js is that it's not a framework. React is only the view layer of your application. To learn more about React.js visit this link: https://reactjs.org/.

Redux.js

Redux.js is created also by Facebook and this library that helps to keep a single entry point to store the state of your application. To learn more about Redux.js visit this link: `https://redux.js.org/introduction`.

Rxjs

Reactive extensions was created by Microsoft. Reactive extensions is available in many languages. This library allows to use piping to program functional and reactive. To learn more about Rxjs visit this link: http://reactivex.io/rxjs/.

Lodash

Lodash is a library based in underscore.js. Lodash is, from our list, the first that was created and focused on functional programming. If you want to learn more about Lodash visit this link: https://lodash.com/.

Ramda.js

Ramda.js is a pure functional programming library that makes easy to create functional pipelines and focused on never mutate user data. If you want to learn more about Ramda.js visit this link: https://ramdajs.com/.

Bacon.js

Last but not least we have Bacon.js a library very similar to the concept of Ramda.js but also focused on making easier functional programming as well as reactive programming. If you want to learn more about Bacon.js visit this link: https://baconjs.github.io/.

Summary

JavaScript is not a functional programming language. That is not to say it isn't possible to apply some ideas from functional programming to it. These approaches enable cleaner, easier-to-debug code. Some might even argue that the number of issues will be reduced although I have never seen any convincing studies on that.

Functional programming is predictable, safe but also transparent and modular.

In this chapter, we looked at six different patterns. Lazy instantiation, memoization, and immutability are all creational patterns. Function passing is a structural pattern as well as a behavioral one. Accumulators are also behavioral in nature. Filters and pipes don't really fall into any of the GoF categories, so one might think of these as a style pattern.

In the next chapter, we'll study at a reactive programming to use streaming and observe for changes so the code will be self managed.

Questions

1. What is the difference between imperative and declarative programming?
2. What are the benefits of functional programming?
3. What makes a function pure or impure?
4. What is the definition of higher-order functions?
5. Which are the classic higher-order functions in JavaScript?
6. What is the difference between map, reduce and filter?
7. Which kind of developers use the pattern map-reduce?
8. How can you avoid side-effects in functions?
9. What type of tree uses Immutable.js or Mori to be more efficient?
10. When you need to use currying?
11. What is the pattern to create super functions using simple functions?

Further Reading

- If you want to keep learning or get more deeply in functional programming, you can read "Functional Programming in JavaScript" from PacktPub.
- In the libraries section you have all the links to the libraries available in JavaScript to facilitate functional programming, this is also something you should read to use them or to get inspired.
- To know the Array methods is also mandatory to learn how they work and to get updated. Available on : `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array`.

Reactive Programming

Introduction

I once read a book that suggested that Newton came up with the idea for calculus when he was observing the flow of a river around a reed. I've never been able to find any other source which supports that assertion. It is, however, a nice picture to hold in your mind. Calculus deals with understanding how the state of a system changes over time. Most developers will rarely have to deal with calculus in their day to day work. They will, however, have to deal with systems changing. After all, having a system which doesn't change at all is pretty boring.

Over the last few years a number of different ideas have arisen in the area of treating change as a stream of events. Given a starting position and a stream of events it should be possible to figure out the state of the system. By replaying this series of events we can recreate the current state of the aggregate. This seems like a roundabout way of storing the state of an object but it is actually very useful for a number of situations. It is also stunningly useful for audit scenarios as it is possible to pull the system back to the state it was in at any point in time by simply halting the replay at a time index. How frequently have you been asked, "why is the system in this state?", and you've been unable to reply? With an event store the answer should be easy to ascertain.

The following topics will be covered in this chapter:

- Application state changes
- Streams

- Filtering streams
- Merging streams
- Streams for multiplexing

Technical Requirements

As we begin with the core part of the chapter, we need to have knowledge about the below listed concepts

- Know how to do functional programming
- Browser to see some examples
- Use Runkit (`https://runkit.com/home`) and Babel.js (`https://babeljs.io/repl/`) for examples with ES6 Modules
- Download and import `http://reactivex.io/rxjs/`
- Exercises can be found in `Github repo`

Application state changes

Within an application we can think of all the events happening as a similar stream of events. The user clicks on a button? Event. The user's mouse enters some region? Event. A clock ticks? Event. In both front and backend applications, events are the things which trigger changes in state. You're likely already using events for event listeners. Consider attaching a click handler to a button:

```
var item = document.getElementById("item1");
item. addEventListener("click", function(event){ /*do something */ });
```

In this code we have attached a handler to the `click` event. This is fairly simple code but think about how rapidly the complexity of this code increases when we add conditions like "ignore additional click for 500ms once a click is fired to prevent people double-clicking" and "Fire a different event if the *Ctrl* key is being held when the button is clicked". Reactive programming or functional reactive programming provides a simple solution to these complex interaction scenarios through use of streams. Let's explore how your code can benefit from leveraging reactive programming.

Streams

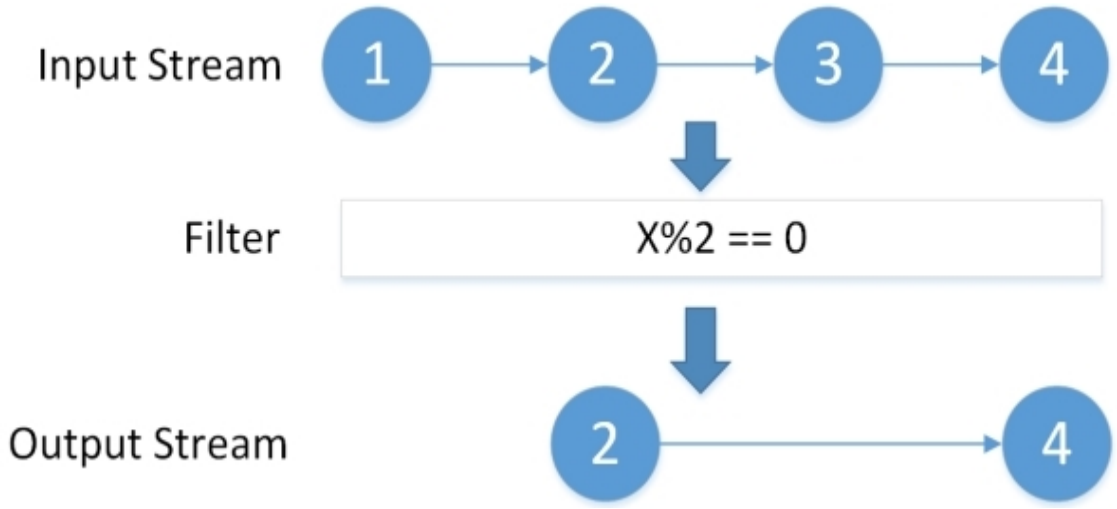
The easiest way to think of an event stream is not to think of the streams you've probably used before in programming, input reader streams, but to think of arrays. Let's say that you have an array with a series of numbers in it:

```
[1, 4, 6, 9, 34, 56, 77, 1, 2, 3, 6, 10]
```

Now you want to filter this array to only show you even numbers. In modern JavaScript this is easily done through the use of the `filter` function on the array:

```
[1, 4, 6, 9, 34, 56, 77, 1, 2, 3, 6, 10].filter((x)=>x%2==0) =>
[4, 6, 34, 56, 2, 6, 10]
```

A graphical representation can be seen here:



The filtering function here remains the same should we have ten items in the array or ten thousand items in the array. Now, what if the source array had new items being appended to it all the time? We would like to keep our dependent array up-to-date by inserting any new items which are even, into it. To do this we could hook into the `add` function on the array using a pattern-like decorator. Using a decorator we could call the filter method and, if a match was found, we would add it to the filtered array.

Streams are, in effect, an observable on a collection of future events. There are a number of interesting problems which can be solved using operations on streams. Let's start with a simple problem: handling clicks. This problem is so simple that, on the surface, it doesn't seem like there is any advantage to using streams. Don't worry we'll make it more difficult as we go along.

For the most part this book avoids making use of any specific JavaScript libraries. The idea is that patterns should be able to be implemented with ease without a great deal of ceremony. However, in this case we're actually going to make use of a library because streams have a few nuances to their implementation for which we'd like some syntactic niceties.

There are a number of stream libraries in JavaScript Reactive.js, Bacon.js, and RxJS to name a few. Each one has various advantages and disadvantages but the specifics are outside the purview of this book. In this book we'll make use of Reactive Extensions for JavaScript, the source code for which can be found on GitHub at `https://github.com/Reactive-Extensions/RxJS`.

Let's start with a brief piece of HTML:

```
<body>
    <button id="button"> Click Me!</button>
    <span id="output"></span>
</body>
```

To this, let's add a quick click counter:

```
<script>
    var counter = 0;
    var button = document.getElementById('button');
    var source = Rx.Observable.fromEvent(button, 'click');
    var subscription = source.subscribe(function (e) {
        counter++;
         output.innerHTML = "Clicked " + counter + " time" + (counter >
1 ? "s" : "");
    });
</script>
```

Here you can see we're creating a new stream of events from the `click` event on the button. The newly created stream is commonly referred to as a metastream. Whenever an event is emitted from the source stream it is automatically manipulated and published, as needed, to the metastream. We subscribe to this stream and increment a counter. If we wanted to react to only the even numbered events, we could do so by subscribing a second function to the stream:

```
var incrementSubscription = source.subscribe(() => counter++);
var subscription = source.filter(x=>counter%2==0).subscribe(function
(e) {
    output.innerHTML = "Clicked " + counter + " time" +(counter > 1 ?
"s" : "");
});
```

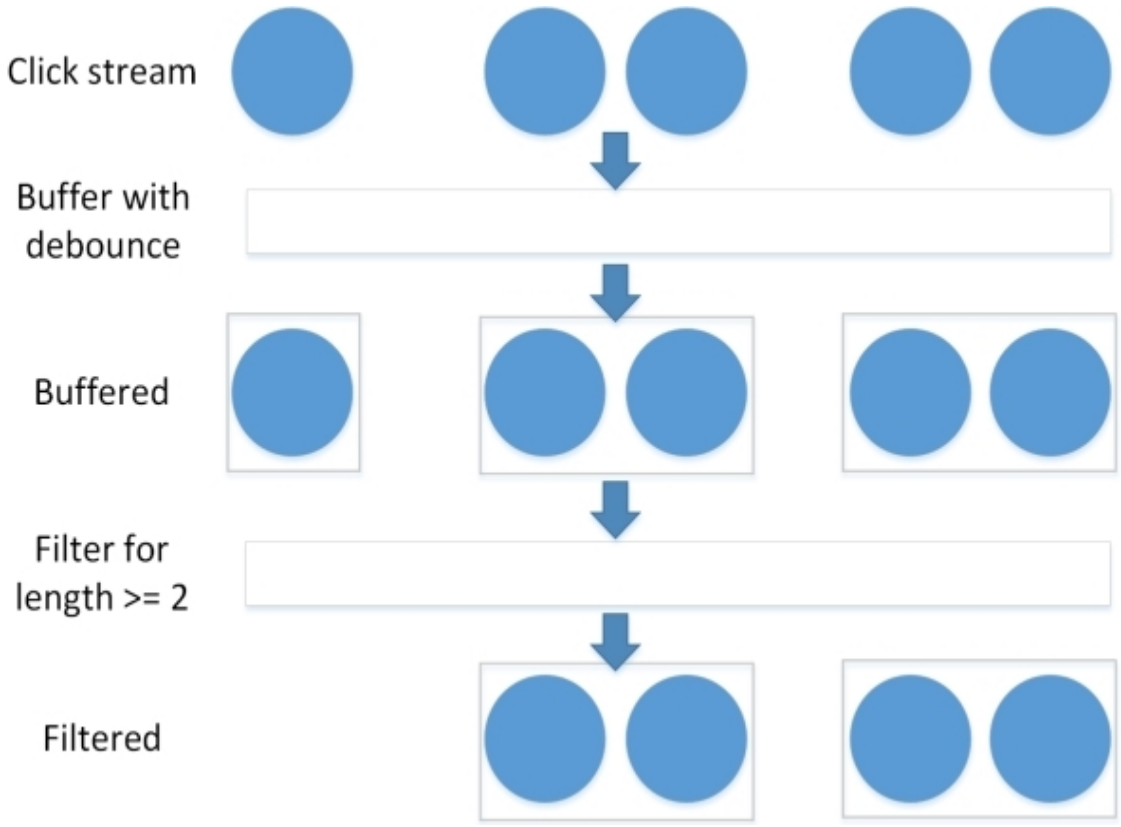
Here you can see that we're applying a filter to the stream such that the counter is distinct from the function which updates the screen. Keeping a counter outside of the streams like this feels dirty, though, doesn't it? Chances are that incrementing every

other click isn't the goal of this function anyway. It is much more likely that we would like to run a function only on double click.

This is difficult to do with traditional methods, however these sorts of complex interactions are easy to achieve using streams. You can see how we might approach the problem in this code:

```
source.buffer(() => source.debounce(250))
    .map((list) => list.length)
    .filter((x) => x >= 2)
    .subscribe((x)=> {
         counter++;
         output.innerHTML = "Clicked " + counter + " time" + (counter >
1 ? "s" : "");
    });
```

Here we take the click stream and buffer the stream using a debounce to generate the boundaries of the buffer. Debouncing is a term from the hardware world which means that we clean up a noisy signal into a single event. When a physical button is pushed, there are often a couple of additional high or low signals instead of the single point signal we would like. In effect we eliminate repeated signals which occur within a window. In this case we wait `250` ms before firing an event to move to a new buffer. The buffer contains all the events fired during the debouncing and passes on a list of them to the next function in the chain. The map function generates a new stream with the list length as the contents. Next, we filter the stream to show only events with a value of 2 or more, that's two clicks or more. The stream of events look like the following diagram:



Performing the same logic as this using traditional event listeners and call-backs would be quite difficult. One could easily imagine a far more complex workflow that would spiral out of control. FRP allows for a more streamlined approach to handling events.

Filtering streams

As we saw in the preceding section, it is possible to filter a stream of events and, from it produce a new stream of events. You might be familiar with being able to filter items in an array. ES5 introduced a number of new operators for arrays such as **filter** and **some**. The first of these produces a new array containing only elements which match the rule in the filter. `Some` is a similar function which simply returns `true` if any element of the array matches. These same sorts of functions are also supported on streams as well as functions you might be familiar with from functional languages such as First and Last. In addition to the functions which would make sense for arrays, there are a number of time series based functions which make much more sense when you consider that streams exist in time.

We've already seen debounce which is an example of a time based filter. Another very simple application of debounce is to prevent the annoying bug of users double-clicking a submit button. Consider how much simpler the code for that is using a stream:

```
Rx.Observable
    .FromEvent(button, "click")
    .debounce(1000).subscribe((x)=>doSomething());
```

You might also find it that functions like Sample – which generates a set of events from a time window. This is a very handy function when we're dealing with observables which may produce a large number of events. Consider an example from our example world of Westeros.

Unfortunately, Westeros is quite a violent place where people seem to die in unpleasant ways. So many people die that we can't possibly keep an eye on each one so we'd like to just sample the data and gather a few causes of death.

To simulate this incoming stream, we will start with an array, something like the following:

```
var deaths = [
    {
         Name:"Stannis",
         Cause: "Cold"
     },
     {
         Name: "Tyrion",
         Cause: "Stabbing"
     },
    …
];
```

You can see we're using an array to simulate a stream of events. This can be done with any stream and is a remarkably easy way to perform testing on complex code. You can build a stream of events in an array and then publish them with appropriate delays giving an accurate representation of anything from a stream of events from the filesystem to user interactions.

Now we need to make our array into a stream of events. Fortunately, there are some shortcuts for doing that using the `from` method. This will simply return a stream which is immediately executed. What we'd like is to pretend we have a regularly distributed stream of events or, in our rather morbid case, deaths. This can be done by using two methods from RxJS: `interval` and `zip`.

`interval` creates a stream of events at a regular interval.

`zip` matches up pairs of events from two streams. Together these two methods will emit a new stream of events at a regular interval:

```
function generateDeathsStream(deaths) {
    return Rx.Observable.from(deaths).zip(Rx.Observable.interval(500),
(death,_)=>death);
}
```

In this code we zip together the deaths array with an interval stream which fires every `500` ms. Because we're not super interested in the interval event we simply discard it and project the item from the array onwards.

Now we can sample this stream by simply taking a sample and then subscribing to it. Here we're sampling every `1500` ms:

```
generateDeathsStream(deaths).sample(1500).subscribe((item) => { /*do
something */ });
```

You can have as many subscribers to a stream as you like so if you wanted to perform some sampling, as well as perhaps some aggregate functions like simply counting the events, you could do so by having several subscribers:

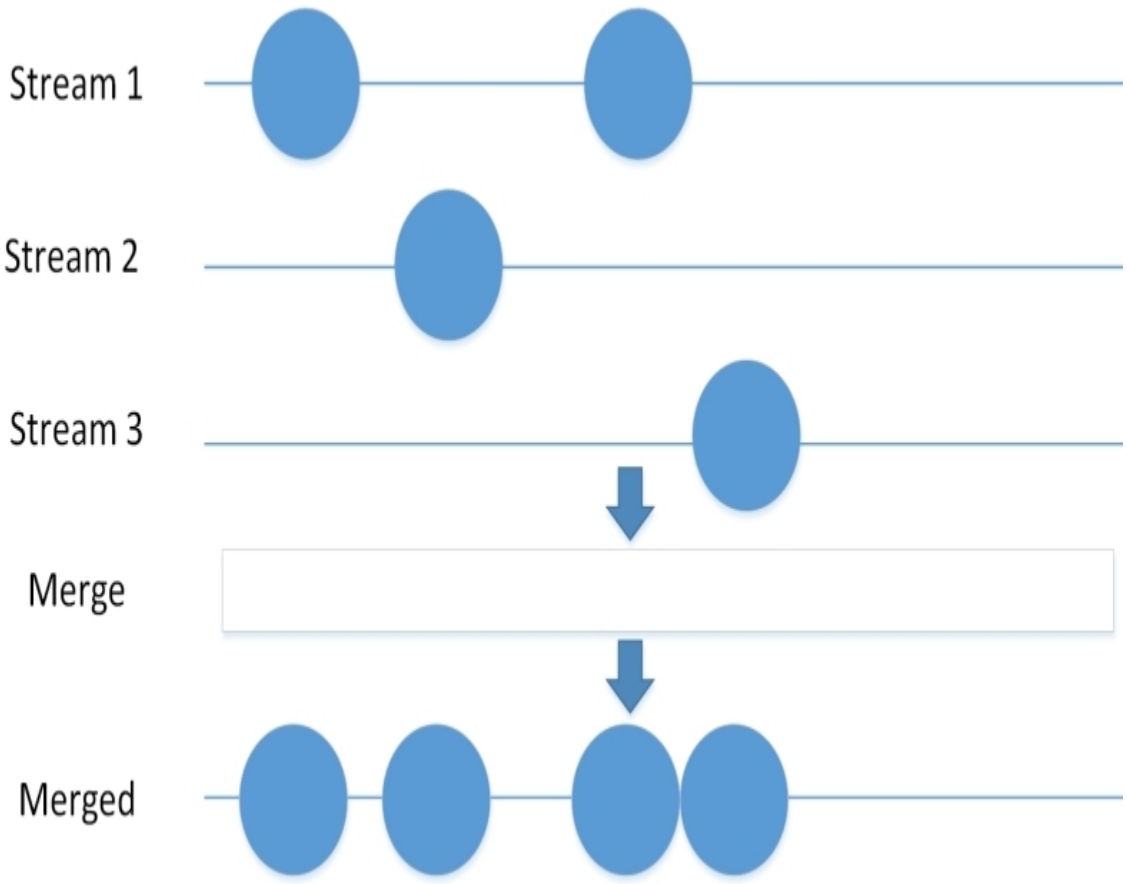
```
var counter = 0;
generateDeathsStream(deaths).subscribe((item) => { counter++ });
```

Merging streams

We've already seen the `zip` function that merges events one-to-one to create a new stream but there are numerous other ways of combining streams. A very simple example might be a page which has several code paths which all want to perform a similar action. Perhaps we have several actions all of which result in a status message being updated:

```
var button1 = document.getElementById("button1");
var button2 = document.getElementById("button2");
var button3 = document.getElementById("button3");
var button1Stream = Rx.Observable.fromEvent(button1, 'click');
var button2Stream = Rx.Observable.fromEvent(button2, 'click');
var button3Stream = Rx.Observable.fromEvent(button3, 'click');
var messageStream = Rx.Observable.merge(button1Stream, button2Stream,
button3Stream);
messageStream.subscribe(function (x) { return console.log(x.type + " on
" + x.srcElement.id); });
```

Here you can see how the various streams are passed into the merge function and the resulting merged stream:



While useful, this code doesn't seem to be particularly better than simply calling the event handler directly, in fact it is longer than necessary. However, consider that there are more sources of status messages than just button pushes. We might want to have asynchronous events also write out information. For instance, sending a request to the server might also want to add status information. Another fantastic application may be with web workers which run in the background
and communicate with the main thread using messaging. For web based JavaScript applications this is how we implement multithreaded applications. Let's see how that would look.

First we can create a stream from a worker role. In our example the worker simply calculates the fibonacci sequence. We've added a fourth button to our page and have it trigger the worker process:

```
var worker = Rx.DOM.fromWorker("worker.js");
button4Stream.subscribe(function (_) {
    worker.onNext({ cmd: "start", number: 35 });
});
```

Now we can subscribe to the merged stream and combine it with all the previous streams:

```
var messageStream = Rx.Observable.merge(button1Stream, button2Stream,
button3Stream, worker);
messageStream.subscribe(
    function (x) {
        appendToOutput(x.type + (x.srcElement.id === undefined ? " with
" + x.data : " on " + x.srcElement.id));
    },
    function (err) { return appendToOutput(err, true); }
);
```

This all looks really nice but we don't want to clobber the users with dozens of notifications at a time. We can throttle the stream of events so that only a single toast shows up at a time by using the same interval zip pattern we saw earlier. In this code we've replaced our `appendToOutput` method with a call to a toast display library:

```
var messageStream = Rx.Observable.merge(button1Stream, button2Stream,
button3Stream, worker);
var intervalStream = Rx.Observable.interval(5000);
messageStream
    .zip(intervalStream, function (x, _) {return x;})
    .subscribe(function (x) {
         toastr.info(x.type + (x.srcElement.id === undefined ? " with "
+ x.data : " on " + x.srcElement.id));
    },
    function (err) { return toastr.error(err); }
);
```

As you can see the code for this functionality is short and easy to understand yet it contains a great deal of functionality.

Streams for multiplexing

One does not rise to a position of power on the King's council in Westeros without being a master at building networks of spies. Often the best spy is one who can respond the quickest. Similarly, we may have some code which has the option of calling one of many different services which can fulfill the same task. A great example would be a credit card processor: it doesn't really matter which processor we use as they're pretty much all the same.

To achieve this, we can kick off a number of HTTP requests to each of the services. If we take each of the requests and put them into a stream, we can use it to select the fastest to respond processor and then perform the rest of the actions using that processor.

With RxJS this looks like the following:

```
var processors = Rx.Observable.amb(processorStream1, processorStream2);
```

You could even include a timeout in the `amb` call which would be called to handle the eventuality that none of the processors responded in time.

Best practices and trouble shooting

There are a large number of different functions that can be applied to streams. If you happen to decide on the RxJS library for your FRP needs in JavaScript, many of the most common functions have been implemented for you. More complex functions can often be written as a chain of the included functions so try to think of a way to create the functionality you want by chaining before writing your own functions.

Frequently, asynchronous calls across the network in JavaScript fail. Networks are unreliable, mobile networks doubly so. For the most part when the network fails, our application fails. Streams provide an easy fix to this by allowing you to easily retry failed subscriptions. In RxJS this method is called `Retry` . Slotting it into any observable chain makes it more resilient to network failures.

Summary

Functional reactive programming has many uses in different applications both on the server and on the client. On the client side it can be used to wrangle a large number of events together into a data flow enabling complex interactions. It can also be used for the simplest of things such as preventing a user from double-clicking a button. There is no huge cost to simply using streams for all of your data changes. They are highly testable and have a minimal impact on performance.

Perhaps the nicest thing about FRP is that it raises the level of abstraction. You have to deal with less finicky process flow code and can, instead, focus on the logical flow of the application.

In the upcoming chapters we are going to start explaining the classical design patterns and in the next chapter we are going to review the Creational Patterns.

Questions

1. What is a stream?
2. How can you merge two streams?
3. When it's useful to merge two streams?
4. Name a few reactive programming libraries?
5. What is an observable?
6. Is it possible to use reactive programming without is functional approach?
7. What is multiplexing and how you can use it together with streams?

Further Reading

To learn more about reactive programming with JavaScript I recommend you to read the book from PacktPub: https://www.packtpub.com/application-development/reactive-programming-javascript.

If you want to learn about reactive programming not related to JavaScript you can see these three pages in Wikipedia:

- https://en.wikipedia.org/wiki/Reactive_programming
- https://en.wikipedia.org/wiki/Reactive_extensions
- https://en.wikipedia.org/wiki/Functional_reactive_programming

Creational Patterns

Introduction

In the next three chapters we will review in depth the classic design patterns or also known as the 'Gang of Four' patterns, as they were four the authors of the first book to collect them all and written about. Classic design patterns have been created focused in object-oriented programming languages like Java, C++, C#. In this chapter we will look at and understand patterns known as Creational Patterns and how to use them in Javascript.

Patterns considered as Creational Patterns are those patterns that deal with the problem of creating objects that can cause design problems or end up adding more complexity to the system.

Every design pattern explained in this chapter is structured to understand, when to use and when not to use it to help you figure out what could work better in what situation.

The following topics will be covered in this chapter:

- Singleton
- Prototype
- Constructor
- Factory Method
- Abstract Factory

- Builder

Technical Requirements

- You need to know how to object-oriented programming works in Javascript (see Chapter 2)
- A modern browser is needed in order to execute the exercises
- For some exercises based in Node.js you will require to use `RunKit`
- You can find the exercises of this chapter in GitHub (`https://github.com/PacktPublishing/Mastering-JavaScript-Design-Patterns-Third-Edition/tree/master/Chapter05`)

Singleton

In JavaScript, unlike other languages that need a class, the simple creation of an object using literal notation generates an object that can be consider a single instance.

```
const obj = {};
```

The Singleton pattern is distinguished by not allowing over one instance of a particular object to exist in our system. Another feature is that it must be possible to access this instance from anywhere in the system.

When to use it?

Singleton is normally used when for specific reasons the system requires a single instance with the intention of being able to be used from anywhere in the system. Some examples of singleton are:

- Configuration objects
- Database connection objects (since the connection is only made once)

How to implement it?

In Javascript you will find different types of Singleton implementation from the most simple one using literal notation, using IIFE and other different ways but the most important is to implement a "lazy initialization" method.

In order to perform the lazy initialization you will need to set a private variable in your ES6 module where you will store the instance once it is created. The method should be a static method of the Singleton and it should check if there is already an instance or not. When the instance doesn't exist the instance should be created and returned but if it already exist the instance should be just returned to the user.

Example

Some usages of singleton are:

- Configuration objects
- Database connection objects (since the connection is only made once)

Example of database connection in Node.js.

```
let DBCInstance = null;
global.DatabaseConnection = class DatabaseConnection {
    get url() {
        return 'mongodb://localhost:27017/myproject';
    }
    get username() {
        return 'admin';
    }
    get connection() {
        let connection;
        // Do something to get the connection to the DB.
        return connection;
    }
    get password() {
        return 'localhost';
    }
    static get instance() {
        if(DBCInstance === null ||
           DBCInstance.getConnection().isClosed()) {
            DBCInstance = new DatabaseConnection();
        }
        return DBCInstance;
    }
}
```

```
DatabaseConnection.instance;
```

You can see how this example works in: https://runkit.com/tcorral/singleton.

When not to use it?

As we have seen in Chapter 2. 'Object-oriented programming' it's wrong to pollute globals and Singleton requires that.

The simple fact of considering using Singleton(s) in our system may imply a design error that needs to be addressed. Singleton, by itself, can become a problem many times because we are sharing an object in our system. This makes the test more complex and also hides the dependencies in our system.

Pros

Using class style Singleton you can implement interfaces, extend from others or be extended from.

Cons

- Hiding dependencies in your system.
- Making more difficult testing.
- Strong coupling between classes.

Alternative

Instead of using a Singleton you can pass the configuration object directly to the constructor of the class that needs it also called inversion of control.

> *If you want to keep your object data immutable you can use Object.freeze.*
>
> *https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/freeze*

Prototype

Do not get confused with the Javascript inheritance feature of the same name.

The Prototype design pattern is one in which a method is used in which through a model object or template we can obtain a clone identical to it, as far as the value of its data is concerned.

As you can see Prototype is very different to Singleton as far as Singleton only allows one instance and Prototype is created just the opposite, to create clones of object instances. The word that describes what the Prototype does is **clone** because it clones instances of objects.

Let's imagine we are told to create the first system on Earth that requires to start cloning human beings. The first thing we will need to do is scan all what determines that human being, skin color, height, weight, hair color. Once we have all this information we can use it to create a clone with the same characteristics this is what Prototype pattern does.

When to use it?

The Prototype design pattern is useful when:

- We have objects with a lot of information and we want to reuse it in a new object but we don't want to create one from scratch, being able to modify only the data not relevant for the new instance.
- We need to perform actions on some objects but we want to keep the original object intact.
- When creating a factory is over-engineering.
- It is better to copy an existing instance than to create a new one.

How to implement it?

In Javascript we can use `Object.assign` for a shallow clone or implement our deep clone method. We need to create a public clone method in our class so it will become the responsible of returning a clone of the same object.

If you want to learn more about Object.assign you can visit this link:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign

Example of Object.assign usage:

```
const urls = {
    packtpub: 'http://packtpub.com',
    google: 'https://www.google.com'
};
const urls2 = {
    wikipedia: 'http://en.wikipedia.org',
    mdn: 'https://developer.mozilla.org'
};

// Object.assign receives at least two arguments.
// .   1. First argument is the target object
//     2. Rest of arguments will be merged from right to left to the
target object
//     3. Returns the target object with everything merged.
// .   4. The source objects remain unmodified.

const target = Object.assign({}, urls, urls2);

/**
* {
*    packtpub: "http://packtpub.com",
*    google: "https://www.google.com",
*    wikipedia: "http://en.wikipedia.org",
*    mdn: "https://developer.mozilla.org"
*  }
*/
```

Example

There are quite good examples, of Prototype design pattern, can be found in DOM and a lot of different Javascript libraries and framework as jQuery or Angular to name a few.

> *DOM cloneNode implementation:* *https://developer.mozilla.org/en-US/docs/Web/API/Node/cloneNode*
>
> *jQuery clone implementation.* *https://api.jquery.com/clone/*
>
> *Angular extend implementation.* *https://docs.angularjs.org/api/ng/function/angular.extend*

The native way to clone objects in Javascript is `Object.assign` the difference with a deep clone is that it will only copy the properties that are set as enumerable or the own properties of the object.

```
class HumanBeing {
    constructor(config) {
        this.skinColor = config.skinColor;
        this.hairColor = config.hairColor;
        this.height = config.height;
        this.weight = config.weight;
        this.gender = config.gender;
        // And more data.
    }
}

const me = new HumanBeing({ skinColor: 'pale', hairColor: 'brown',
height:'173cm', weight: '100kg', gender: 'male'});

const clone = Object.assign(new HumanBeing({}), me);

me === clone; // false but the data they contain is the same.
```

Below you can see a more purist implementation of a clone method added to our HumanBeing class.

```
class HumanBeing {
    constructor(config) {
        this.skinColor = config.skinColor;
        this.hairColor = config.hairColor;
        this.height = config.height;
        this.weight = config.weight;
        this.gender = config.gender;
        // And more data.
    }
    clone() {
        return new HumanBeing(Object.assign({}, this));
    }
}

const me = new HumanBeing({ skinColor: 'pale', hairColor: 'brown',
height:'173cm', weight: '100kg', gender: 'male'});

const clone = me.clone();

me === clone; // false but the data they contain is the same.
```

When not to use it?

You should not use Prototype design pattern when:

- You have to many different objects to be cloned or re-created.
- The information in the object is not much and it's better to create a new instance from scratch.

Pros

- Allows to use objects as templates of themselves to reduce initialization time.
- It's a different way of implementing sub classing when dealing with complex objects.
- Less time on initializing new instances.

Cons

- Complex object with references to other objects can be very difficult to clone and even sometimes can lead to circular dependency and it will get in an infinite loop.

Constructor

Constructor design pattern is something that has been added to Javascript together with class. Before ES6 the constructor of a class-like object was the function itself but now we can use the constructor function to define the properties of the instantiated object.

The main difference between constructors in Javascript and other languages as Java is that in Java you can have multiple constructors and in Javascript you only have one.

When to use it?

If you use ES6> Javascript you will use it every time you create a new class.

Example

```
class Car {
    constructor(color, brand, doors, miles) {
        this.color = color;
        this.brand = brand;
        this.doors = doors;
        this.miles = miles;
    }
}

var myCar = new Car('blue', 'Mercedes', 2, 9000);
```

When not to use it?

You do not need to use it when you don't want to set any instance property or method.

Factory method

Sometimes is required to create different types of objects but we also don't know before running time what sort of object we will instantiate. Factory method solves this problem delegating the instantiation logic to the child classes.

When to use it?

When the user doesn't knows what object will be created or also when the decision of what class is dynamically decided at runtime.

How to implement it?

To implement a Factory Method it requires to have a very good knowledge of the domain.

In order to implement this design pattern to have to follow the following steps:

- Extract the common interface for each product.
- Create an abstract Factory class that needs to implement the creation of the product.
- If you were refactoring your code you will also need the instantiation of the product by invoking the new method you created in the previous point, if not invoke the method where the product should be created.
- Create one or more subclasses according with your needs and implement the method to create the product according to your needs.

Example

In order to understand the concept of Factory method we are going to create our own pizza store.

```
class PizzaStore {
  createPizza() {
    throw new Error("This method must be overwritten!");
  }

  orderPizza(type) {
    let pizza = this.createPizza(type);

    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
  }
}
```

As you can see the type of the pizza is defined on runtime just when the `orderPizza` method is invoked.

Below you can find the definition of pizza that contains the four methods used in our pizza store on ordering a pizza.

```
class Pizza {
  constructor({ name = '', dough = '', sauce = '', toppings = []}) {
    this.name = name;
    this.dough = dough;
    this.sauce = sauce;
    this.toppings = toppings;
  }

  prepare() {
    console.log("Preparing " + this.name);
```

```
    console.log("Tossing dough...");
    console.log("Adding sauce");
    console.log("Adding toppings:");

    for (let topping of this.toppings) {
      console.log(topping + "  ");
    }
  }

  bake() {
    console.log("Bake for 25 minutes at 350");
  }

  cut() {
    console.log("Cutting the pizza into diagonal slices");
  }

  box() {
    console.log("Place pizza in official PizzaStore box");
  }

  getName() {
    return this.name;
  }
}
```

The pizza we just created is a very generic class so that it can be use in our factory. Now it's time of creating our menu with a few pizzas of diverse taste.

```
class CheesePizza extends Pizza {
  constructor() {
    super({
      name: 'NY Style Sauce and Cheese Pizza',
      dough: 'Thin Crust Dough',
      sauce: 'Marinara sauce',
      toppings: ["Grated Reggiano Cheese"]
    });
  }
}
```

```
class ClamPizza extends Pizza {
  constructor() {
    super({
```

```
      name: 'NY Style Sauce and Clam Pizza',
      dough: 'Thin Crust Dough',
      sauce: 'Marinara sauce',
      toppings: ["Grated Reggiano Cheese", 'Clams']
    });
  }
}
```

Now we have two pizzas we can serve our customers we have to create our store because who knows maybe in the future we will have different pizza stores around the world.

```
import PizzaStore from '../../../common/PizzaStore';
import CheesePizza from '../pizzas/NewYorkStyle/CheesePizza';
import ClamPizza from '../pizzas/NewYorkStyle/ClamPizza';

const PIZZAS = {
  cheese: CheesePizza,
  clam: ClamPizza
};

class NewYorkPizzaStore extends PizzaStore {
  createPizza(type) {
    let PizzaConstructor = PIZZAS[type];
    let pizza = null;
    if (PizzaConstructor) {
      pizza = new PizzaConstructor();
    }
    return pizza;
  }
}

export default NewYorkPizzaStore;
```

We have set everything needed to start our pizza business lets' see how it works.

```
import NewYorkPizzaStore from './stores/NewYorkPizzaStore';

var oPizzaStore = new NewYorkPizzaStore();
oPizzaStore.orderPizza("cheese");
```

You can see it working in: https://runkit.com/tcorral/factory-method

When the factory is executed it checks what kind of pizza the customer ordered and then it gets the constructor of the pizza and that's pretty much it, the magic is done.

When not to use it?

When you don't need that complexity to create different objects or when it's even more complex and you also require to manage the things that conform each item.

Pros

- Hides instantiation complexity from client.
- Less coupling architecture between products and the code that uses them.
- Simplifies code because everything has been moved to a single place.
- Open to extension and closed to modification.

Cons

- Inheritance hell in progress due to the amount of extra subclasses required.

Abstract Factory

In the previous section we have introduced the concept of factory as a design pattern. In this section we are going to go a step forward because the Abstract Factory design pattern is a factory of factories. The factories involved required to be related between them.

When to use it?

When a single Factory method design pattern is not enough.

How to implement it?

As this is also a factory you will have a very good knowledge of the domain of your application.

You have to:

- Group the products by families of products.
- Every factory variant should be implemented in a separate factory class.
- You should implement the instantiation of the factory and pass it to the consumer object through the constructor.
- Replace all the calls to the product instantiation by the creation method of the factory.

Example

Continuing with our pizza store example now we are going to create a ingredients factory so we can get each ingredient but being managed by its own factory. The idea is to be able to track how many toppings we put on our pizzas.

Let's see what changes we need to do in our previous code.

```
class Pizza {
  constructor({ name = '', dough = '', sauce = '', veggies = [], cheese
= '', pepperoni = '', clams = '' }) {
    this.name = name;
    this.dough = dough;
    this.sauce = sauce;
    this.veggies = veggies;
    this.cheese = cheese;
    this.pepperoni = pepperoni;
    this.clams = clams;
  }

  // Methods remain the same.
}

export default Pizza;
```

Previous Pizza constructor definition was

```
class Pizza {
  constructor({ name = '', dough = '', sauce = '', toppings = []}) {
    this.name = name;
    this.dough = dough;
    this.sauce = sauce;
    this.toppings = toppings;
  }
```

```
  // Other methods.
}
```

Now every type of topping has been separated by its kind instead of putting them all together and this makes simpler to track what kind of topping is more used or if we are doing anything wrong.

In the next snippet of code you can see how we created an abstract class* to create different ingredients of our pizza.

> *In Javascript there is no way to create abstract classes as in other OOP languages but the most simple way to force the user or your class to extend it is to throw errors because if he/she doesn't implements it's own the extension will not be completed.*

```
class PizzaIngredientFactory {
  createDough() {
    throw new Error("This method must be overwritten!");
  }

  createSauce() {
    throw new Error("This method must be overwritten!");
  }

  createCheese() {
    throw new Error("This method must be overwritten!");
  }

  createVeggies() {
    throw new Error("This method must be overwritten!");
  }

  createPepperoni() {
    throw new Error("This method must be overwritten!");
  }

  createClam() {
    throw new Error("This method must be overwritten!");
  }
}

export default PizzaIngredientFactory;
```

At the previous code we could see the implementation of an abstract class that defined the methods required for a new ingredients factory so let's implement our own.

```
import PizzaIngredientFactory from '../PizzaIngredientFactory';
import ThinCrustDough from '../ingredients/ThinCrustDough';
import MarinaraSauce from '../ingredients/MarinaraSauce';
import ReggianoCheese from '../ingredients/ReggianoCheese';
import Garlic from '../ingredients/Garlic';
import Mushroom from '../ingredients/Mushroom';
import RedPepper from '../ingredients/RedPepper';

class NewYorkPizzaIngredientFactory extends PizzaIngredientFactory {
  createDough() {
    return new ThinCrustDough();
  }

  createSauce() {
    return new MarinaraSauce();
  }

  createCheese() {
    return new ReggianoCheese();
  }

  createVeggies() {
    return [new Garlic(), new Mushroom(), new RedPepper()];
  }

  createPepperoni() {}

  createClam() {}
}

export default NewYorkPizzaIngredientFactory;
```

Below you can see how we pass the `ingredientFactory` to the constructor of our pizza so we can easily change the recipe of our pizza only changing it from outside of our pizza class.

```
import Pizza from '../Pizza';

class CheesePizza extends Pizza {
```

```
  constructor(style, ingredientFactory) {
    super({
      name: style + ' Cheese Pizza'
    });
    console.log(this.name);
    this.ingredientFactory = ingredientFactory;
  }

  prepare() {
    let ingredientFactory = this.ingredientFactory;
    console.log("Preparing " + this.name);
    this.dough = ingredientFactory.createDough();
    this.sauce = ingredientFactory.createSauce();
    this.cheese = ingredientFactory.createCheese();
  }
}

export default CheesePizza;
```

After this line you will find how to create a product class.

```
class Garlic {}

export default Garlic;
```

See it running at: https://runkit.com/tcorral/abstract-factory

When not to use it?

If it can be solved using Factory Method design pattern only.

Pros

- Hides instantiation complexity from client.
- Less coupling architecture between products and the code that uses them.
- Easy to group the products in families keeping their compatibility.
- Different factories have different responsibility.

Cons

- Complexity increases because the multiple classes required to be created.

Builder

In order to understand this pattern we should look at Builder design pattern as those construction blocks we used when we were kids to build structures because this is actually what the pattern does. It's an implementation to allow create or build complex objects step by step or piece by piece.

When to use it?

When the complexity of creating an object can lead to a constructor telescoping anti-pattern. When one object can have different flavours or options that make impossible to construct it without having too many different ways to construct the object.

What the telescoping constructor anti-pattern is

In Javascript is not so marked as in other languages as Java because in Java you have to create a different constructor per each different amount or type of parameters you receive while in Javascript you only work with one constructor.

The problem even not being the same can give problems because if there is something to be added the class is required to be modified to add the new flavours or options.

Example of telescoping constructor anti-pattern:

```
class Meal {
 constructor(id, name, calories = 0, servingSize = 0, fat = 0,
description = 'default description') {
 this.id = id;
 this.name = name;
 this.calories = calories;
 this.servingSize = servingSize;
 this.fat = fat;
 this.description = description;
 }
}

var meal1 = new Meal(1, 'Dutch meal');
var meal2 = new Meal(2, 'British meal', 6, 9);
var meal3 = new Meal(3, 'Spanish meal', 8, 5, 35);
```

How to implement it?

In order to be successful with this pattern you need a very good knowledge of the domain of the system.

You have to:

- Abstract the common steps of building the product.
- Abstract the variations of each step that lead to each different possibility of creating a product.
- Create the Builder abstract class and declare production steps.
- Create each concrete builder and implement the construction steps defined in the abstract Builder class
- Create a builder director to use builder to create different product flavours.

Example

To illustrate this design pattern we are going to use a real example where we are the proud owners of a restaurant that is specialized in burgers.

As we want to have a wide variety of clients we have vegan, regular and deluxe meals. Every menu or meal have different sort of products but in the end the customer must get a burger with a side dish and drink.

We have to create our abstract class `Item` that uses getters.

```
class Item {
    get name() {
        throw new Error('This method should be implemented');
    }
    get packing() {
        throw new Error('This method should be implemented');
    }
    get price() {
        throw new Error('This method should be implemented');
    }
}
```

As our product has a different kind of packaging we have to implement the abstract class `Packing`.

```
class Packing {
    pack() {
        throw new Error('This method should be implemented');
    }
}
```

Once we have the abstract classes we have to create the implementation of our products and type of packaging.

Packing types:

```
class Wrapper extends Packing {
    pack() {
        return 'Wrapping';
    }
}

class BoxUp extends Packing {
    pack() {
        return 'Boxing';
    }
}

class Bottle extends Packing {
    pack() {
        return 'Bottle';
    }
}
```

Item types:

```
class Burger extends Item {
    get packing() {
        return new Wrapper();
    }
}

class SideDishes extends Item {
    get packing() {
        return new BoxUp();
    }
}

class Drink extends Item {
    get packing() {
        return new Bottle();
    }
}
```

In the previous code you can see how each main item sets the type of packing but this is only the generic one so we have to implement the specific ones.

```
class Water extends Drink {
    get price() {
        return 2.5;
    }
    get name() {
        return 'Water';
    }
}

// Other drink implementations

class Salad extends SideDishes {
    get price() {
        return 6;
    }
    get name() {
        return 'Salad';
    }
}

// Other side dish implementation

class VeganBurger extends Burger {
    get price() {
        return 16;
    }
    get name() {
        return 'Vegan Burger';
    }
}

// Other burger implementations
```

Below we create the Meal class to add the items, show the items and get the final cost.

```
const itemList = new WeakMap();
class Meal {
    get list() {
```

```
        if(!itemList.get(this)) {
            itemList.set(this, []);
        }
        return itemList.get(this);
    }
    addItem(item) {
        this.list.push(item);
    }
    getCost() {
        return this.list.reduce((accum, item) => {
            accum += item.price;
            return accum;
        }, 0);
    }
    showItems() {
        this.list.forEach((item) => {
            console.log(`Item: ${item.name}, Packing:
${item.packing.pack()}, Price: $${item.price}`);
        });
    }
}
```

The next step is to create the builder director with a method per each type of product flavour we need to create, in our case each builder method should create a new menu meal.

```
class MealBuilder {
    prepareVeganMeal() {
        var meal = new Meal();
        meal.addItem(new VeganBurger());
        meal.addItem(new Water());
        meal.addItem(new Salad());
        return meal;
    }
    prepareNonVeganMeal() {
        var meal = new Meal();
        meal.addItem(new BeefBurger());
        meal.addItem(new Coke());
        meal.addItem(new Fries());
        return meal;
    }
    prepareDeluxeMeal() {
        var meal = new Meal();
        meal.addItem(new KobeBurger());
        meal.addItem(new Champagne());
```

```
        meal.addItem(new Crudettes());
        return meal;
    }
}
```

See it working at: https://runkit.com/tcorral/builder-design-pattern

When not to use it?

When there is no need for different steps to create your product.

> *If you have to create a product that requires to create the product adding ingredients it's better if you use Decorator Pattern. See next chapter about the implementation.*

Pros

- Building products step by step.
- Reuse same code to build different products.
- No more construction complexity isolating it from business logic.

Cons

- Due to the quantity of classes required it increases the complexity of your code.

Summary

In this chapter we have explained and implemented all the different Creational Design Patterns. With everything we learned in this chapter we will reduce the complexity of our system.

You have learn how to use Singleton to create a common entry point for every piece in the system but also when not to use it. We have used Prototype to create a clone of our objects. The two factory patterns have simplified what could be very difficult to do if we don't know what type of class we are going to use. At the end of the chapter we used Builder to understand how to create a product step by step.

In the next chapter we will look at Structural Design Patterns that ease our work by identifying a simple way to realize relationships between entities.

Questions

- What problem solves a Creational Design Pattern?
- When to use Singleton?
- When not to use Singleton?
- What is the alternative to using Singleton?
- When to use Factory Method?
- When not to use Factory Method?
- When to use Abstract Factory?
- When not to use Abstract Factory?
- When to use Builder?
- When not to use Builder?

Further Reading

- To get deep into ES6 you can read this book. https://www.packtpub.com/web-development/learning-ecmascript-6
- In order to know more about the classic design patterns and the GoF you can visit: http://wiki.c2.com/?GangOfFour
- To understand better what is the background of Creational pattern visit: https://en.wikipedia.org/wiki/Creational_pattern

Structural Patterns

Introduction

In this chapter we will review the Structural Design Patterns of the "Gang of Four". Structural Design Patterns are those patterns that which responsibility is to ease the design of a system by building simple and efficient hierarchies and relations between entities .

Some times is difficult dealing with how to structure our application besides how you organize your project in folders you also need to identify relationships between the objects in your system. Structural design patterns are these patterns that have demonstrated being efficient on managing efficiently these sort of problems.

Every design pattern is structured to understand when to use and when not to use it to help you figure out what could work better in what situation.

The following topics will be covered in this chapter:

- Adapter
- Composite
- Decorator
- Bridge
- Façade
- Flyweight

- Proxy

Technical Requirements

- You need to know how to object-oriented programming works in Javascript (see Chapter 2)
- A modern browser is needed in order to execute the exercises.
- For some exercises based in Node.js you will require to use RunKit (`https://runkit.com/home`)
- You can find the exercises of this chapter in GitHub (`https://github.com/PacktPublishing/Mastering-JavaScript-Design-Patterns-Third-Edition/tree/master/Chapter06`)

Adapter

The Adapter pattern allows objects with incompatible interfaces to collaborate.

An example of an adapter in the real world is using a socket plug adapter when we travel around the world.

When to use it?

The idea behind Adapter pattern is to adapt an existing interface of a class but without changing it.

- When a client requires an interface to perform a task by creating a class that converts the incompatible interface into the expected interface.
- When you want to reuse an existing class but using it in your system is not possible because it's not compatible.

How to implement it?

This design patterns requires of two actors:

- A class we need in our system and that we cannot modify.
- Our system should not be able to use the class we need because incompatible interface or data formats.

Once you have detected the two actors you have to:

- Analyze what is the interface the client expects so that we will use it to create our adapter class later on.
- Create an adapter class:
    - Setting a property that stores a reference to the, incompatible, service object.
    - Implementing the interface the client expects performing changes accordingly to the expectations.
- Use the adapter class as our client interface.

Example

The most common usage of Adapter is transforming data making service classes compatible with any system.

In our example we are a company that used to work with different duck species but this year we want to open our business to work with other different types of bird.

Our system is used to work with ducks so it makes uses of two methods we can find in alive ducks species `fly` and `quack` .

Let's see how our Duck abstract class was implemented:

```
class Duck {
    constructor() {}
    fly() {
        throw new Error('This method must be overwritten!');
     }
     quack() {
     throw new Error('This method must be overwritten!');
     }
}

export default Duck;
```

Example of Turkey adapter .

```
import Duck from './Duck';

const MAX_FLIES = 5;

class TurkeyAdapter extends Duck {
  constructor(oTurkey) {
    super(oTurkey);
```

```
    this.oTurkey = oTurkey;
  }

  fly() {
    for (let index = 0; index < MAX_FLIES; index++) {
      this.oTurkey.fly();
    }
  }

  quack() {
    this.oTurkey.gobble();
  }
}

export default TurkeyAdapter;
```

The adapter class should extend of our abstract class and implement the method by it's own.

You can see how this example works in: https://runkit.com/tcorral/adapter

When not to use it?

When not need reusing existing classes in your system.

Pros

- Hiding unnecessary implementation details.

Cons

- Increases code complexity because it requires extra classes.

Composite

As its own says, the Composite Pattern intent is to compose complex objects using other simpler objects into tree structures allowing the client(s) to treat these structures as if they were individual objects.

An example of Composite pattern in web development is the creation of a form using smaller parts like buttons, input controls.

Another real-world example of Composite pattern is a car where it's created by joining different pieces that have identity by themselves.

When to use it?

Composite pattern is useful when you have to implement a tree-like structure where we create complex components using simpler ones.

How to implement it?

- Review your business logic and check if it can be represented as a tree structure.
- Break into simple elements and containers. These containers can contain simple elements as other containers.
- Describe an interface for all the parts a.k.a Components. The interface should have sense to be used in the same way by simple and complex elements.
- Create an element class representing simple components.
- Create a container class.
    - Should have a property to store subcomponents.
    - It should delegate most of the actions to subcomponents.
    - Implement methods to add or remove child elements to the container.

Example

To explain how to use Composite pattern we are going to create a system for a restaurant where we use the pattern to implement a system to deal with different menus and it's composition.

`MenuComponent` is the basic class that implements the basic interface that can be used by simple and complex elements.

```
class MenuComponent {
  constructor(name = '', description = '', isVegetarian = false, price
= 0) {
    this.name = name;
    this.description = description;
    this._isVegetarian = isVegetarian;
    this.price = price;
  }

  getName() {
    return this.name;
  }

  getDescription() {
    return this.description;
  }

  getPrice() {
    return this.price;
  }

  isVegetarian() {
    return this._isVegetarian;
  }

  print() {
    shouldBeOverwritten();
  }
```

```
  add() {
    shouldBeOverwritten();
  }

  remove() {
    shouldBeOverwritten();
  }

  getChild() {
    shouldBeOverwritten();
  }
}

export default MenuComponent;
```

In the Composite pattern all the rest of the elements, simple and complex, will require to extend from this abstract class.

`Menu` is our basic container class and it extends from `MenuComponent` implementing the variable to store the components as well as the add, remove or get component methods. The `Menu` class will be used to create other subclasses of menu.

```
import MenuComponent from './MenuComponent';

class Menu extends MenuComponent {
  constructor(name, description) {
    super();
    this.menuComponents = [];
    this.name = name;
    this.description = description;
  }

  add(menuComponent) {
    this.menuComponents.push(menuComponent);
  }

  remove(menuComponent) {
    this.menuComponents = this.menuComponents.filter(component => {
      return component !== component;
    });
  }
```

```
  getChild(index) {
    return this.menuComponents[index];
  }

  getName() {
    return this.name;
  }

  getDescription() {
    return this.description;
  }

  print() {
    console.log(this.getName() + ": " + this.getDescription());
    console.log("-----------------------------------------");
    this.menuComponents.forEach(component => {
      component.print();
    });
  }
}

export default Menu;
```

MenuItem is our simple element class that will be used to create more specific menu items.

```
import MenuComponent from './MenuComponent';

class MenuItem extends MenuComponent {
  print() {
    console.log(this.name + ": " + this.description + ", " + this.price
+ "euros");
  }
}

export default MenuItem;
```

You can see this example working at: https://runkit.com/tcorral/composite

When not to use it?

Composite pattern should not be used when:

- The system cannot be understood as a tree.
- Because the infinite possibilities of construction using Decorator pattern is a simpler solution.

Pros

- Simplifies client code on interacting with a complex tree structure.
- Easy way to add new component types.

Cons

- Requires to create too generic classes

Decorator

Decorator pattern is a pattern that use wrapping to add behaviours to existing objects.

One of the most common problems that Decorator pattern can solve is when in one system the different possibilities of creating objects can become infinite.

A real-world example can be an ice cream shop where there are many different syrups of different flavours as well as a bunch of different ice cream flavours and depending of the customer there are infinite possibilities that you will never predict in advance.

When to use it?

When using Composite pattern is not enough because your system can not be understood as a tree and you have to add and or remove responsibilities to or from an object dynamically keeping these objects compatible with the rest of your system code. Note that because the complexity you cannot use inheritance to create the different use cases.

How to implement it?

In Decorator pattern every object acts as a decorator or wrapper.

- Review your system and detect what is the element that can be represented as the primary component as well as all the other optional extensions.
- Create a `Component` abstract class describing all common methods for the primary component and extensions.
- Every concrete implementation of component should extend from the `Component` class and implement the methods.
- Create the common decorator class that will be used to create the extensions.
- Each decorator should receive by the constructor and store the basic element instance it is extending.

Example

In order to understand better the Decorator Pattern and how this pattern can simplify a complex system we are going to create the Point of Sale Terminal of a coffee shop where the customer chooses a beverage and then he or she can decide if he wants to add sugar, milk...

The first we need to create is a `Beverage` class because in our coffee shop we can serve coffee, tea or any other type of beverage.

```
class Beverage {
  constructor(description = 'Unknown beverage') {
    this.description = description;
  }

  getDescription() {
    return this.description;
  }

  cost() {
    throw new Error("This method must be overwritten!");
  }
}

export default Beverage;
```

In the Decorator Pattern any element of the system should exist from the basic item as you can see in our `CondimentDecorator`.

```
import Beverage from './Beverage';

class CondimentDecorator extends Beverage {}
```

```
export default CondimentDecorator;
```

Now we can create the basic beverage types:

Espresso implementation:

```
import Beverage from './Beverage';

class Espresso extends Beverage {
  cost() {
    return 1.99;
  }
}

export default Espresso;
```

House blend implementation:

```
import Beverage from './Beverage';

class HouseBlend extends Beverage {
  cost() {
    return 0.89;
  }
}

export default HouseBlend;
```

Once we have our basic beverages we can implement the extensions or condiments.

Mocha condiment implementation:

```
import CondimentDecorator from './CondimentDecorator';

class Mocha extends CondimentDecorator {
  constructor(beverage) {
    super();
    this.beverage = beverage;
```

```
  }

  getDescription() {
    return this.beverage.getDescription() + ", Mocha";
  }

  cost() {
    return 0.20 + this.beverage.cost();
  }
}

export default Mocha;
```

Whip condiment implementation:

```
import CondimentDecorator from './CondimentDecorator';

class Whip extends CondimentDecorator {
  constructor(beverage) {
    super();
    this.beverage = beverage;
  }

  getDescription() {
    return this.beverage.getDescription() + ', Whip';
  }

  cost() {
    return 0.60 + this.beverage.cost();
  }
}

export default Whip;
```

You can see this example working at: https://runkit.com/tcorral/decorator

When not to use it?

- If using Composite is a better option because the system is becoming complex enough to create a tree, in this case use Composite pattern instead.
- When there are not many different options and they can be handled by specific classes using or not inheritance.

Pros

- Simplify your class system.
- Is more flexible than using inheritance.
- Using wrappers/decorators can add or remove behaviours at runtime.
- It's very good option to compose complex objects from simple ones.

Cons

- A lot of small classes are needed.
- If you need to create a very complex object using decorators can become very hard because all the interactions between classes.

Bridge

Bridge Pattern main usage is to decouple the abstraction from its implementation. Using Bridge pattern allows to change the abstraction and the implementation independently.

When to use it?

Bridge pattern is designed to deal with different structures where your system has one problem with the basic classes and it's different implementations. If we have a set of classes but implementing the different use cases requires to create a class per each class multiplied by the different implementations.

A real-world example it's for example a car builder company where we have different car models. The car models can be delivered to the customer in a different color according to the taste of the customer.

How to implement it?

- Determine what is the client or basic item abstracting anything else not relevant.
- Anything that was not relevant for the client will become different use cases for the client.
- Create concrete implementation classes for each abstraction.
- Create an Abstraction class.
    - Storing the implementation type.
    - Implementing all the abstraction methods.

Example

Following with our example of cars and colors we are going to develop our system we will use to deliver to the customer the same car painted with a different color becoming a different car at the end.

First, as we know that our class will be painted with different colors we need to create our basic `Color` class because its interface will be implemented in the car classes.

```
class Color {
    applyColor() {
        throw new Error("This method should be overwritten");
    }
}

export default Color;
```

Once we have the `Color` class we need to create our basic `Car` class. Note that we pass the color instance through the constructor as well as declaring the `applyColor` method.

```
class Car {
    constructor(name, description, price, places = 2, color, brand =
'Cartisfaction') {
        this.brand = brand;
        this.name = name;
        this.description = description;
        this.price = price;
        this.places = places;
        this.color = color;
    }
    applyColor() {
        console.log(`${this.name} car painted with color
```

```
${this.color.applyColor()}`);
    }
}

export default Car;
```

Our company has three different cars to cover almost all the different customers in the market.

- Family car is our car to enjoy with your family
- 4 x 4 adventure car is our car for all the people that likes the adventure and doesn't care about creating new paths
- Urban car is our car for all the other needs. Is a small car that saves money to it's owner and that was designed to be, mainly, used at the city.

Family car implementation:

```
import Car from './Car';

class FamilyCar extends Car {
    constructor(color) {
        super('Family car', 'Enjoy with your family', 30000, 5, color);
    }
}

export default FamilyCar;
```

4 x 4 adventure car implementation:

```
import Car from './Car';

class Adventure4x4Car extends Car {
    constructor(color) {
```

```
        super('4x4 Adventure car', 'For people that does not care about
existing paths', 55000, 2, color);
    }
}

export default Adventure4x4Car;
```

Urban car implementation:

```
import Car from './Car';

class UrbanCar extends Car {
    constructor(color) {
        super('Urban car', 'Small and designed for the city', 12000, 2,
color);
    }
}

export default UrbanCar;
```

Once we have all our car classes we have to implement the different color catalog we want to offer to our customers to purchase his or her car according to his or her taste.

In the next three snippets of code you can see two color implementations.

For readability's sake we are going to implement one color only because the rest of them are too similar to add any relevant information. You can find the rest of implementations in the code repository.

```
import Color from './Color';

class RedColor extends Color {
    applyColor() {
        return 'red';
    }
}
```

```
export default RedColor
```

If you want to see how it actually works visit: https://runkit.com/tcorral/bridge

When not to use it?

When you don't have a clear differentiation between abstraction of client classes and implementations.

Pros

- Using Open/Closed principle
- Hiding implementation details
- Separate client and implementation classes.

Cons

- Needing to add a lot of different classes adding more complexity to the system.

Facade

When you have to deal with a very complex system Facade pattern can be your solution. Facade provides a simple interface to interact with a complex service.

A real-world example of Facade pattern used often at home of many people is a home theater system but instead of dealing with different remote per each interface we use only one from which we can do everything needed to set it up or close it.

When to use it?

When you need to lead with a complex subsystem but through a simple limited interface or when you want to use layers to structure your subsystem.

How to implement it?

- Detect what is needed to setup every element in the complex system.
- Abstract what is needed to be done per each item in the system in order to accomplish a task.
- Create a Facade class.
    - Storing each element from the subsystem.
    - Implement methods where we execute any needed action on the elements to perform a common task.

Example

The example to illustrate Facade pattern is a Home Theater at home were we want to use a single remote to perform everything needed to :

- Watch a movie
- Listen our MP3 playlist.
- Listen to the radio.

First, we have to identify every single element of the system.

In our system we have a bunch of elements:

- Amplifier
- MP3 Player
- BlueRay Player
- Popcorn Popper
- Projector
- Screen
- Theater Lights
- Tuner

When we analize every element we detect that all of them are elements that can be switch on/off and some of them are players. So we abstract those two behaviors in two different abstract classes.

As we need to implement two different behaviors we will use a multi-inheritance design pattern.

Implementation of `Switchable` abstract class:

```
const Switchable = Sup => class extends Sup {
  on() {
    throw new Error('This method should be overwritten!');
  }

  off() {
    throw new Error('This method should be overwritten!');
  }
};

export default Switchable;
```

Switchable is not a class by itself. It's a function that will receive a class as an arguments and then we use anonymous class, that is not possible in other languages like Java or C#. The class extends the one we pass and returns the new abstract class. This pattern will be also used for `Playable` abstract class.

Implementation of `Playable` abstract class:

```
const Playable = Sup => class extends Sup {
  eject() {
    throw new Error('This method should be overwritten!');
  }

  play() {
    throw new Error('This method should be overwritten!');
  }
```

```
  stop() {
    throw new Error('This method should be overwritten!');
  }
};

export default Playable;
```

Now we have to define each element extending these behaviors.

For the sake of the simplicity I will only show two different implementations, one implementing Switchable and another one implementing Switchable and Playable behavior. The rest of the code will be available in the Github repository.

Implementation of an element that is not a player.

```
import Switchable from './Switchable';

class Amplifier extends Switchable(null) {
  constructor() {
    super();
    this.volume = 0;
    this.dvdPlayer = null;
    this.cdPlayer = null;
    this.tuner = null;
    this.surroundSound = false;
    this.stereoSound = false;
  }

  on() {
    console.log("Amplifier is on!");
  }

  off() {
    console.log("Amplifier is off!");
  }

  setVolume(volume) {
    this.volume = volume;
    console.log("Volume change to " + volume);
  }

  setDvdPlayer(dvdPlayer) {
```

```
    this.dvdPlayer = dvdPlayer;
    console.log("Plugged DVD Player to Amplifier!");
  }

  setCdPlayer(cdPlayer) {
    this.cdPlayer = cdPlayer;
    console.log("Plugged Cd Player to Amplifier!");
  }

  setTuner(tuner) {
    this.tuner = tuner;
    console.log("Plugged on Tuner to Amplifier!");
  }

  setSurroundSound() {
    this.surroundSound = true;
    console.log("Surround Mode is active!");
  }

  setStereoSound() {
    this.stereoSound = true;
    console.log("Stereo Mode is active!");
  }
}

export default Amplifier;
```

Implementation of a player element.

```
import Switchable from './Switchable';
import Playable from './Playable';

class BlueRayPlayer extends Switchable(Playable(null)) {
  on() {
    console.log("BlueRay player is on!");
  }

  off() {
    console.log("BlueRay player is off!");
  }

  eject() {
    console.log("Eject BlueRay!");
  }

  play(movie) {
```

```
    console.log("Playing " + movie.name);
  }

  stop() {
    console.log("Stop BlueRay player!");
  }
}

export default BlueRayPlayer;
```

Once we have every element setup implement the needed behavior we have to work in our Facade.

```
class HomeTheaterFacade {
  constructor({ amplifier = null, tuner = null, brPlayer = null,
mp3Player = null, projector = null, theaterLights = null, screen =
null, popcornPopper = null}) {
    this.amplifier = amplifier;
    this.tuner = tuner;
    this.brPlayer = brPlayer;
    this.mp3Player = mp3Player;
    this.projector = projector;
    this.theaterLights = theaterLights;
    this.screen = screen;
    this.popcornPopper = popcornPopper;
  }

  watchMovie(movie) {
    console.log('Get ready to watch a movie...');

    this.popcornPopper.on();
    this.popcornPopper.pop();

    this.theaterLights.off();

    this.screen.down();

    this.projector.on();
    this.projector.setWideScreenMode();

    this.amplifier.on();
    this.amplifier.setDvdPlayer(this.dvdPlayer);
    this.amplifier.setSurroundSound();
    this.amplifier.setVolume(5);
```

```
    this.brPlayer.on();
    this.brPlayer.play(movie);
  }

  endMovie() {
    console.log("Shutting movie theater down...");
    this.popcornPopper.off();

    this.theaterLights.on();

    this.screen.up();

    this.projector.off();

    this.amplifier.off();

    this.brPlayer.stop();
    this.brPlayer.eject();
    this.brPlayer.off();
  }

  listenToMP3(playlist) {
    console.log("Start listening your music...");

    this.amplifier.on();
    this.amplifier.setCdPlayer(this.cdPlayer);
    this.amplifier.setStereoSound();
    this.amplifier.setVolume(5);

    this.mp3Player.on();
    this.mp3Player.play(playlist);
  }

  endMP3() {
    console.log("End listening your music or the Cd has finished!");

    this.amplifier.off();

    this.mp3Player.stop();
    this.mp3Player.off();
  }

  listenToRadio() {
    console.log("Start listening your favorite radio station...");

    this.amplifier.on();
    this.amplifier.setTuner(this.tuner);
    this.amplifier.setStereoSound();
```

```
    this.amplifier.setVolume(5);

    this.tuner.on();
    this.tuner.setFm();
    this.tuner.setFrequency(90.9);
  }

  endRadio() {
    console.log("End listening your favorite radio station...");

    this.amplifier.off();

    this.tuner.off();
  }
}

export default HomeTheaterFacade;
```

As you can see in our Facade we have implemented very basic actions and as a user of this system we don't need to know what is happening in the background we only have to press a button and wait for everything being setup for us.

If you want to see this example running you can visit: https://runkit.com/tcorral/facade

When not to use it?

When the system doesn't have complex subsystems or when the system is not complex enough.

Pros

- Clients don't know about subsystem components
- Less coupling between client and subsystem.

Cons

- There is a high risk of create a God object with a lot of coupling to all application classes.

Flyweight

Flyweight Pattern main purpose is to reduce the RAM memory usage by abstracting common parts related to the state allowing to fit more objects in the same amount of memory.

A real-world usage of Flyweight pattern is on Gaming due to the quantity of resources can be required specially if the game is multi-player and requires 3D rendering.

When to use it?

When the quantity of RAM memory is going to low due to massive quantity of objects with a lot of changes in the state.

When because the huge amount of objects the memory can not support it.

How to implement it?

- Analyze your objects and differentiate between fields with different states:
    - Unchanged data that can be shared across all the objects. (Common between objects) a.k.a Intrinsic data
    - Data that change in any object one or more time. (Unique data) a.k.a. Extrinsic data
- Leave the intrinsic data in the class but make it immutable. They can only be set in the moment of the instantiation, through the constructor.
- Move the extrinsic data to the arguments of the classes that referred to them.
- Create a Flyweight factory class. Objects should be requested through this factory and before returning it must be checked if it already exist.
- Context should be calculated and stored by the clients to be able to call methods of Flyweight objects.

Example

Following our example of real-world we are going to create a system that will create and render a whole forest with a huge amount of trees.

First, we create the Tree class that contains the extrinsic data, data that changes per each object.

```
const privateX = new WeakMap();
const privateY = new WeakMap();
const privateTreeType = new WeakMap();

class Tree {
    constructor(x = 0, y = 0, treeType) {
        privateX.set(this, x);
        privateY.set(this, y);
        privateTreeType.set(this, treeType);
    }
    get x() {
        return privateX.get(this);
    }
    get y() {
        return privateY.get(this);
    }
    get treeType() {
        return privateTreeType.get(this);
    }
    render(canvas) {
        const context = canvas.getContext("2d");
        this.treeType.render(context, this.x, this.y);
    }
}

export default Tree;
```

Now we have to create the TreeType class that will contain the intrinsic data, data that doesn't change between objects.

```
const privateName = new WeakMap();
const privateColor = new WeakMap();
const privateTreeConfig = new WeakMap();

class TreeType {
    constructor(name, color, treeConfig) {
        privateName.set(this, name);
        privateColor.set(this, color);
        privateTreeConfig.set(this, treeConfig);
    }
    get name() {
        return privateName.set(this);
    }
    get color() {
        return privateColor.set(this);
    }
    get treeConfig() {
        return privateTreeConfig.set(this);
    }
    render(context, x, y) {
        context.fillStyle = "black";
        context.fillRect(x - 1, y, 3, 5);
    }
}

export default TreeType;
```

Once we have the intrinsic data and extrinsic data stored in both different classes we have to encapsulate the creation of flyweight elements in a factory.

```
import TreeType from './TreeType';

const treeTypesMap = new Map();
class TreeFactory {
    static treeTypes() {
        return treeTypesMap;
    }
    static getTreeType(name, color, treeConfig) {
        let result = TreeFactory.treeTypes.get(name);
```

```
        if(result === null) {
            result = new TreeType(name, color, treeConfig);
            treeTypes.set(name, result);
        }
        return result;
    }
}

export default TreeFactory;
```

Finally we have to create our class to render a forest making use of our Flyweight pattern.

```
import TreeFactory from './TreeFactory';
import Tree from './Tree';

const privateTrees = new WeakMap();
class Forest {
    constructor() {
        privateTrees.set(this, []);
    }
    get trees() {
        return privateTrees.get(this);
    }
    plantTree(x, y, name, color, treeConfig) {
        const type = TreeFactory.getTreeType(name, color, treeConfig);
        const tree = new Tree(x, y, type);
        this.trees.push(tree);
    }
    render(canvas) {
        this.trees.forEach((tree) => {
            tree.render(canvas);
        });
    }
}

export default Forest;
```

On rendering 99900 trees the memory usage is improved a lot:

"Total: 0.7622337341308594MB (instead of 3.6203384399414062MB)"

To see this example running visit: https://runkit.com/tcorral/flyweight

When not to use it?

- When RAM memory usage is not a problem and when your system needs to deal with a huge amount of objects in memory.

Pros

- Saves RAM.
- Allows supporting more objects.

Cons

- CPU usage increase due to the context resolution
- More complexity due to the amount of classes needed to implement the pattern.

Proxy

Proxy pattern is an object that will be between the client and the real object.

A real-world of example can be a VPN in our system we use to connect to our company network from a different one.

When to use it?

- When it's needed to access to a heavy object that has access to the filesystem, database or network.
- When you have to execute locally a remote service.
- When you want to keep a history of request to a service object.

How to implement it?

- Analyze the real object and abstract the interface you want to expose.
- Create a proxy class and implement the interface so you can use it to emulate the use of the real object.
- If the object is heavy you have to consider use lazy initialization.

Example

For this pattern we are going to implement two different Proxy Pattern

- Proxy
- Virtual Proxy.

Our example will be used to emulate a public library.

In the next piece of code we have to deal with a heavy object called `PublicLibrary` this object has access to a lot of information, catalogs of data and use it from our system will have a performance impact on it.

```
class PublicLibrary {
  constructor(books) {
    this.catalog = {};
    this.setCatalogFromBooks(books);
  }

  setCatalogFromBooks(books) {
    books.forEach(book => {
      this.catalog[book.getIsbn()] = {
        book: book,
        available: true
      };
    });
  }

  findBooks(query) {
    console.log("Enter findBooks PublicLibrary");
    let results = [];
    for(let book of this.catalog) {
```

```
      if (query.match(book.getTitle()) ||
query.match(book.getAuthor())) {
        results.push(book);
      }
    }
    return results;
  }

  checkoutBook(book) {
    let isbn = book.getIsbn();
    book = this.catalog[isbn];
    if(book) {
      if(book.available) {
        book.available = false;
        return book;
      } else {
        throw new Error('PublicLibrary: book ' + book.getTitle() + ' is
not currently available.');
      }
    } else {
      throw new Error('PublicLibrary: book ' + book.getTitle() + ' not
found.');
    }
  }

  returnBook(book) {
    let isbn = book.getIsbn();
    book = this.catalog[isbn];
    if(book) {
      book.available = true;
    } else {
      throw new Error('PublicLibrary: book ' + book.getTitle() + ' not
found.');
    }
  }
}

export default PublicLibrary;
```

The implementation of a simple proxy requires to import the original object and store it in our proxy class so that we can interact with it.

```
import PublicLibrary from '../../common/PublicLibrary';
```

```
class PublicLibraryProxy {
  constructor(catalog = []) {
    this.library = new PublicLibrary(catalog);
  }

  findBooks(query) {
    console.log("Enter findBooks PublicLibraryProxy");
    return this.library.findBooks(query);
  }

  checkoutBook(book) {
    return this.library.checkoutBook(book);
  }

  returnBook(book) {
    return this.library.returnBook(book);
  }
}

export default PublicLibraryProxy;
```

The next snippet shows the implementation of a Virtual Proxy.

```
import PublicLibrary from '../../common/PublicLibrary';

function initializeLibrary(instance) {
  if (instance.library === null) {
    instance.library = new PublicLibrary(instance.catalog);
  }
}

class PublicLibraryVirtualProxy {
  constructor(catalog = []) {
    this.library = null;
    this.catalog = catalog;
  }

  findBooks(query) {
    console.log("Enter findBooks PublicLibraryVirtualProxy");
    initializeLibrary(this);
    return this.library.findBooks(query);
  }

  checkoutBook(book) {
    initializeLibrary(this);
    return this.library.checkoutBook(book);
```

```
  }

  returnBook(book) {
    initializeLibrary(this);
    return this.library.returnBook(book);
  }
}

export default PublicLibraryVirtualProxy;
```

Using a Virtual Proxy you can delegate the initialization to the moment when it's actually needed.

You can see both examples running in the next two links.

Simple Proxy: https://runkit.com/tcorral/proxy

Virtual Proxy: https://runkit.com/tcorral/virtual-proxy

When not to use it?

When you can use Facade instead.

Pros

- Clients don't notice they are not using the original object.
- Lifecycle management.
- If service object is not ready it will still work.

Cons

- The response takes longer to be resolved.

Summary

In this chapter we have explained and implemented all the different Structural Design Patterns.

You have learn how to use Adapter so you can use incompatible objects in your system without noticing it. We have created Composite pattern that makes easier to understand how tree structures work. Decorator Pattern has demonstrated to be very useful when the different flavors of one or more products cannot be managed by inheritance. With a simple example of a Home Theater we got deeply in the Facade pattern demonstrating how to simplify the interface to the user and encapsulating the implementation from the client. With the Flyweight we have, drastically, reduced the amount of memory we can use in our applications when we have to deal with thousands of instances of objects. And last but not least we have used Proxy pattern to work with heavyweight objects.

In the next chapter we will look at Behavioral Design Patterns that will make more efficient and safe how the program manage the behaviors between elements of the same system.

Questions

- What problem solves a Structural Design Pattern?
- When to use Adapter?
- When not to use Adapter?
- When to use Composite?
- When not to use Composite?
- When to use Decorator?
- When not to use Decorator?
- When to use Bridge?
- When not to use Bridge?
- When to use Facade?
- When not to use Facade?
- When to use Flyweight?
- When to use Flyweight?
- When not to use Proxy?
- When not to use Proxy?

Further Reading

To get deep into ES6 you can read this book. https://www.packtpub.com/web-development/learning-ecmascript-6

In order to know more about the classic design patterns and the GoF you can visit: http://wiki.c2.com/?GangOfFour

Behavioural Patterns

Coming soon...

Performance patterns

Coming soon...

Asynchronous patterns

Coming soon...

Patterns for Testing

Coming soon...

Advanced Patterns

Coming soon...

Application Patterns

Coming soon...

Web Patterns

Coming soon...

Messaging Patterns

Coming soon...

Micro-services

Coming soon...

ES2015/2017/2018 Solutions Today and the Road ahead

Coming soon...

ES2019 What is ESNEXT

Coming soon...