Ved Antani, Simon Timms,
Dan Mantyla

JavaScript:
Functional Programming for JavaScript Developers
Learning Path

Leverage the power of functional programming with modern JavaScript techniques to build faster and reliable web applications



Packt>

JavaScript: Functional Programming for JavaScript Developers

Leverage the power of functional programming with modern JavaScript techniques to build faster and reliable web applications

A course in three modules



BIRMINGHAM - MUMBAI

JavaScript: Functional Programming for JavaScript Developers

Copyright © 2016 Packt Publishing

All rights reserved. No part of this course may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this course to ensure the accuracy of the information presented. However, the information contained in this course is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this course.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this course by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Published on: August 2016

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78712-466-0

www.packtpub.com

Credits

Authors

Ved Antani

Simon Timms

Dan Mantyla

Reviewers

Ivano Malavolta

Dobrin Ganev

Dom Derrien

Joe Dorocak

Peter Ehrlich

Edward E. Griebel Jr.

Content Development Editor

Pooja Mhapsekar

Graphics

Disha Haria

Production Coordinator

Aparna Bhagat

Preface

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

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

What this learning path covers

Module 1, Mastering JavaScript, provides a detailed overview of the language fundamentals and some of the modern tools and libraries – like jQuery, underscore.js and jasmine.

Module 2, Mastering JavaScript Design Patterns- Second Edition, is divided into two main parts. The first part covers the classical design patterns, which are found in the GoF book whereas the second part looks at patterns, which are either not covered in the GoF book or ones that are more specific to JavaScript.

Module 3, Functional Programming in JavaScript, explores the core concepts of functional programming common to all functional languages, with examples of their use in JavaScript.

What you need for this learning path

All the examples in this course can be run on any of the modern browsers. For the last chapter from first module, you will need Node.js. You will need the following to run the examples and samples from this course:

- A computer with Windows 7 or higher, Linux or Mac OSX installed
- Latest version of Google Chrome or Mozilla Firefox browser
- A texteditor of your choice. Sublime Text, Vi, Atom or Notepad++ would be ideal. The choice is entirely yours.

There are standalone JavaScript engines written in C++ (V8) and Java (Rhino) and these are used to power all sorts of tools such as nodejs, couchdb and even elasticsearch. These patterns can be applied to any of these technologies.

Who this learning path is for

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

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this course—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the course's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packtproduct, we have a number of things to help you to get the most from your purchase.

Downloading theexample code

You can download the example code files for this course from your account at `http://www.packtpub.com`. If you purchased this course elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enterthe name of the course in the **Search** box.
5. Select the course for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this course from.
7. Click on **Code Download**.

You can also download the code files by clicking on the**Code Files**button on the course's webpage at the Packt Publishing website. This page can be accessed by entering the course's name in the **Search** box. Please note that you need to be logged into your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the course is also hosted on GitHub at `https://github.com/PacktPublishing/JavaScript--Functional-Programming-for-JavaScript-Developers`.We also have other code bundles from our rich catalog of books, courses and videos available at `https://github.com/PacktPublishing/`. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our courses—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this course. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your course, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the coursein the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this course, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

Module 1: Mastering JavaScript

Chapter 1: JavaScript Primer 3

A little bit of history 4
How to use this book 5
Hello World 8
Summary 45

Chapter 2: Functions, Closures, and Modules 47

A function literal 48
Functions as data 51
Scoping 52
Function declarations versus function expressions 60
The arguments parameter 62
Anonymous functions 66
Closures 68
Timers and callbacks 71
Private variables 71
Loops and closures 72
Modules 73
Summary 75

Chapter 3: Data Structures and Manipulation 77

Regular expressions 78
Exact match 79
Match from a class of characters 79
Repeated occurrences 83
Beginning and end 86
Backreferences 86
Greedy and lazy quantifiers 87
Arrays 88
Maps 97
Sets 97
A matter of style 99
Summary 99

Chapter 4: Object-Oriented JavaScript 101
Understanding objects 101
Instance properties versus prototype properties 106
Inheritance 112
Getters and setters 119
Summary 122
Chapter 5: Testing and Debugging 123
Unit testing 124
JavaScript debugging 132
Summary 140
Chapter 6: ECMAScript 6 141
Shims or polyfills 142
Transpilers 142
ES6 syntax changes 143
Summary 157
Chapter 7: DOM Manipulation and Events 159
DOM 159
Chaining 169
Traversal and manipulation 169
Working with browser events 171
Propagation 172
jQuery event handling and propagation 173
Event delegation 176
The event object 177
Summary 178
Chapter 8: Server-Side JavaScript 179
An asynchronous evented-model in a browser 180
Callbacks 184
Timers 188
EventEmitters 189
Modules 190
npm 193
JavaScript performance 196
Summary 200

Module 2: Mastering JavaScript Design Patterns - Second Edition

Chapter 1: Designing for Fun and Profit 203
- The road to JavaScript 203
- What is a design pattern? 211
- Anti-patterns 214
- Summary 215

Chapter 2: Organizing Code 219
- Chunks of code 219
- What's the matter with global scope anyway? 222
- Objects in JavaScript 223
- Build me a prototype 227
- Inheritance 230
- Modules 232
- ECMAScript 2015 classes and modules 236
- Best practices and troubleshooting 237
- Summary 237

Chapter 3: Creational Patterns 239
- Abstract factory 240
- Builder 247
- Factory method 250
- Singleton 254
- Prototype 256
- Tips and tricks 257
- Summary 258

Chapter 4: Structural Patterns 259
- Adapter 260
- Bridge 264
- Composite 267
- Decorator 272
- Façade 274
- Flyweight 276
- Proxy 278
- Hints and tips 281
- Summary 281

Chapter 5: Behavioral Patterns 283
Chain of responsibility 284
Command 288
Interpreter 292
Iterator 294
Mediator 297
Memento 299
Observer 302
State 305
Strategy 308
Template method 312
Visitor 315
Hints and tips 319
Summary 320

Chapter 6: Functional Programming 323
Functional functions are side-effect-free 324
Function passing 324
Filters and pipes 328
Accumulators 331
Memoization 333
Immutability 335
Lazy instantiation 337
Hints and tips 339
Summary 339

Chapter 7: Reactive Programming 341
Application state changes 342
Streams 342
Filtering streams 345
Merging streams 347
Streams for multiplexing 349
Hints and tips 349
Summary 350

Chapter 8: Application Patterns 351
First, some history 352
Model View Controller 352
Model View Presenter 361
Model View ViewModel 365
Tips and tricks 371
Summary 371

Chapter 9: Web Patterns 373
Sending JavaScript 373
Plugins 379
Doing two things at once – multithreading 384
Circuit breaker pattern 386
Promise pattern 389
Hints and tips 391
Summary 392
Chapter 10: Messaging Patterns 393
What's a message anyway? 394
Request-reply 398
Publish-subscribe 401
Dead letter queues 407
Hints and tips 412
Summary 412
Chapter 11: Microservices 415
Façade 417
Service selector 418
Aggregate services 419
Pipeline 420
Message upgrader 421
Failure patterns 422
Hints and tips 425
Summary 425
Chapter 12: Patterns for Testing 427
The testing pyramid 427
Testing in the small with unit tests 428
Arrange-Act-Assert 430
Fake objects 431
Test spies 433
Stubs 434
Mock 436
Monkey patching 437
Interacting with the user interface 437
Tips and tricks 440
Summary 441

Chapter 13: Advanced Patterns 443
Dependency injection 443
Live post processing 446
Aspect oriented programming 448
Mixins 451
Macros 452
Tips and tricks 453
Summary 454
Chapter 14: ECMAScript-2015/2016 Solutions Today 455
TypeScript 455
BabelJS 459
Tips and tricks 466
Summary 467

Module 3: Functional Programming in JavaScript

Chapter 1: The Powers of JavaScript's Functional Side – a Demonstration 471
Introduction 471
The demonstration 472
The application – an e-commerce website 472
Functional programming 474
Summary 477
Chapter 2: Fundamentals of Functional Programming 479
Functional programming languages 479
Working with functions 487
The functional programmer's toolkit 497
Summary 505
Chapter 3: Setting Up the Functional Programming Environment 507
Introduction 507
Functional libraries for JavaScript 508
Development and production environments 518
Summary 522

Chapter 4: Implementing Functional Programming Techniques in JavaScript 523
Partial function application and currying 524
Function composition 532
Mostly functional programming 538
Functional reactive programming 541
Summary 545
Chapter 5: Category Theory 547
Category theory 548
Functors 553
Monads 557
Implementing categories 565
Summary 568
Chapter 6: Advanced Topics and Pitfalls in JavaScript 569
Recursion 570
Variable scope 579
Function declarations versus function expressions versus the function constructor 584
Summary 587
Chapter 7: Functional and Object-oriented Programming in JavaScript 589
JavaScript – the multi-paradigm language 590
JavaScript's object-oriented implementation – using prototypes 591
Mixing functional and object-oriented programming in JavaScript 595
Summary 603
Appendix A: Common Functions for Functional Programming in JavaScript 605
Appendix B: Glossary of Terms 613
Bibliography 617

Module 1

Mastering JavaScript

Explore and master modern JavaScript techniques in order to build large-scale web applications

1
JavaScript Primer

It is always difficult to pen the first few words, especially on a subject like JavaScript. This difficulty arises primarily because so many things have been said about this language. JavaScript has been the *Language of the Web*—lingua franca, if you will, since the earliest days of the Netscape Navigator. JavaScript went from a tool of the amateur to the weapon of the connoisseur in a shockingly short period of time.

JavaScript is the most popular language on the web and open source ecosystem. `http://githut.info/` charts the number of active repositories and overall popularity of the language on GitHub for the last few years. JavaScript's popularity and importance can be attributed to its association with the browser. Google's V8 and Mozilla's SpiderMonkey are extremely optimized JavaScript engines that power Google Chrome and Mozilla Firefox browsers, respectively.

Although web browsers are the most widely used platforms for JavaScript, modern databases such as MongoDB and CouchDB use JavaScript as their scripting and query language. JavaScript has become an important platform outside browsers as well. Projects such as **Node.js** and **io.js** provide powerful platforms to develop scalable server environments using JavaScript. Several interesting projects are pushing the language capabilities to its limits, for example, **Emscripten** (`http://kripken.github.io/emscripten-site/`) is a **Low-Level Virtual Machine** (**LLVM**)-based project that compiles C and C++ into highly optimizable JavaScript in an **asm.js** format. This allows you to run C and C++ on the web at near native speed.

JavaScript is built around solid foundations regarding, for example, functions, dynamic objects, loose typing, prototypal inheritance, and a powerful object literal notation.

While JavaScript is built on sound design principles, unfortunately, the language had to evolve along with the browser. Web browsers are notorious in the way they support various features and standards. JavaScript tried to accommodate all the whims of the browsers and ended up making some very bad design decisions. These bad parts (the term made famous by Douglas Crockford) overshadowed the good parts of the language for most people. Programmers wrote bad code, other programmers had nightmares trying to debug that bad code, and the language eventually got a bad reputation. Unfortunately, JavaScript is one of the most misunderstood programming languages (`http://javascript.crockford.com/javascript.html`).

Another criticism leveled at JavaScript is that it lets you get things done without you being an expert in the language. I have seen programmers write exceptionally bad JavaScript code just because they wanted to get the things done quickly and JavaScript allowed them to do just this. I have spent hours debugging very bad quality JavaScript written by someone who clearly was not a programmer. However, the language is a tool and cannot be blamed for sloppy programming. Like all crafts, programming demands extreme dedication and discipline.

A little bit of history

In 1993, the **Mosaic** browser of **National Center for Supercomputing Applications** (**NCSA**) was one of the first popular web browsers. A year later, Netscape Communications created the proprietary web browser, **Netscape Navigator**. Several original Mosaic authors worked on Navigator.

In 1995, Netscape Communications hired Brendan Eich with the promise of letting him implement **Scheme** (a Lisp dialect) in the browser. Before this happened, Netscape got in touch with Sun Microsystems (now Oracle) to include Java in the Navigator browser.

Due to the popularity and easy programming of Java, Netscape decided that a scripting language had to have a syntax similar to that of Java. This ruled out adopting existing languages such as Python, **Tool Command Language** (**TCL**), or Scheme. Eich wrote the initial prototype in just 10 days (`http://www.computer.org/csdl/mags/co/2012/02/mco2012020007.pdf`), in May 1995. JavaScript's first code name was **Mocha**, coined by Marc Andreessen. Netscape later changed it to **LiveScript**, for trademark reasons. In early December 1995, Sun licensed the trademark Java to Netscape. The language was renamed to its final name, JavaScript.

How to use this book

This book is not going to help if you are looking to get things done quickly. This book is going to focus on the correct ways to code in JavaScript. We are going to spend a lot of time understanding how to avoid the bad parts of the language and build reliable and readable code in JavaScript. We will skirt away from sloppy features of the language just to make sure that you are not getting used to them—if you have already learned to code using these habits, this book will try to nudge you away from this. There will be a lot of focus on the correct style and tools to make your code better.

Most of the concepts in this book are going to be examples and patterns from real-world problems. I will insist that you code each of the snippets to make sure that your understanding of the concept is getting programmed into your muscle memory. Trust me on this, there is no better way to learn programming than writing a lot of code.

Typically, you will need to create an HTML page to run an embedded JavaScript code as follows:

```
<!DOCTYPE html>
<html>
<head>
  <script type="text/javascript" src="script.js"></script>
  <script type="text/javascript">
    var x = "Hello World";
    console.log(x);
  </script>
</head>
<body>
</body>
</html>
```

This sample code shows two ways in which JavaScript is embedded into the HTML page. First, the `<script>` tag in `<head>` imports JavaScript, while the second `<script>` tag is used to embed inline JavaScript.



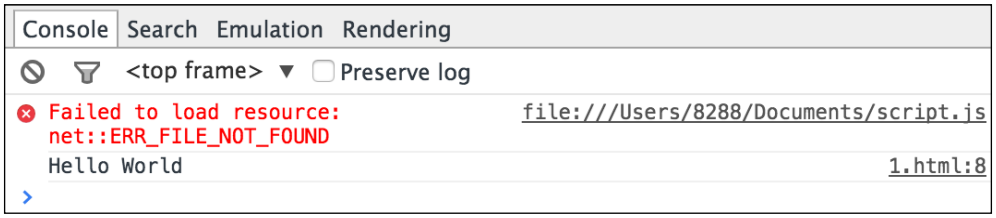
Downloading the example code

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

You can save this HTML page locally and open it in a browser. On Firefox, you can open the **Developer** console (Firefox menu | **Developer** | **Web Console**) and you can see the **"Hello World"** text on the **Console** tab. Based on your OS and browser version, the screen may look different:



You can run the page and inspect it using Chrome's **Developer Tool**:



A very interesting thing to notice here is that there is an error displayed on the console regarding the missing `.js` file that we are trying to import using the following line of code:

```
<script type="text/javascript" src="script.js"></script>
```

Using browser developer consoles or an extension such as **Firebug** can be very useful in debugging error conditions in the code. We will discuss in detail the debugging techniques in later chapters.

Creating such HTML scaffolds can be tedious for every exercise in this book. Instead, we want to use a **Read-Eval-Print-Loop** (**REPL**) for JavaScript. Unlike Python, JavaScript does not come packaged with an REPL. We can use Node.js as an REPL. If you have Node.js installed on your machine, you can just type `node` on the command line and start experimenting with it. You will observe that Node REPL errors are not very elegantly displayed.

Let's see the following example:

```
EN-VedA:~$ node
>function greeter(){
  x="World"l
SyntaxError: Unexpected identifier
  at Object.exports.createScript (vm.js:44:10)
  at REPLServer.defaultEval (repl.js:117:23)
  at bound (domain.js:254:14)
  …
```

After this error, you will have to restart. Still, it can help you try out small fragments of code a lot faster.

Another tool that I personally use a lot is **JS Bin** (`http://jsbin.com/`). JS Bin provides you with a great set of tools to test JavaScript, such as syntax highlighting and runtime error detection. The following is a screenshot of JS Bin:



Based on your preference, you can pick the tool that makes it easier to try out the code samples. Regardless of which tool you use, make sure that you type out every exercise in this book.

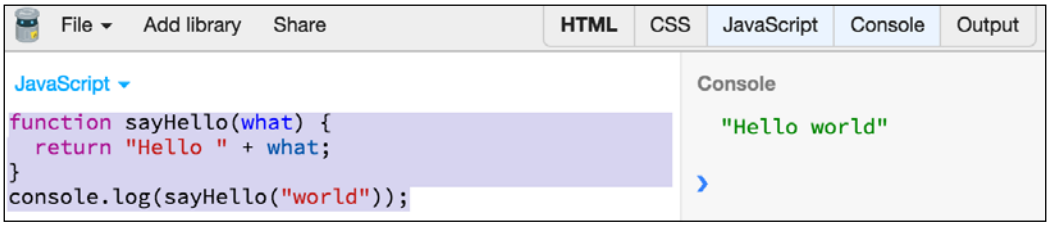
Hello World

No programming language should be published without a customary Hello World program—why should this book be any different?

Type (don't copy and paste) the following code in JS Bin:

```
function sayHello(what) {
  return "Hello " + what;
}
console.log(sayHello("world"));
```

Your screen should look something as follows:



An overview of JavaScript

In a nutshell, JavaScript is a prototype-based scripting language with dynamic typing and first-class function support. JavaScript borrows most of its syntax from Java, but is also influenced by Awk, Perl, and Python. JavaScript is case-sensitive and white space-agnostic.

Comments

JavaScript allows single line or multiple line comments. The syntax is similar to C or Java:

```
// a one line comment

/* this is a longer,
   multi-line comment
 */

/* You can't /* nest comments */ SyntaxError */
```

Variables

Variables are symbolic names for values. The names of variables, or identifiers, must follow certain rules.

A JavaScript variable name must start with a letter, underscore (_), or dollar sign ($); subsequent characters can also be digits (0-9). As JavaScript is case sensitive, letters include the characters *A* through *Z* (uppercase) and the characters *a* through *z* (lowercase).

You can use ISO 8859-1 or Unicode letters in variable names.

New variables in JavaScript should be defined with the **var** keyword. If you declare a variable without assigning a value to it, its type is undefined by default. One terrible thing is that if you don't declare your variable with the var keyword, they become implicit globals. Let me reiterate that implicit globals are a terrible thing—we will discuss this in detail later in the book when we discuss variable scopes and closures, but it's important to remember that you should always declare a variable with the var keyword unless you know what you are doing:

```
var a;      //declares a variable but its undefined
var b = 0;
console.log(b);    //0
console.log(a);    //undefined
console.log(a+b);  //NaN
```

The `NaN` value is a special value that indicates that the entity is *not a number*.

Constants

You can create a read-only named constant with the **const** keyword. The constant name must start with a letter, underscore, or dollar sign and can contain alphabetic, numeric, or underscore characters:

```
const area_code = '515';
```

A constant cannot change the value through assignment or be redeclared, and it has to be initialized to a value.

JavaScript supports the standard variations of types:

- Number
- String
- Boolean
- Symbol (new in ECMAScript 6)
- Object:
    - Function
    - Array
    - Date
    - RegExp
- Null
- Undefined

Number

The **Number** type can represent both 32-bit integer and 64-bit floating point values. For example, the following line of code declares a variable to hold an integer value, which is defined by the literal 555:

```
var aNumber = 555;
```

To define a floating point value, you need to include a decimal point and one digit after the decimal point:

```
var aFloat = 555.0;
```

Essentially, there's no such thing as an integer in JavaScript. JavaScript uses a 64-bit floating point representation, which is the same as Java's double.

Hence, you would see something as follows:

```
EN-VedA:~$ node
> 0.1+0.2
0.30000000000000004
> (0.1+0.2)===0.3
false
```

I recommend that you read the exhaustive answer on Stack Overflow (`http://stackoverflow.com/questions/588004/is-floating-point-math-broken`) and (`http://floating-point-gui.de/`), which explains why this is the case. However, it is important to understand that floating point arithmetic should be handled with due care. In most cases, you will not have to rely on extreme precision of decimal points but if you have to, you can try using libraries such as **big.js** (`https://github.com/MikeMcl/big.js`) that try to solve this problem.

If you intend to code extremely precise financial systems, you should represent $ values as cents to avoid rounding errors. One of the systems that I worked on used to round off the **Value Added Tax** (**VAT**) amount to two decimal points. With thousands of orders a day, this rounding off amount per order became a massive accounting headache. We needed to overhaul the entire Java web service stack and JavaScript frontend for this.

A few special values are also defined as part of the Number type. The first two are `Number.MAX_VALUE` and `Number.MIN_VALUE`, which define the outer bounds of the Number value set. All ECMAScript numbers must fall between these two values, without exception. A calculation can, however, result in a number that does not fall in between these two numbers. When a calculation results in a number greater than `Number.MAX_VALUE`, it is assigned a value of `Number.POSITIVE_INFINITY`, meaning that it has no numeric value anymore. Likewise, a calculation that results in a number less than `Number.MIN_VALUE` is assigned a value of `Number.NEGATIVE_INFINITY`, which also has no numeric value. If a calculation returns an infinite value, the result cannot be used in any further calculations. You can use the `isInfinite()` method to verify if the calculation result is an infinity.

Another peculiarity of JavaScript is a special value called NaN (short for *Not a Number*). In general, this occurs when conversion from another type (String, Boolean, and so on) fails. Observe the following peculiarity of NaN:

```
EN-VedA:~ $ node
> isNaN(NaN);
true
> NaN==NaN;
false
> isNaN("elephant");
true
> NaN+5;
NaN
```

The second line is strange—NaN is not equal to NaN. If NaN is part of any mathematical operation, the result also becomes NaN. As a general rule, stay away from using NaN in any expression. For any advanced mathematical operations, you can use the `Math` global object and its methods:

```
> Math.E
2.718281828459045
> Math.SQRT2
1.4142135623730951
> Math.abs(-900)
900
> Math.pow(2,3)
8
```

You can use the `parseInt()` and `parseFloat()` methods to convert a string expression to an integer or float:

```
> parseInt("230",10);
230
> parseInt("010",10);
10
> parseInt("010",8); //octal base
8
> parseInt("010",2); //binary
2
> + "4"
4
```

With `parseInt()`, you should provide an explicit base to prevent nasty surprises on older browsers. The last trick is just using a `+` sign to auto-convert the `"42"` string to a number, `42`. It is also prudent to handle the `parseInt()` result with `isNaN()`. Let's see the following example:

```
var underterminedValue = "elephant";
if (isNaN(parseInt(underterminedValue,2)))
{
   console.log("handle not a number case");
}
else
{
   console.log("handle number case");
}
```

In this example, you are not sure of the type of the value that the `underterminedValue` variable can hold if the value is being set from an external interface. If `isNaN()` is not handled, `parseInt()` will cause an exception and the program can crash.

String

In JavaScript, strings are a sequence of Unicode characters (each character takes 16 bits). Each character in the string can be accessed by its index. The first character index is zero. Strings are enclosed inside `"` or `'`—both are valid ways to represent strings. Let's see the following:

```
> console.log("Hippopotamus chewing gum");
Hippopotamus chewing gum
> console.log('Single quoted hippopotamus');
Single quoted hippopotamus
> console.log("Broken \n lines");
Broken
 lines
```

The last line shows you how certain character literals when escaped with a backslash `\` can be used as special characters. The following is a list of such special characters:

- `\n`: Newline
- `\t`: Tab
- `\b`: Backspace
- `\r`: Carriage return
- `\\`: Backslash
- `\'`: Single quote
- `\"`: Double quote

You get default support for special characters and Unicode literals with JavaScript strings:

```
> '\xA9'
'©'
> '\u00A9'
'©'
```

One important thing about JavaScript Strings, Numbers, and Booleans is that they actually have wrapper objects around their primitive equivalent. The following example shows the usage of the wrapper objects:

```
var s = new String("dummy"); //Creates a String object
console.log(s); //"dummy"
console.log(typeof s); //"object"
var nonObject = "1" + "2"; //Create a String primitive
console.log(typeof nonObject); //"string"
var objString = new String("1" + "2"); //Creates a String object
console.log(typeof objString); //"object"
//Helper functions
console.log("Hello".length); //5
console.log("Hello".charAt(0)); //"H"
console.log("Hello".charAt(1)); //"e"
console.log("Hello".indexOf("e")); //1
console.log("Hello".lastIndexOf("l")); //3
console.log("Hello".startsWith("H")); //true
console.log("Hello".endsWith("o")); //true
console.log("Hello".includes("X")); //false
var splitStringByWords = "Hello World".split(" ");
console.log(splitStringByWords); //["Hello", "World"]
var splitStringByChars = "Hello World".split("");
console.log(splitStringByChars); //["H", "e", "l", "l", "o", " ",
  "W", "o", "r", "l", "d"]
console.log("lowercasestring".toUpperCase()); //"LOWERCASESTRING"
console.log("UPPPERCASESTRING".toLowerCase());
  //"upppercasestring"
console.log("There are no spaces in the end     ".trim());
  //"There are no spaces in the end"
```

JavaScript allows multiline strings also. Strings enclosed within ` (Grave accent—`https://en.wikipedia.org/wiki/Grave_accent`) are considered multiline. Let's see the following example:

```
> console.log(`string text on first line
string text on second line `);
"string text on first line
string text on second line "
```

This kind of string is also known as a template string and can be used for string interpolation. JavaScript allows Python-like string interpolation using this syntax.

Normally, you would do something similar to the following:

```
var a=1, b=2;
console.log("Sum of values is :" + (a+b) + " and multiplication is :"
  + (a*b));
```

However, with string interpolation, things become a bit clearer:

```
console.log(`Sum of values is :${a+b} and multiplication is :
  ${a*b}`);
```

Undefined values

JavaScript indicates an absence of meaningful values by two special values—null, when the non-value is deliberate, and undefined, when the value is not assigned to the variable yet. Let's see the following example:

```
> var xl;
> console.log(typeof xl);
undefined
> console.log(null==undefined);
true
```

Booleans

JavaScript Boolean primitives are represented by `true` and `false` keywords. The following rules govern what becomes false and what turns out to be true:

- False, 0, the empty string (""), NaN, null, and undefined are represented as false
- Everything else is true

JavaScript Booleans are tricky primarily because the behavior is radically different in the way you create them.

There are two ways in which you can create Booleans in JavaScript:

- You can create primitive Booleans by assigning a true or false literal to a variable. Consider the following example:

  ```
  var pBooleanTrue = true;
  var pBooleanFalse = false;
  ```

- Use the `Boolean()` function; this is an ordinary function that returns a primitive Boolean:

```
var fBooleanTrue = Boolean(true);
var fBooleanFalse = Boolean(false);
```

Both these methods return expected *truthy* or *falsy* values. However, if you create a Boolean object using the `new` operator, things can go really wrong.

Essentially, when you use the `new` operator and the `Boolean(value)` constructor, you don't get a primitive `true` or `false` in return, you get an object instead—and unfortunately, JavaScript considers an object as *truthy*:

```
var oBooleanTrue = new Boolean(true);
var oBooleanFalse = new Boolean(false);
console.log(oBooleanTrue); //true
console.log(typeof oBooleanTrue); //object
if(oBooleanFalse){
 console.log("I am seriously truthy, don't believe me");
}
>"I am seriously truthy, don't believe me"

if(oBooleanTrue){
 console.log("I am also truthy, see ?");
}
>"I am also truthy, see ?"

//Use valueOf() to extract real value within the Boolean object
if(oBooleanFalse.valueOf()){
 console.log("With valueOf, I am false");
}else{
 console.log("Without valueOf, I am still truthy");
}
>"Without valueOf, I am still truthy"
```

So, the smart thing to do is to always avoid Boolean constructors to create a new Boolean object. It breaks the fundamental contract of Boolean logic and you should stay away from such difficult-to-debug buggy code.

The instanceof operator

One of the problems with using reference types to store values has been the use of the **typeof** operator, which returns `object` no matter what type of object is being referenced. To provide a solution, you can use the **instanceof** operator. Let's see some examples:

```
var aStringObject = new String("string");
console.log(typeof aStringObject);        //"object"
console.log(aStringObject instanceof String);    //true
var aString = "This is a string";
console.log(aString instanceof String);     //false
```

The third line returns `false`. We will discuss why this is the case when we discuss prototype chains.

Date objects

JavaScript does not have a date data type. Instead, you can use the **Date** object and its methods to work with dates and times in your applications. A Date object is pretty exhaustive and contains several methods to handle most date- and time-related use cases.

JavaScript treats dates similarly to Java. JavaScript store dates as the number of milliseconds since January 1, 1970, 00:00:00.

You can create a Date object using the following declaration:

```
var dataObject = new Date([parameters]);
```

The parameters for the Date object constructors can be as follows:

- No parameters creates today's date and time. For example, `var today = new Date();`.
- A String representing a date as `Month day, year hours:minutes:seconds`. For example, `var twoThousandFifteen = new Date("December 31, 2015 23:59:59");`. If you omit hours, minutes, or seconds, the value will be set to `0`.
- A set of integer values for the year, month, and day. For example, `var christmas = new Date(2015, 11, 25);`.
- A set of integer values for the year, month, day, hour, minute, and seconds. For example, `var christmas = new Date(2015, 11, 25, 21, 00, 0);`.

Here are some examples on how to create and manipulate dates in JavaScript:

```
var today = new Date();
console.log(today.getDate()); //27
console.log(today.getMonth()); //4
console.log(today.getFullYear()); //2015
console.log(today.getHours()); //23
console.log(today.getMinutes()); //13
console.log(today.getSeconds()); //10
//number of milliseconds since January 1, 1970, 00:00:00 UTC
console.log(today.getTime()); //1432748611392
console.log(today.getTimezoneOffset()); //-330 Minutes

//Calculating elapsed time
var start = Date.now();
// loop for a long time
for (var i=0;i<100000;i++);
var end = Date.now();
var elapsed = end - start; // elapsed time in milliseconds
console.log(elapsed); //71
```

For any serious applications that require fine-grained control over date and time objects, we recommend using libraries such as **Moment.js** (`https://github.com/moment/moment`), **Timezone.js** (`https://github.com/mde/timezone-js`), or **date.js** (`https://github.com/MatthewMueller/date`). These libraries simplify a lot of recurrent tasks for you and help you focus on other important things.

The + operator

The **+** operator, when used as a unary, does not have any effect on a number. However, when applied to a String, the + operator converts it to numbers as follows:

```
var a=25;
a=+a;            //No impact on a's value
console.log(a);  //25

var b="70";
console.log(typeof b); //string
b=+b;           //converts string to number
console.log(b); //70
console.log(typeof b); //number
```

The + operator is used often by a programmer to quickly convert a numeric representation of a String to a number. However, if the String literal is not something that can be converted to a number, you get slightly unpredictable results as follows:

```
var c="foo";
c=+c;            //Converts foo to number
console.log(c);  //NaN
console.log(typeof c);  //number

var zero="";
zero=+zero; //empty strings are converted to 0
console.log(zero);
console.log(typeof zero);
```

We will discuss the effects of the + operator on several other data types later in the text.

The ++ and -- operators

The ++ operator is a shorthand version of adding `1` to a value and -- is a shorthand to subtract `1` from a value. Java and C have equivalent operators and most will be familiar with them. How about this?

```
var a= 1;
var b= a++;
console.log(a); //2
console.log(b); //1
```

Err, what happened here? Shouldn't the `b` variable have the value `2`? The ++ and -- operators are unary operators that can be used either prefix or postfix. The order in which they are used matters. When ++ is used in the prefix position as `++a`, it increments the value before the value is returned from the expression rather than after as with `a++`. Let's see the following code:

```
var a= 1;
var b= ++a;
console.log(a);  //2
console.log(b);  //2
```

Many programmers use the chained assignments to assign a single value to multiple variables as follows:

```
var a, b, c;
a = b = c = 0;
```

This is fine because the assignment operator (=) results in the value being assigned. In this case, `c=0` is evaluated to `0`; this would result in `b=0`, which also evaluates to `0`, and hence, `a=0` is evaluated.

However, a slight change to the previous example yields extraordinary results. Consider this:

```
var a = b = 0;
```

In this case, only the `a` variable is declared with `var`, while the `b` variable is created as an accidental global. (If you are in the strict mode, you will get an error for this.) With JavaScript, be careful what you wish for, you might get it.

Boolean operators

There are three Boolean operators in JavaScript—AND(&), OR(|), and NOT(!).

Before we discuss logical AND and OR operators, we need to understand how they produce a Boolean result. Logical operators are evaluated from left to right and they are tested using the following short-circuit rules:

- **Logical AND**: If the first operand determines the result, the second operand is not evaluated.

  In the following example, I have highlighted the right-hand side expression if it gets executed as part of short-circuit evaluation rules:

```
console.log(true  && true); // true AND true returns true
console.log(true  && false);// true AND false returns false
console.log(false && true);// false AND true returns false
console.log("Foo" && "Bar");// Foo(true) AND Bar(true)
  returns Bar
console.log(false && "Foo");// false && Foo(true) returns
  false
console.log("Foo" && false);// Foo(true) && false returns
  false
console.log(false && (1 == 2));// false && false(1==2) returns
false
```

- **Logical OR**: If the first operand is true, the second operand is not evaluated:

```
console.log(true  || true); // true AND true returns true
console.log(true  || false);// true AND false returns true
console.log(false || true);// false AND true returns true
console.log("Foo" || "Bar");// Foo(true) AND Bar(true) returns Foo
console.log(false || "Foo");// false && Foo(true) returns Foo
console.log("Foo" || false);// Foo(true) && false returns Foo
console.log(false || (1 == 2));// false && false(1==2) returns
false
```

  However, both logical AND and logical OR can also be used for non-Boolean operands. When either the left or right operand is not a primitive Boolean value, AND and OR do not return Boolean values.

Now we will explain the three logical Boolean operators:

- Logical AND(&&): If the first operand object is *falsy*, it returns that object. If its *truthy*, the second operand object is returned:

```
console.log (0 && "Foo");  //First operand is falsy -
  return it
console.log ("Foo" && "Bar"); //First operand is truthy,
  return the second operand
```

- Logical OR(||): If the first operand is *truthy*, it's returned. Otherwise, the second operand is returned:

```
console.log (0 || "Foo");  //First operand is falsy -
  return second operand
console.log ("Foo" || "Bar"); //First operand is truthy,
  return it
console.log (0 || false); //First operand is falsy, return
  second operand
```

  The typical use of a logical OR is to assign a default value to a variable:

```
function greeting(name){
    name = name || "John";
    console.log("Hello " + name);
}

greeting("Johnson"); // alerts "Hi Johnson";
greeting(); //alerts "Hello John"
```

  You will see this pattern frequently in most professional JavaScript libraries. You should understand how the defaulting is done by using a logical OR operator.

- **Logical NOT**: This always returns a Boolean value. The value returned depends on the following:

```
//If the operand is an object, false is returned.
var s = new String("string");
console.log(!s);              //false

//If the operand is the number 0, true is returned.
var t = 0;
console.log(!t);              //true

//If the operand is any number other than 0, false is returned.
var x = 11;
console.log(!x);              //false

//If operand is null or NaN, true is returned
var y =null;
var z = NaN;
console.log(!y);              //true
console.log(!z);              //true
//If operand is undefined, you get true
var foo;
console.log(!foo);            //true
```

Additionally, JavaScript supports C-like ternary operators as follows:

```
var allowedToDrive = (age > 21) ? "yes" : "no";
```

If `(age>21)`, the expression after `?` will be assigned to the `allowedToDrive` variable and the expression after `:` is assigned otherwise. This is equivalent to an if-else conditional statement. Let's see another example:

```
function isAllowedToDrive(age){
  if(age>21){
    return true;
  }else{
    return false;
  }
}
console.log(isAllowedToDrive(22));
```

In this example, the `isAllowedToDrive()` function accepts one integer parameter, `age`. Based on the value of this variable, we return true or false to the calling function. This is a well-known and most familiar if-else conditional logic. Most of the time, if-else keeps the code easier to read. For simpler cases of single conditions, using the ternary operator is also okay, but if you see that you are using the ternary operator for more complicated expressions, try to stick with if-else because it is easier to interpret if-else conditions than a very complex ternary expression.

If-else conditional statements can be nested as follows:

```
if (condition1) {
  statement1
} else if (condition2) {
  statement2
} else if (condition3) {
  statement3
}
..
} else {
  statementN
}
```

Purely as a matter of taste, you can indent the nested `else if` as follows:

```
if (condition1) {
  statement1
} else
    if (condition2) {
```

Do not use assignments in place of a conditional statement. Most of the time, they are used because of a mistake as follows:

```
if(a=b) {
  //do something
}
```

Mostly, this happens by mistake; the intended code was `if(a==b)`, or better, `if(a===b)`. When you make this mistake and replace a conditional statement with an assignment statement, you end up committing a very difficult-to-find bug. However, if you really want to use an assignment statement with an if statement, make sure that you make your intentions very clear.

One way is to put extra parentheses around your assignment statement:

```
if((a=b)){
  //this is really something you want to do
}
```

Another way to handle conditional execution is to use switch-case statements. The switch-case construct in JavaScript is similar to that in C or Java. Let's see the following example:

```
function sayDay(day){
  switch(day){
    case 1: console.log("Sunday");
      break;
    case 2: console.log("Monday");
      break;
    default:
      console.log("We live in a binary world. Go to Pluto");
  }
}

sayDay(1); //Sunday
sayDay(3); //We live in a binary world. Go to Pluto
```

One problem with this structure is that you have `break` out of every case; otherwise, the execution will fall through to the next level. If we remove the `break` statement from the first case statement, the output will be as follows:

```
>sayDay(1);
Sunday
Monday
```

As you can see, if we omit the `break` statement to break the execution immediately after a condition is satisfied, the execution sequence follows to the next level. This can lead to difficult-to-detect problems in your code. However, this is also a popular style of writing conditional logic if you intend to fall through to the next level:

```
function debug(level,msg){
  switch(level){
    case "INFO": //intentional fall-through
    case "WARN" :
    case "DEBUG": console.log(level+ ": " + msg);
      break;
    case "ERROR": console.error(msg);
  }
}
```

```
debug("INFO","Info Message");
debug("DEBUG","Debug Message");
debug("ERROR","Fatal Exception");
```

In this example, we are intentionally letting the execution fall through to write a concise switch-case. If levels are either INFO, WARN, or DEBUG, we use the switch-case to fall through to a single point of execution. We omit the `break` statement for this. If you want to follow this pattern of writing switch statements, make sure that you document your usage for better readability.

Switch statements can have a `default` case to handle any value that cannot be evaluated by any other case.

JavaScript has a while and do-while loop. The while loop lets you iterate a set of expressions till a condition is met. The following first example iterates the statements enclosed within `{}` till the `i<10` expression is true. Remember that if the value of the `i` counter is already greater than `10`, the loop will not execute at all:

```
var i=0;
while(i<10){
  i=i+1;
  console.log(i);
}
```

The following loop keeps executing till infinity because the condition is always true—this can lead to disastrous effects. Your program can use up all your memory or something equally unpleasant:

```
//infinite loop
while(true){
  //keep doing this
}
```

If you want to make sure that you execute the loop at least once, you can use the do-while loop (sometimes known as a post-condition loop):

```
var choice;
do {
choice=getChoiceFromUserInput();
} while(!isInputValid(choice));
```

In this example, we are asking the user for an input till we find a valid input from the user. While the user types invalid input, we keep asking for an input to the user. It is always argued that, logically, every do-while loop can be transformed into a while loop. However, a do-while loop has a very valid use case like the one we just saw where you want the condition to be checked only after there has been one execution of the loop block.

JavaScript has a very powerful loop similar to C or Java—the for loop. The for loop is popular because it allows you to define the control conditions of the loop in a single line.

The following example prints `Hello` five times:

```
for (var i=0;i<5;i++){
  console.log("Hello");
}
```

Within the definition of the loop, you defined the initial value of the loop counter `i` to be `0`, you defined the `i<5` exit condition, and finally, you defined the increment factor.

All three expressions in the previous example are optional. You can omit them if required. For example, the following variations are all going to produce the same result as the previous loop:

```
var x=0;
//Omit initialitzation
for (;x<5;x++){
  console.log("Hello");
}

//Omit exit condition
for (var j=0;;j++){
  //exit condition
  if(j>=5){
    break;
  }else{
    console.log("Hello");
  }
}
//Omit increment
for (var k=0; k<5;){
  console.log("Hello");
  k++;
}
```

You can also omit all three of these expressions and write for loops. One interesting idiom used frequently is to use for loops with empty statements. The following loop is used to set all the elements of the array to `100`. Notice how there is no body to the for-loop:

```
var arr = [10, 20, 30];
// Assign all array values to 100
for (i = 0; i < arr.length; arr[i++] = 100);
console.log(arr);
```

The empty statement here is just the single that we see after the for loop statement. The increment factor also modifies the array content. We will discuss arrays later in the book, but here it's sufficient to see that the array elements are set to the `100` value within the loop definition itself.

Equality

JavaScript offers two modes of equality—strict and loose. Essentially, loose equality will perform the type conversion when comparing two values, while strict equality will check the values without any type conversion. A strict equality check is performed by === while a loose equality check is performed by ==.

ECMAScript 6 also offers the `Object.is` method to do a strict equality check like ===. However, `Object.is` has a special handling for NaN: -0 and +0. When *NaN*===*NaN* and *NaN*==*NaN* evaluates to false, `Object.is(NaN,NaN)` will return true.

Strict equality using ===

Strict equality compares two values without any implicit type conversions. The following rules apply:

- If the values are of a different type, they are unequal.
- For non-numerical values of the same type, they are equal if their values are the same.
- For primitive numbers, strict equality works for values. If the values are the same, === results in `true`. However, a NaN doesn't equal to any number and `NaN===<a number>` would be a `false`.

Strict equality is always the correct equality check to use. Make it a rule to always use === instead of ==:

Condition	Output
`"" === "0"`	false
`0 === ""`	false
`0 === "0"`	false
`false === "false"`	false
`false === "0"`	false
`false === undefined`	false
`false === null`	false
`null === undefined`	false

In case of comparing objects, we get results as follows:

Condition	Output
`{} === {};`	false
`new String('bah') === 'bah';`	false
`new Number(1) === 1;`	false
`var bar = {};` `bar === bar;`	true

The following are further examples that you should try on either JS Bin or Node REPL:

```
var n = 0;
var o = new String("0");
var s = "0";
var b = false;

console.log(n === n); // true - same values for numbers
console.log(o === o); // true - non numbers are compared for their
values
console.log(s === s); // true - ditto

console.log(n === o); // false - no implicit type conversion, types
are different
console.log(n === s); // false - types are different
console.log(o === s); // false - types are different
console.log(null === undefined); // false
console.log(o === null); // false
console.log(o === undefined); // false
```

You can use `!==` to handle the **Not Equal To** case while doing strict equality checks.

Weak equality using ==

Nothing should tempt you to use this form of equality. Seriously, stay away from this form. There are many bad things with this form of equality primarily due to the weak typing in JavaScript. The equality operator, ==, first tries to coerce the type before doing a comparison. The following examples show you how this works:

Condition	Output
`"" == "0"`	false
`0 == ""`	true
`0 == "0"`	true
`false == "false"`	false
`false == "0"`	true
`false == undefined`	false
`false == null`	false
`null == undefined`	true

From these examples, it's evident that weak equality can result in unexpected outcomes. Also, implicit type coercion is costly in terms of performance. So, in general, stay away from weak equality in JavaScript.

JavaScript types

We briefly discussed that JavaScript is a dynamic language. If you have a previous experience of strongly typed languages such as Java, you may feel a bit uncomfortable about the complete lack of type checks that you are used to. Purists argue that JavaScript should claim to have **tags** or perhaps **subtypes**, but not types. Though JavaScript does not have the traditional definition of **types**, it is absolutely essential to understand how JavaScript handles data types and coercion internally. Every nontrivial JavaScript program will need to handle value coercion in some form, so it's important that you understand the concept well.

Explicit coercion happens when you modify the type yourself. In the following example, you will convert a number to a String using the `toString()` method and extract the second character out of it:

```
var fortyTwo = 42;
console.log(fortyTwo.toString()[1]); //prints "2"
```

This is an example of an explicit type conversion. Again, we are using the word **type** loosely because type was not enforced anywhere when you declared the `fortyTwo` variable.

However, there are many different ways in which such coercion can happen. Coercion happening explicitly can be easy to understand and mostly reliable; but if you're not careful, coercion can happen in very strange and surprising ways.

Confusion around coercion is perhaps one of the most talked about frustrations for JavaScript developers. To make sure that you never have this confusion in your mind, let's revisit types in JavaScript. We talked about some concepts earlier:

```
typeof 1              === "number";    // true
typeof "1"            === "string";    // true
typeof { age: 39 }    === "object";    // true
typeof Symbol()       === "symbol";    // true
typeof undefined      === "undefined"; // true
typeof true           === "boolean";   // true
```

So far, so good. We already knew this and the examples that we just saw reinforce our ideas about types.

Conversion of a value from one type to another is called **casting** or explicit coercion. JavaScript also does implicit coercion by changing the type of a value based on certain guesses. These guesses make JavaScript work around several cases and unfortunately make it fail quietly and unexpectedly. The following snippet shows cases of explicit and implicit coercion:

```
var t=1;
var u=""+t; //implicit coercion
console.log(typeof t);  //"number"
console.log(typeof u);  //"string"
var v=String(t);  //Explicit coercion
console.log(typeof v);  //"string"
var x=null
console.log(""+x); //"null"
```

It is easy to see what is happening here. When you use `""+t` to a numeric value of `t` (`1`, in this case), JavaScript figures out that you are trying to concatenate *something* with a `""` string. As only strings can be concatenated with other strings, JavaScript goes ahead and converts a numeric `1` to a `"1"` string and concatenates both into a resulting string value. This is what happens when JavaScript is asked to convert values implicitly. However, `String(t)` is a very deliberate call to convert a number to a String. This is an explicit conversion of types. The last bit is surprising. We are concatenating `null` with `""`—shouldn't this fail?

So how does JavaScript do type conversions? How will an abstract value become a String or number or Boolean? JavaScript relies on `toString()`, `toNumber()`, and `toBoolean()` methods to do this internally.

When a non-String value is coerced into a String, JavaScript uses the `toString()` method internally to do this. All primitives have a natural string form—null has a string form of `"null"`, undefined has a string form of `"undefined"`, and so on. For Java developers, this is analogous to a class having a `toString()` method that returns a string representation of the class. We will see exactly how this works in case of objects.

So essentially you can do something similar to the following:

```
var a="abc";
console.log(a.length);
console.log(a.toUpperCase());
```

If you are keenly following and typing all these little snippets, you would have realized something strange in the previous snippet. How are we calling properties and methods on primitives? How come primitives have objects such as properties and methods? They don't.

As we discussed earlier, JavaScript kindly wraps these primitives in their wrappers by default thus making it possible for us to directly access the wrapper's methods and properties as if they were of the primitives themselves.

When any non-number value needs to be coerced into a number, JavaScript uses the `toNumber()` method internally: `true` becomes `1`, `undefined` becomes `NaN`, `false` becomes `0`, and `null` becomes `0`. The `toNumber()` method on strings works with literal conversion and if this fails, the method returns `NaN`.

What about some other cases?

```
typeof null ==="object" //true
```

Well, null is an object? Yes, an especially long-lasting bug makes this possible. Due to this bug, you need to be careful while testing if a value is null:

```
var x = null;
if (!x && typeof x === "object"){
  console.log("100% null");
}
```

What about other things that may have types, such as functions?

```
f = function test() {
  return 12;
}
console.log(typeof f === "function");  //prints "true"
```

What about arrays?

```
console.log (typeof [1,2,3,4]); //"object"
```

Sure enough, they are also objects. We will take a detailed look at functions and arrays later in the book.

In JavaScript, values have types, variables don't. Due to the dynamic nature of the language, variables can hold any value at any time.

JavaScript doesn't does not enforce types, which means that the language doesn't insist that a variable always hold values of the same initial type that it starts out with. A variable can hold a String, and in the next assignment, hold a number, and so on:

```
var a = 1;
typeof a; // "number"
a = false;
typeof a; // "boolean"
```

The `typeof` operator always returns a String:

```
typeof typeof 1; // "string"
```

Automatic semicolon insertion

Although JavaScript is based on the C style syntax, it does not enforce the use of semicolons in the source code.

However, JavaScript is not a semicolon-less language. A JavaScript language parser needs the semicolons in order to understand the source code. Therefore, the JavaScript parser automatically inserts them whenever it encounters a parse error due to a missing semicolon. It's important to note that **automatic semicolon insertion** (**ASI**) will only take effect in the presence of a newline (also known as a line break). Semicolons are not inserted in the middle of a line.

Basically, if the JavaScript parser parses a line where a parser error would occur (a missing expected ;) and it can insert one, it does so. What are the criteria to insert a semicolon? Only if there's nothing but white space and/or comments between the end of some statement and that line's newline/line break.

There have been raging debates on ASI—a feature justifiably considered to be a very bad design choice. There have been epic discussions on the Internet, such as `https://github.com/twbs/bootstrap/issues/3057` and `https://brendaneich.com/2012/04/the-infernal-semicolon/`.

Before you judge the validity of these arguments, you need to understand what is affected by ASI. The following statements are affected by ASI:

- An empty statement
- A var statement
- An expression statement
- A do-while statement
- A continue statement
- A break statement
- A return statement
- A throw statement

The idea behind ASI is to make semicolons optional at the end of a line. This way, ASI helps the parser to determine when a statement ends. Normally, it ends with a semicolon. ASI dictates that a statement also ends in the following cases:

- A line terminator (for example, a newline) is followed by an illegal token
- A closing brace is encountered
- The end of the file has been reached

Let's see the following example:

```
if (a < 1) a = 1 console.log(a)
```

The `console` token is illegal after `1` and triggers ASI as follows:

```
if (a < 1) a = 1; console.log(a);
```

In the following code, the statement inside the braces is not terminated by a semicolon:

```
function add(a,b) { return a+b }
```

ASI creates a syntactically correct version of the preceding code:

```
function add(a,b) { return a+b; }
```

JavaScript style guide

Every programming language develops its own style and structure. Unfortunately, new developers don't put much effort in learning the stylistic nuances of a language. It is very difficult to develop this skill later once you have acquired bad practices. To produce beautiful, readable, and easily maintainable code, it is important to learn the correct style. There are a ton of style suggestions. We will be picking the most practical ones. Whenever applicable, we will discuss the appropriate style. Let's set some stylistic ground rules.

Whitespaces

Though whitespace is not important in JavaScript, the correct use of whitespace can make the code easy to read. The following guidelines will help in managing whitespaces in your code:

- Never mix spaces and tabs.
- Before you write any code, choose between soft indents (spaces) or real tabs. For readability, I always recommend that you set your editor's indent size to two characters—this means two spaces or two spaces representing a real tab.
- Always work with the *show invisibles* setting turned on. The benefits of this practice are as follows:
    - Enforced consistency.
    - Eliminates the end-of-line white spaces.
    - Eliminates blank line white spaces.
    - Commits and diffs that are easier to read.
    - Uses **EditorConfig** (`http://editorconfig.org/`) when possible.

Parentheses, line breaks, and braces

If, else, for, while, and try always have spaces and braces and span multiple lines. This style encourages readability. Let's see the following code:

```
//Cramped style (Bad)
if(condition) doSomeTask();

while(condition) i++;

for(var i=0;i<10;i++) iterate();

//Use whitespace for better readability (Good)
```

```
//Place 1 space before the leading brace.
if (condition) {
  // statements
}

while ( condition ) {
  // statements
}

for ( var i = 0; i < 100; i++ ) {
  // statements
}

// Better:

var i,
    length = 100;

for ( i = 0; i < length; i++ ) {
  // statements
}

// Or...

var i = 0,
    length = 100;

for ( ; i < length; i++ ) {
  // statements
}

var value;

for ( value in object ) {
  // statements
}


if ( true ) {
  // statements
} else {
  // statements
}
```

```
//Set off operators with spaces.
// bad
var x=y+5;

// good
var x = y + 5;

//End files with a single newline character.
// bad
(function(global) {
  // ...stuff...
})(this);

// bad
(function(global) {
  // ...stuff...
})(this);↵
↵

// good
(function(global) {
  // ...stuff...
})(this);↵
```

Quotes

Whether you prefer single or double quotes shouldn't matter; there is no difference in how JavaScript parses them. However, for the sake of consistency, never mix quotes in the same project. Pick one style and stick with it.

End of lines and empty lines

Whitespace can make it impossible to decipher code diffs and changelists. Many editors allow you to automatically remove extra empty lines and end of lines—you should use these.

Type checking

Checking the type of a variable can be done as follows:

```
//String:
typeof variable === "string"
//Number:
typeof variable === "number"
//Boolean:
typeof variable === "boolean"
//Object:
typeof variable === "object"
//null:
variable === null
//null or undefined:
variable == null
```

Type casting

Perform type coercion at the beginning of the statement as follows:

```
// bad
const totalScore = this.reviewScore + '';
// good
const totalScore = String(this.reviewScore);
```

Use `parseInt()` for Numbers and always with a radix for the type casting:

```
const inputValue = '4';
// bad
const val = new Number(inputValue);
// bad
const val = +inputValue;
// bad
const val = inputValue >> 0;
// bad
const val = parseInt(inputValue);
// good
const val = Number(inputValue);
// good
const val = parseInt(inputValue, 10);
```

The following example shows you how to type cast using Booleans:

```
const age = 0;  // bad
const hasAge = new Boolean(age);  // good
const hasAge = Boolean(age); // good
const hasAge = !!age;
```

Conditional evaluation

There are various stylistic guidelines around conditional statements. Let's study the following code:

```
// When evaluating that array has length,
// WRONG:
if ( array.length > 0 ) ...

// evaluate truthiness(GOOD):
if ( array.length ) ...

// When evaluating that an array is empty,
// (BAD):
if ( array.length === 0 ) ...

// evaluate truthiness(GOOD):
if ( !array.length ) ...

// When checking if string is not empty,
// (BAD):
if ( string !== "" ) ...

// evaluate truthiness (GOOD):
if ( string ) ...

// When checking if a string is empty,
// BAD:
if ( string === "" ) ...

// evaluate falsy-ness (GOOD):
if ( !string ) ...

// When checking if a reference is true,
// BAD:
if ( foo === true ) ...
```

```
// GOOD
if ( foo ) ...

// When checking if a reference is false,
// BAD:
if ( foo === false ) ...

// GOOD
if ( !foo ) ...

// this will also match: 0, "", null, undefined, NaN
// If you MUST test for a boolean false, then use
if ( foo === false ) ...

// a reference that might be null or undefined, but NOT false, "" or
0,
// BAD:
if ( foo === null || foo === undefined ) ...

// GOOD
if ( foo == null ) ...

// Don't complicate matters
return x === 0 ? 'sunday' : x === 1 ? 'Monday' : 'Tuesday';

// Better:
if (x === 0) {
    return 'Sunday';
} else if (x === 1) {
    return 'Monday';
} else {
    return 'Tuesday';
}

// Even Better:
switch (x) {
    case 0:
        return 'Sunday';
    case 1:
        return 'Monday';
    default:
        return 'Tuesday';
}
```

Naming

Naming is super important. I am sure that you have encountered code with terse and undecipherable naming. Let's study the following lines of code:

```
//Avoid single letter names. Be descriptive with your naming.
// bad
function q() {

}

// good
function query() {
}

//Use camelCase when naming objects, functions, and instances.
// bad
const OBJEcT = {};
const this_is_object = {};
function c() {}

// good
const thisIsObject = {};
function thisIsFunction() {}

//Use PascalCase when naming constructors or classes.
// bad
function user(options) {
  this.name = options.name;
}

const bad = new user({
  name: 'nope',
});

// good
class User {
  constructor(options) {
    this.name = options.name;
  }
}
```

```
const good = new User({
  name: 'yup',
});

// Use a leading underscore _ when naming private properties.
// bad
this.__firstName__ = 'Panda';
this.firstName_ = 'Panda';

// good
this._firstName = 'Panda';
```

The eval() method is evil

The `eval()` method, which takes a String containing JavaScript code, compiles it and runs it, is one of the most misused methods in JavaScript. There are a few situations where you will find yourself using `eval()`, for example, when you are building an expression based on the user input.

However, most of the time, `eval()` is used is just because it gets the job done. The `eval()` method is too hacky and makes the code unpredictable. It's slow, unwieldy, and tends to magnify the damage when you make a mistake. If you are considering using `eval()`, then there is probably a better way.

The following snippet shows the usage of `eval()`:

```
console.log(typeof eval(new String("1+1"))); // "object"
console.log(eval(new String("1+1")));        //1+1
console.log(eval("1+1"));                    // 2
console.log(typeof eval("1+1"));             // returns "number"
var expression = new String("1+1");
console.log(eval(expression.toString()));    //2
```

I will refrain from showing other uses of `eval()` and make sure that you are discouraged enough to stay away from it.

The strict mode

ECMAScript 5 has a strict mode that results in cleaner JavaScript, with fewer unsafe features, more warnings, and more logical behavior. The normal (non-strict) mode is also called **sloppy mode**. The strict mode can help you avoid a few sloppy programming practices. If you are starting a new JavaScript project, I would highly recommend that you use the strict mode by default.

You switch on the strict mode by typing the following line first in your JavaScript file or in your `<script>` element:

```
'use strict';
```

Note that JavaScript engines that don't support ECMAScript 5 will simply ignore the preceding statement and continue as non-strict mode.

If you want to switch on the strict mode per function, you can do it as follows:

```
function foo() {
    'use strict';

}
```

This is handy when you are working with a legacy code base where switching on the strict mode everywhere may break things.

If you are working on an existing legacy code, be careful because using the strict mode can break things. There are caveats on this:

Enabling the strict mode for an existing code can break it

The code may rely on a feature that is not available anymore or on behavior that is different in a sloppy mode than in a strict mode. Don't forget that you have the option to add single strict mode functions to files that are in the sloppy mode.

Package with care

When you concatenate and/or minify files, you have to be careful that the strict mode isn't switched off where it should be switched on or vice versa. Both can break code.

The following sections explain the strict mode features in more detail. You normally don't need to know them as you will mostly get warnings for things that you shouldn't do anyway.

Variables must be declared in strict mode

All variables must be explicitly declared in strict mode. This helps to prevent typos. In the sloppy mode, assigning to an undeclared variable creates a global variable:

```
function sloppyFunc() {
  sloppyVar = 123;
} sloppyFunc();  // creates global variable `sloppyVar`
console.log(sloppyVar);  // 123
```

In the strict mode, assigning to an undeclared variable throws an exception:

```
function strictFunc() {
  'use strict';
  strictVar = 123;
}
strictFunc();  // ReferenceError: strictVar is not defined
```

The eval() function is cleaner in strict mode

In strict mode, the `eval()` function becomes less quirky: variables declared in the evaluated string are not added to the scope surrounding `eval()` anymore.

Features that are blocked in strict mode

The with statement is not allowed. (We will discuss this in the book later.) You get a syntax error at compile time (when loading the code).

In the sloppy mode, an integer with a leading zero is interpreted as octal (base 8) as follows:

```
> 010 === 8 true
```

In strict mode, you get a syntax error if you use this kind of literal:

```
function f() {
'use strict';
return 010
}
//SyntaxError: Octal literals are not allowed in
```

Running JSHint

JSHint is a program that flags suspicious usage in programs written in JavaScript. The core project consists of a library itself as well as a **command line interface** (**CLI**) program distributed as a Node module.

If you have Node.js installed, you can install JSHint using `npm` as follows:

```
npm install jshint -g
```

Once JSHint is installed, you can lint a single or multiple JavaScript files. Save the following JavaScript code snippet in the `test.js` file:

```
function f(condition) {
  switch (condition) {
  case 1:
    console.log(1);
  case 2:
    console.log(1);
  }
}
```

When we run the file using JSHint, it will warn us of a missing `break` statement in the switch case as follows:

```
>jshint test.js
test.js: line 4, col 19, Expected a 'break' statement before 'case'.
1 error
```

JSHint is configurable to suit your needs. Check the documentation at `http://jshint.com/docs/` to see how you can customize JSHint according to your project needs. I use JSHint extensively and suggest you start using it. You will be surprised to see how many hidden bugs and stylistic issues you will be able to fix in your code with such a simple tool.

You can run JSHint at the root of your project and lint the entire project. You can place JSHint directives in the `.jshintrc` file. This file may look something as follows:

```
{
     "asi": false,
     "expr": true,
     "loopfunc": true,
     "curly": false,
     "evil": true,
     "white": true,
     "undef": true,
     "indent": 4
}
```

Summary

In this chapter, we set some foundations around JavaScript grammar, types, and stylistic considerations. We have consciously not talked about other important aspects such as functions, variable scopes, and closures primarily because they deserve their own place in this book. I am sure that this chapter helps you understand some of the primary concepts of JavaScript. With these foundations in place, we will take a look at how we can write professional quality JavaScript code.

2
Functions, Closures, and Modules

In the previous chapter, we deliberately did not discuss certain aspects of JavaScript. These are some of the features of the language that give JavaScript its power and elegance. If you are an intermediate- or advanced-level JavaScript programmer, you may be actively using objects and functions. In many cases, however, developers stumble at these fundamental levels and develop a half-baked or sometimes wrong understanding of the core JavaScript constructs. There is generally a very poor understanding of the concept of closures in JavaScript, due to which many programmers cannot use the functional aspects of JavaScript very well. In JavaScript, there is a strong interconnection between objects, functions, and closures. Understanding the strong relationship between these three concepts can vastly improve our JavaScript programming ability, giving us a strong foundation for any type of application development.

Functions are fundamental to JavaScript. Understanding functions in JavaScript is the single most important weapon in your arsenal. The most important fact about functions is that in JavaScript, functions are first-class objects. They are treated like any other JavaScript object. Just like other JavaScript data types, they can be referenced by variables, declared with literals, and even passed as function parameters.

As with any other object in JavaScript, functions have the following capabilities:

- They can be created via literals
- They can be assigned to variables, array entries, and properties of other objects
- They can be passed as arguments to functions
- They can be returned as values from functions
- They can possess properties that can be dynamically created and assigned

We will talk about each of these unique abilities of a JavaScript function in this chapter and the rest of the book.

A function literal

One of the most important concepts in JavaScript is that the functions are the primary unit of execution. Functions are the pieces where you will wrap all your code, hence they will give your programs a structure.

JavaScript functions are declared using a function literal.

Function literals are composed of the following four parts:

- The function keyword.
- An optional name that, if specified, must be a valid JavaScript identifier.
- A list of parameter names enclosed in parentheses. If there are no parameters to the function, you need to provide empty parentheses.
- The body of the function as a series of JavaScript statements enclosed in braces.

A function declaration

The following is a very trivial example to demonstrate all the components of a function declaration:

```
function add(a,b){
  return a+b;
}
c = add(1,2);
console.log(c);  //prints 3
```

The declaration begins with a `function` keyword followed by the function name. The function name is optional. If a function is not given a name, it is said to be anonymous. We will see how anonymous functions are used. The third part is the set of parameters of the function, wrapped in parentheses. Within the parentheses is a set of zero or more parameter names separated by commas. These names will be defined as variables in the function, and instead of being initialized to undefined, they will be initialized to the arguments supplied when the function is invoked. The fourth part is a set of statements wrapped in curly braces. These statements are the body of the function. They are executed when the function is invoked.

This method of function declaration is also known as **function statement**. When you declare functions like this, the content of the function is compiled and an object with the same name as the function is created.

Another way of function declaration is via **function expressions**:

```
var add = function(a,b){
  return a+b;
}
c = add(1,2);
console.log(c);  //prints 3
```

Here, we are creating an anonymous function and assigning it to an `add` variable; this variable is used to invoke the function as in the earlier example. One problem with this style of function declaration is that we cannot have recursive calls to this kind of function. Recursion is an elegant style of coding where the function calls itself. You can use named function expressions to solve this limitation. As an example, refer to the following function to compute the factorial of a given number, `n`:

```
var facto = function factorial(n) {
  if (n <= 1)
    return 1;
  return n * factorial(n - 1);
};
console.log(facto(3));  //prints 6
```

Here, instead of creating an anonymous function, you are creating a named function. Now, because the function has a name, it can call itself recursively.

Finally, you can create self-invoking function expressions (we will discuss them later):

```
(function sayHello() {
  console.log("hello!");
})();
```

Once defined, a function can be called in other JavaScript functions. After the function body is executed, the caller code (that executed the function) continues to execute. You can also pass a function as a parameter to another function:

```
function changeCase(val) {
  return val.toUpperCase();
}
function demofunc(a, passfunction) {
  console.log(passfunction(a));
}
demofunc("smallcase", changeCase);
```

In the preceding example, we are calling the `demofunc()` function with two parameters. The first parameter is the string that we want to convert to uppercase and the second one is the function reference to the `changeCase()` function. In `demofunc()`, we call the `changeCase()` function via its reference passed to the `passfunction` argument. Here we are passing a function reference as an argument to another function. This powerful concept will be discussed in detail later in the book when we discuss callbacks.

A function may or may not return a value. In the previous examples, we saw that the `add` function returned a value to the calling code. Apart from returning a value at the end of the function, calling `return` explicitly allows you to conditionally return from a function:

```
var looper = function(x){
  if (x%5===0) {
    return;
  }
  console.log(x)
}
for(var i=1;i<10;i++){
  looper(i);
}
```

This code snippet prints `1`, `2`, `3`, `4`, `6`, `7`, `8`, and `9`, and not 5. When the `if (x%5===0)` condition is evaluated to true, the code simply returns from the function and the rest of the code is not executed.

Functions as data

In JavaScript, functions can be assigned to variables, and variables are data. You will shortly see that this is a powerful concept. Let's see the following example:

```
var say = console.log;
say("I can also say things");
```

In the preceding example, we assigned the familiar `console.log()` function to the say variable. Any function can be assigned to a variable as shown in the preceding example. Adding parentheses to the variable will invoke it. Moreover, you can pass functions in other functions as parameters. Study the following example carefully and type it in JS Bin:

```
var validateDataForAge = function(data) {
  person = data();
  console.log(person);
  if (person.age <1 || person.age > 99){
    return true;
  }else{
    return false;
  }
};

var errorHandlerForAge = function(error) {
  console.log("Error while processing age");
};

function parseRequest(data,validateData,errorHandler) {
  var error = validateData(data);
  if (!error) {
    console.log("no errors");
  } else {
    errorHandler();
  }
}

var generateDataForScientist = function() {
  return {
    name: "Albert Einstein",
    age : Math.floor(Math.random() * (100 - 1)) + 1,
  };
};
```

```
var generateDataForComposer = function() {
  return {
    name: "J S Bach",
    age : Math.floor(Math.random() * (100 - 1)) + 1,
  };
};

//parse request
parseRequest(generateDataForScientist, validateDataForAge,
errorHandlerForAge);
parseRequest(generateDataForComposer, validateDataForAge,
errorHandlerForAge);
```

In this example, we are passing functions as parameters to a `parseRequest()` function. We are passing different functions for two different calls, `generateDataForScientist` and `generateDataForComposers`, while the other two functions remain the same. You can observe that we defined a generic `parseRequest()`. It takes three functions as arguments, which are responsible for stitching together the specifics: the data, validator, and error handler. The `parseRequest()` function is fully extensible and customizable, and because it will be invoked by every request, there is a single, clean debugging point. I am sure that you have started to appreciate the incredible power that JavaScript functions provide.

Scoping

For beginners, JavaScript scoping is slightly confusing. These concepts may seem straightforward; however, they are not. Some important subtleties exist that must be understood in order to master the concept. So what is Scope? In JavaScript, scope refers to the current context of code.

A variable's scope is the context in which the variable exists. The scope specifies from where you can access a variable and whether you have access to the variable in that context. Scopes can be globally or locally defined.

Global scope

Any variable that you declare is by default defined in global scope. This is one of the most annoying language design decisions taken in JavaScript. As a global variable is visible in all other scopes, a global variable can be modified by any scope. Global variables make it harder to run loosely coupled subprograms in the same program/module. If the subprograms happen to have global variables that share the same names, then they will interfere with each other and likely fail, usually in difficult-to-diagnose ways. This is sometimes known as namespace clash. We discussed global scope in the previous chapter but let's revisit it briefly to understand how best to avoid this.

You can create a global variable in two ways:

- The first way is to place a var statement outside any function. Essentially, any variable declared outside a function is defined in the global scope.
- The second way is to omit the var statement while declaring a variable (also called implied globals). I think this was designed as a convenience for new programmers but turned out to be a nightmare. Even within a function scope, if you omit the var statement while declaring a variable, it's created by default in the global scope. This is nasty. You should always run your program against **ESLint** or **JSHint** to let them flag such violations. The following example shows how global scope behaves:

```
//Global Scope
var a = 1;
function scopeTest() {
  console.log(a);
}
scopeTest();  //prints 1
```

Here we are declaring a variable outside the function and in the global scope. This variable is available in the `scopeTest()` function. If you assign a new value to a global scope variable within a function scope (local), the original value in the global scope is overwritten:

```
//Global Scope
var a = 1;
function scopeTest() {
  a = 2; //Overwrites global variable 2, you omit 'var'
  console.log(a);
}
console.log(a); //prints 1
scopeTest();  //prints 2
console.log(a); //prints 2 (global value is overwritten)
```

Local scope

Unlike most programming languages, JavaScript does not have block-level scope (variables scoped to surrounding curly brackets); instead, JavaScript has function-level scope. Variables declared within a function are local variables and are only accessible within that function or by functions inside that function:

```
var scope_name = "Global";
function showScopeName () {
  // local variable; only accessible in this function
  var scope_name = "Local";
  console.log (scope_name); // Local
}
console.log (scope_name);     //prints - Global
showScopeName();             //prints - Local
```

Function-level scope versus block-level scope

JavaScript variables are scoped at the function level. You can think of this as a small bubble getting created that prevents the variable to be visible from outside this bubble. A function creates such a bubble for variables declared inside the function. You can visualize the bubbles as follows:

```
-GLOBAL SCOPE-------------------------------------------|
var g =0;                                                |
function foo(a) { ----------------------|                |
    var b = 1;                          |                |
    //code                              |                |
    function bar() { ------|            |                |
        // ...             |ScopeBar    | ScopeFoo       |
    }                ------|            |                |
    // code                             |                |
    var c = 2;                          |                |
}---------------------------------------|                |
foo();   //WORKS                                         |
bar();   //FAILS                                         |
---------------------------------------------------------|
```

JavaScript uses scope chains to establish the scope for a given function. There is typically one global scope, and each function defined has its own nested scope. Any function defined within another function has a local scope that is linked to the outer function. *It's always the position in the source that defines the scope*. When resolving a variable, JavaScript starts at the innermost scope and searches outwards. With this, let's look at various scoping rules in JavaScript.

In the preceding crudely drawn visual, you can see that the `foo()` function is defined in the global scope. The `foo()` function has its local scope and access to the `g` variable because it's in the global scope. The `a`, `b`, and `c` variables are available in the local scope because they are defined within the function scope. The `bar()` function is also declared within the function scope and is available within the `foo()` function. However, once the function scope is over, the `bar()` function is not available. You cannot see or call the `bar()` function from outside the `foo()` function—a scope bubble.

Now that the `bar()` function also has its own function scope (bubble), what is available in here? The `bar()` function has access to the `foo()` function and all the variables created in the parent scope of the `foo()` function—`a`, `b`, and `c`. The `bar()` function also has access to the global scoped variable, `g`.

This is a powerful idea. Take a moment to think about it. We just discussed how rampant and uncontrolled global scope can get in JavaScript. How about we take an arbitrary piece of code and wrap it around with a function? We will be able to hide and create a scope bubble around this piece of code. Creating the correct scope using function wrapping will help us create correct code and prevent difficult-to-detect bugs.

Another advantage of the function scope and hiding variables and functions within this scope is that you can avoid collisions between two identifiers. The following example shows such a bad case:

```
function foo() {
  function bar(a) {
    i = 2; // changing the 'i' in the enclosing scope's for-loop
    console.log(a+i);
  }
  for (var i=0; i<10; i++) {
    bar(i); // infinite loop
  }
}
foo();
```

In the `bar()` function, we are inadvertently modifying the value of `i=2`. When we call `bar()` from within the `for` loop, the value of the `i` variable is set to `2` and we never come out of an infinite loop. This is a bad case of namespace collision.

So far, using functions as a scope sounds like a great way to achieve modularity and correctness in JavaScript. Well, though this technique works, it's not really ideal. The first problem is that we must create a named function. If we keep creating such functions just to introduce the function scope, we pollute the global scope or parent scope. Additionally, we have to keep calling such functions. This introduces a lot of boilerplate, which makes the code unreadable over time:

```
var a = 1;
//Lets introduce a function -scope
//1. Add a named function foo() into the global scope
function foo() {
  var a = 2;
  console.log( a ); // 2
}
//2. Now call the named function foo()
foo();
console.log( a ); // 1
```

We introduced the function scope by creating a new function `foo()` to the global scope and called this function later to execute the code.

In JavaScript, you can solve both these problems by creating functions that immediately get executed. Carefully study and type the following example:

```
var a = 1;
//Lets introduce a function -scope
//1. Add a named function foo() into the global scope
(function foo() {
    var a = 2;
    console.log( a ); // 2
})(); //<---this function executes immediately
console.log( a ); // 1
```

Notice that the wrapping function statement starts with `function`. This means that instead of treating the function as a standard declaration, the function is treated as a function expression.

The `(function foo(){ })` statement as an expression means that the identifier `foo` is found only in the scope of the `foo()` function, not in the outer scope. Hiding the name `foo` in itself means that it does not pollute the enclosing scope unnecessarily. This is so useful and far better. We add `()` after the function expression to execute it immediately. So the complete pattern looks as follows:

```
(function foo(){ /* code */ })();
```

This pattern is so common that it has a name: **IIFE**, which stands for **Immediately Invoked Function Expression**. Several programmers omit the function name when they use IIFE. As the primary use of IIFE is to introduce function-level scope, naming the function is not really required. We can write the earlier example as follows:

```
var a = 1;
(function() {
    var a = 2;
    console.log( a ); // 2
})();
console.log( a ); // 1
```

Here we are creating an anonymous function as IIFE. While this is identical to the earlier named IIFE, there are a few drawbacks of using anonymous IIFEs:

- As you can't see the function name in the stack traces, debugging such code is very difficult
- You cannot use recursion on anonymous functions (as we discussed earlier)
- Overusing anonymous IIFEs sometimes results in unreadable code

Douglas Crockford and a few other experts recommend a slight variation of IIFE:

```
(function(){ /* code */ }());
```

Both these IIFE forms are popular and you will see a lot of code using both these variations.

You can pass parameters to IIFEs. The following example shows you how to pass parameters to IIFEs:

```
(function foo(b) {
    var a = 2;
    console.log( a + b );
})(3); //prints 5
```

Inline function expressions

There is another popular usage of inline function expressions where the functions are passed as parameters to other functions:

```
function setActiveTab(activeTabHandler, tab){
  //set active tab
  //call handler
  activeTabHandler();
}
setActiveTab( function (){
  console.log( "Setting active tab" );
}, 1 );
//prints "Setting active tab"
```

Again, you can name this inline function expression to make sure that you get a correct stack trace while you are debugging the code.

Block scopes

As we discussed earlier, JavaScript does not have the concept of block scopes. Programmers familiar with other languages such as Java or C find this very uncomfortable. **ECMAScript 6** (**ES6**) introduces the **let** keyword to introduce traditional block scope. This is so incredibly convenient that if you are sure your environment is going to support ES6, you should always use the `let` keyword. See the following code:

```
var foo = true;
if (foo) {
  let bar = 42; //variable bar is local in this block { }
  console.log( bar );
}
console.log( bar ); // ReferenceError
```

However, as things stand today, ES6 is not supported by default in most popular browsers.

This chapter so far should have given you a fair understanding of how scoping works in JavaScript. If you are still unclear, I would suggest that you stop here and revisit the earlier sections of this chapter. Research your doubts on the Internet or put your questions on Stack Overflow. In short, make sure that you have no doubts related to the scoping rules.

It is very natural for us to think of code execution happening from top to bottom, line by line. This is how most of JavaScript code is executed but with some exceptions.

Consider the following code:

```
console.log( a );
var a = 1;
```

If you said this is an invalid code and will result in `undefined` when we call `console.log()`, you are absolutely correct. However, what about this?

```
a = 1;
var a;
console.log( a );
```

What should be the output of the preceding code? It is natural to expect `undefined` as the `var a` statement comes after `a = 1`, and it would seem natural to assume that the variable is redefined and thus assigned the default `undefined`. However, the output will be `1`.

When you see `var a = 1`, JavaScript splits it into two statements: `var a` and `a = 1`. The first statement, the declaration, is processed during the compilation phase. The second statement, the assignment, is left in place for the execution phase.

So the preceding snippet would actually be executed as follows:

```
var a;   //----Compilation phase

a = 1;    //------execution phase
console.log( a );
```

The first snippet is actually executed as follows:

```
var a;     //-----Compilation phase

console.log( a );
a = 1;     //------execution phase
```

So, as we can see, variable and function declarations are moved up to the top of the code during compilation phase—this is also popularly known as **hoisting**. It is very important to remember that only the declarations themselves are hoisted, while any assignments or other executable logic are left in place. The following snippet shows you how function declarations are hoisted:

```
foo();
function foo() {
  console.log(a); // undefined
  var a = 1;
}
```

The declaration of the `foo()` function is hoisted such that we are able to execute the function before defining it. One important aspect of hoisting is that it works per scope. Within the `foo()` function, declaration of the a variable will be hoisted to the top of the `foo()` function, and not to the top of the program. The actual execution of the `foo()` function with hoisting will be something as follows:

```
function foo() {
  var a;
  console.log(a); // undefined
  a = 1;
}
```

We saw that function declarations are hoisted but function expressions are not. The next section explains this case.

Function declarations versus function expressions

We saw two ways by which functions are defined. Though they both serve identical purposes, there is a difference between these two types of declarations. Check the following example:

```
//Function expression
functionOne();
//Error
//"TypeError: functionOne is not a function

var functionOne = function() {
  console.log("functionOne");
};
//Function declaration
functionTwo();
//No error
//Prints - functionTwo

function functionTwo() {
  console.log("functionTwo");
}
```

A function declaration is processed when execution enters the context in which it appears before any step-by-step code is executed. The function that it creates is given a proper name (`functionTwo()` in the preceding example) and this name is put in the scope in which the declaration appears. As it's processed before any step-by-step code in the same context, calling `functionTwo()` before defining it works without an error.

However, `functionOne()` is an anonymous function expression, evaluated when it's reached in the step-by-step execution of the code (also called runtime execution); we have to declare it before we can invoke it.

So essentially, the function declaration of `functionTwo()` was hoisted while the function expression of `functionOne()` was executed when line-by-line execution encountered it.



Both function declarations and variable declarations are hoisted but functions are hoisted first, and then variables.

One thing to remember is that you should never use function declarations conditionally. This behavior is non-standardized and can behave differently across platforms. The following example shows such a snippet where we try to use function declarations conditionally. We are trying to assign different function body to function `sayMoo()` but such a conditional code is not guaranteed to work across all browsers and can result in unpredictable results:

```
// Never do this - different browsers will behave differently
if (true) {
  function sayMoo() {
    return 'trueMoo';
  }
}
else {
  function sayMoo() {
    return 'falseMoo';
  }
}
foo();
```

However, it's perfectly safe and, in fact, smart to do the same with function expressions:

```
var sayMoo;
if (true) {
  sayMoo = function() {
```

```
    return 'trueMoo';
  };
}
else {
  sayMoo = function() {
    return 'falseMoo';
  };
}
foo();
```

If you are curious to know why you should not use function declarations in conditional blocks, read on; otherwise, you can skip the following paragraph.

Function declarations are allowed to appear only in the program or function body. They cannot appear in a block ({ ... }). Blocks can only contain statements and not function declarations. Due to this, almost all implementations of JavaScript have behavior different from this. It is always advisable to *never* use function declarations in a conditional block.

Function expressions, on the other hand, are very popular. A very common pattern among JavaScript programmers is to fork function definitions based on some kind of a condition. As such forks usually happen in the same scope, it is almost always necessary to use function expressions.

The arguments parameter

The arguments parameter is a collection of all the arguments passed to the function. The collection has a property named `length` that contains the count of arguments, and the individual argument values can be obtained using an array indexing notation. Okay, we lied a bit. The arguments parameter is not a JavaScript array, and if you try to use array methods on arguments, you'll fail miserably. You can think of arguments as an array-like structure. This makes it possible to write functions that take an unspecified number of parameters. The following snippet shows you how you can pass a variable number of arguments to the function and iterate through them using an arguments array:

```
var sum = function () {
  var i, total = 0;
  for (i = 0; i < arguments.length; i += 1) {
    total += arguments[i];
  }
  return total;
};
console.log(sum(1,2,3,4,5,6,7,8,9)); // prints 45
console.log(sum(1,2,3,4,5)); // prints 15
```

As we discussed, the arguments parameter is not really an array; it is possible to convert it to an array as follows:

```
var args = Array.prototype.slice.call(arguments);
```

Once converted to an array, you can manipulate the list as you wish.

The this parameter

Whenever a function is invoked, in addition to the parameters that represent the explicit arguments that were provided on the function call, an implicit parameter named `this` is also passed to the function. It refers to an object that's implicitly associated with the function invocation, termed as a **function context**. If you have coded in Java, the `this` keyword will be familiar to you; like Java, `this` points to an instance of the class in which the method is defined.

Equipped with this knowledge, let's talk about various invocation methods.

Invocation as a function

If a function is not invoked as a method, constructor, or via `apply()` or `call()`, it's simply invoked *as a function*:

```
function add() {}
add();
var substract = function() {

};
substract();
```

When a function is invoked with this pattern, `this` is bound to the global object. Many experts believe this to be a bad design choice. It is natural to assume that `this` would be bound to the parent context. When you are in a situation such as this, you can capture the value of `this` in another variable. We will focus on this pattern later.

Invocation as a method

A method is a function tied to a property on an object. For methods, `this` is bound to the object on invocation:

```
var person = {
  name: 'Albert Einstein',
  age: 66,
  greet: function () {
```

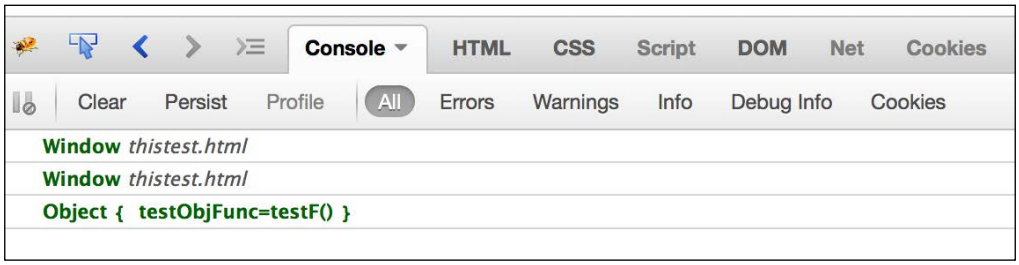
```
      console.log(this.name);
    }
  };
  person.greet();
```

In this example, `this` is bound to the person object on invoking `greet` because `greet` is a method of person. Let's see how this behaves in both these invocation patterns.

Let's prepare this HTML and JavaScript harness:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>This test</title>
  <script type="text/javascript">
    function testF(){ return this; }
    console.log(testF());
    var testFCopy = testF;
    console.log(testFCopy());
    var testObj = {
      testObjFunc: testF
    };
    console.log(testObj.testObjFunc ());
  </script>
</head>
<body>
</body>
</html>
```

In the **Firebug** console, you can see the following output:



The first two method invocations were invocation as a function; hence, the `this` parameter pointed to the global context (`Window`, in this case).

Next, we define an object with a `testObj` variable with a property named `testObjFunc` that receives a reference to `testF()`—don't fret if you are not really aware of object creation yet. By doing this, we created a `testObjMethod()` method. Now, when we invoke this method, we expect the function context to be displayed when we display the value of `this`.

Invocation as a constructor

Constructor functions are declared just like any other functions and there's nothing special about a function that's going to be used as a constructor. However, the way in which they are invoked is very different.

To invoke the function as a constructor, we precede the function invocation with the **new** keyword. When this happens, `this` is bound to the new object.

Before we discuss more, let's take a quick introduction to object orientation in JavaScript. We will, of course, discuss the topic in great detail in the next chapter. JavaScript is a prototypal inheritance language. This means that objects can inherit properties directly from other objects. The language is class-free. Functions that are designed to be called with the `new` prefix are called constructors. Usually, they are named using **PascalCase** as opposed to **CamelCase** for easier distinction. In the following example, notice that the `greet` function uses this to access the `name` property. The `this` parameter is bound to `Person`:

```
var Person = function (name) {
  this.name = name;
};
Person.prototype.greet = function () {
  return this.name;
};
var albert = new Person('Albert Einstein');
console.log(albert.greet());
```

We will discuss this particular invocation method when we study objects in the next chapter.

Invocation using apply() and call() methods

We said earlier that JavaScript functions are objects. Like other objects, they also have certain methods. To invoke a function using its `apply()` method, we pass two parameters to `apply()`: the object to be used as the function context and an array of values to be used as the invocation arguments. The `call()` method is used in a similar manner, except that the arguments are passed directly in the argument list rather than as an array.

Anonymous functions

We introduced you to anonymous functions a bit earlier in this chapter, and as they're a crucial concept, we will take a detailed look at them. For a language inspired by Scheme, anonymous functions are an important logical and structural construct.

Anonymous functions are typically used in cases where the function doesn't need to have a name for later reference. Let's look at some of the most popular usages of anonymous functions.

Anonymous functions while creating an object

An anonymous function can be assigned to an object property. When we do that, we can call that function with a dot (`.`) operator. If you are coming from a Java or other OO language background, you will find this very familiar. In such languages, a function, which is part of a class is generally called with a notation—`Class.function()`. Let's consider the following example:

```
var santa = {
  say :function(){
    console.log("ho ho ho");
  }
}
santa.say();
```

In this example, we are creating an object with a `say` property, which is an anonymous function. In this particular case, this property is known as a method and not a function. We don't need to name this function because we are going to invoke it as the object property. This is a popular pattern and should come in handy.

Anonymous functions while creating a list

Here, we are creating two anonymous functions and adding them to an array. (We will take a detailed look at arrays later.) Then, you loop through this array and execute the functions in a loop:

```
<script type="text/javascript">
var things = [
  function() { alert("ThingOne") },
  function() { alert("ThingTwo") },
];
for(var x=0; x<things.length; x++) {
  things[x]();
}
</script>
```

Anonymous functions as a parameter to another function

This is one of the most popular patterns and you will find such code in most professional libraries:

```
// function statement
function eventHandler(event){
  event();
}

eventHandler(function(){
  //do a lot of event related things
  console.log("Event fired");
});
```

You are passing the anonymous function to another function. In the receiving function, you are executing the function passed as a parameter. This can be very convenient if you are creating single-use functions such as object methods or event handlers. The anonymous function syntax is more concise than declaring a function and then doing something with it as two separate steps.

Anonymous functions in conditional logic

You can use anonymous function expressions to conditionally change behavior. The following example shows this pattern:

```
var shape;
if(shape_name === "SQUARE") {
  shape = function() {
    return "drawing square";
  }
}
else {
  shape = function() {
    return "drawing square";
  }
}
alert(shape());
```

Here, based on a condition, we are assigning a different implementation to the `shape` variable. This pattern can be very useful if used with care. Overusing this can result in unreadable and difficult-to-debug code.

Later in this book, we will look at several functional tricks such as **memoization** and caching function calls. If you have reached here by quickly reading through the entire chapter, I would suggest that you stop for a while and contemplate on what we have discussed so far. The last few pages contain a ton of information and it will take some time for all this information to sink in. I would suggest that you reread this chapter before proceeding further. The next section will focus on closures and the module pattern.

Closures

Traditionally, closures have been a feature of purely functional programming languages. JavaScript shows its affinity with such functional programming languages by considering closures integral to the core language constructs. Closures are gaining popularity in mainstream JavaScript libraries and advanced production code because they let you simplify complex operations. You will hear experienced JavaScript programmers talking almost reverently about closures—as if they are some magical construct far beyond the reach of the intellect that common men possess. However, this is not so. When you study this concept, you will find closures to be very obvious, almost matter-of-fact. Till you reach closure enlightenment, I suggest you read and reread this chapter, research on the Internet, write code, and read JavaScript libraries to understand how closures behave—but do not give up.

The first realization that you must have is that closure is everywhere in JavaScript. It is not a hidden special part of the language.

Before we jump into the nitty-gritty, let's quickly refresh the lexical scope in JavaScript. We discussed in great detail how lexical scope is determined at the function level in JavaScript. Lexical scope essentially determines where and how all identifiers are declared and predicts how they will be looked up during execution.

In a nutshell, closure is the scope created when a function is declared that allows the function to access and manipulate variables that are external to this function. In other words, closures allow a function to access all the variables, as well as other functions, that are in scope when the function itself is declared.

Let's look at some example code to understand this definition:

```
var outer = 'I am outer'; //Define a value in global scope
function outerFn() { //Declare a a function in global scope
  console.log(outer);
}
outerFn(); //prints - I am outer
```

Were you expecting something shiny? No, this is really the most ordinary case of a closure. We are declaring a variable in the global scope and declaring a function in the global scope. In the function, we are able to access the variable declared in the global scope—`outer`. So essentially, the outer scope for the `outerFn()` function is a closure and always available to `outerFn()`. This is a good start but perhaps then you are not sure why this is such a great thing.

Let's make things a bit more complex:

```
var outer = 'Outer'; //Variable declared in global scope
var copy;
function outerFn(){  //Function declared in global scope

  var inner = 'Inner'; //Variable has function scope only, can not be
  //accessed from outside

  function innerFn(){     //Inner function within Outer function,
    //both global context and outer
    //context are available hence can access
    //'outer' and 'inner'
    console.log(outer);
    console.log(inner);
  }
  copy=innerFn;          //Store reference to inner function,
  //because 'copy' itself is declared
  //in global context, it will be available
  //outside also
}
outerFn();
copy();  //Cant invoke innerFn() directly but can invoke via a
//variable declared in global scope
```

Let's analyze the preceding example. In `innerFn()`, the `outer` variable is available as it's part of the global context. We're executing the inner function after the outer function has been executed via copying a reference to the function to a global reference variable, `copy`. When `innerFn()` executes, the scope in `outerFn()` is gone and not visible at the point at which we're invoking the function through the copy variable. So shouldn't the following line fail?

```
console.log(inner);
```

Should the `inner` variable be undefined? However, the output of the preceding code snippet is as follows:

```
"Outer"
"Inner"
```

What phenomenon allows the `inner` variable to still be available when we execute the inner function, long after the scope in which it was created has gone away? When we declared `innerFn()` in `outerFn()`, not only was the function declaration defined, but a closure was also created that encompasses not only the function declaration, but also all the variables that are in scope at the point of the declaration. When `innerFn()` executes, even if it's executed after the scope in which it was declared goes away, it has access to the original scope in which it was declared through its closure.

Let's continue to expand this example to understand how far you can go with closures:

```
var outer='outer';
var copy;
function outerFn() {
  var inner='inner';
  function innerFn(param){
    console.log(outer);
    console.log(inner);
    console.log(param);
    console.log(magic);
  }
  copy=innerFn;
}
console.log(magic); //ERROR: magic not defined
var magic="Magic";
outerFn();
copy("copy");
```

In the preceding example, we have added a few more things. First, we added a parameter to `innerFn()`—just to illustrate that parameters are also part of the closure. There are two important points that we want to highlight.

All variables in an outer scope are included even if they are declared after the function is declared. This makes it possible for the line, `console.log(magic)`, in `innerFn()`, to work.

However, the same line, `console.log(magic)`, in the global scope will fail because even within the same scope, variables not yet defined cannot be referenced.

All these examples were intended to convey a few concepts that govern how closures work. Closures are a prominent feature in the JavaScript language and you can see them in most libraries.

Let's look at some popular patterns around closures.

Timers and callbacks

In implementing timers or callbacks, you need to call the handler asynchronously, mostly at a later point in time. Due to the asynchronous calls, we need to access variables from outside the scope in such functions. Consider the following example:

```
function delay(message) {
  setTimeout( function timerFn(){
    console.log( message );
  }, 1000 );
}
delay( "Hello World" );
```

We pass the inner `timerFn()` function to the built-in library function, `setTimeout()`. However, `timerFn()` has a scope closure over the scope of `delay()`, and hence it can reference the variable message.

Private variables

Closures are frequently used to encapsulate some information as private variables. JavaScript does not allow such encapsulation found in programming languages such as Java or C++, but by using closures, we can achieve similar encapsulation:

```
function privateTest(){
  var points=0;
  this.getPoints=function(){
    return points;
  };
  this.score=function(){
    points++;
  };
}

var private = new privateTest();
private.score();
console.log(private.points); // undefined
console.log(private.getPoints());
```

In the preceding example, we are creating a function that we intend to call as a constructor. In this `privateTest()` function, we are creating a `var points=0` variable as a function-scoped variable. This variable is available only in `privateTest()`. Additionally, we create an accessor function (also called a getter)—`getPoints()`—this method allows us to read the value of only the points variable from outside `privateTest()`, making this variable private to the function. However, another method, `score()`, allows us to modify the value of the private point variable without directly accessing it from outside. This makes it possible for us to write code where a private variable is updated in a controlled fashion. This pattern can be very useful when you are writing libraries where you want to control how variables are accessed based on a contract and pre-established interface.

Loops and closures

Consider the following example of using functions inside loops:

```
for (var i=1; i<=5; i++) {
  setTimeout( function delay(){
    console.log( i );
  }, i*100);
}
```

This snippet should print `1`, `2`, `3`, `4`, and `5` on the console at an interval of 100 ms, right? Instead, it prints `6`, `6`, `6`, `6`, and `6` at an interval of 100 ms. Why is this happening? Here, we encounter a common issue with closures and looping. The `i` variable is being updated after the function is bound. This means that every bound function handler will always print the last value stored in `i`. In fact, the timeout function callbacks are running after the completion of the loop. This is such a common problem that JSLint will warn you if you try to use functions this way inside a loop.

How can we fix this behavior? We can introduce a function scope and local copy of the `i` variable in that scope. The following snippet shows you how we can do this:

```
for (var i=1; i<=5; i++) {
  (function(j){
    setTimeout( function delay(){
      console.log( j );
    }, j*100);
  })( i );
}
```

We pass the `i` variable and copy it to the `j` variable local to the IIFE. The introduction of an IIFE inside each iteration creates a new scope for each iteration and hence updates the local copy with the correct value.

Modules

Modules are used to mimic classes and focus on public and private access to variables and functions. Modules help in reducing the global scope pollution. Effective use of modules can reduce name collisions across a large code base. A typical format that this pattern takes is as follows:

```
Var moduleName=function() {
  //private state
  //private functions
  return {
     //public state
     //public variables
  }
}
```

There are two requirements to implement this pattern in the preceding format:

- There must be an outer enclosing function that needs to be executed at least once.
- This enclosing function must return at least one inner function. This is necessary to create a closure over the private state—without this, you can't access the private state at all.

Check the following example of a module:

```
var superModule = (function (){
  var secret = 'supersecretkey';
  var passcode = 'nuke';

  function getSecret() {
    console.log( secret );
  }

  function getPassCode() {
    console.log( passcode );
  }
```

```
    return {
      getSecret: getSecret,
      getPassCode: getPassCode
    };
  })();
  superModule.getSecret();
  superModule.getPassCode();
```

This example satisfies both the conditions. Firstly, we create an IIFE or a named function to act as an outer enclosure. The variables defined will remain private because they are scoped in the function. We return the public functions to make sure that we have a closure over the private scope. Using IIFE in the module pattern will actually result in a singleton instance of this function. If you want to create multiple instances, you can create named function expressions as part of the module as well.

We will keep exploring various facets of functional aspects of JavaScript and closures in particular. There can be a lot of imaginative uses of such elegant constructs. An effective way to understand various patterns is to study the code of popular libraries and practice writing these patterns in your code.

Stylistic considerations

As in the previous chapter, we will conclude this discussion with certain stylistic considerations. Again, these are generally accepted guidelines and not rules—feel free to deviate from them if you have reason to believe otherwise:

- Use function declarations instead of function expressions:

  ```
  // bad
  const foo = function () {
  };

  // good
  function foo() {
  }
  ```

- Never declare a function in a non-function block (if, while, and so on). Assign the function to a variable instead. Browsers allow you to do it, but they all interpret it differently.
- Never name a parameter `arguments`. This will take precedence over the `arguments` object that is given to every function scope.

Summary

In this chapter, we studied JavaScript functions. In JavaScript, functions play a critical role. We discussed how functions are created and used. We also discussed important ideas of closures and the scope of variables in terms of functions. We discussed functions as a way to create visibility classes and encapsulation.

In the next chapter, we will look at various data structures and data manipulation techniques in JavaScript.

3
Data Structures and Manipulation

Most of the time that you spend in programming, you do something to manipulate data. You process properties of data, derive conclusions based on the data, and change the nature of the data. In this chapter, we will take an exhaustive look at various data structures and data manipulation techniques in JavaScript. With the correct usage of these expressive constructs, your programs will be correct, concise, easy to read, and most probably faster. This will be explained with the help of the following topics:

- Regular expressions
- Exact match
- Match from a class of characters
- Repeated occurrences
- Beginning and end
- Backreferences
- Greedy and lazy quantifiers
- Arrays
- Maps
- Sets
- A matter of style

Regular expressions

If you are not familiar with regular expressions, I request you to spend time learning them. Learning and using regular expressions effectively is one of the most rewarding skills that you will gain. During most of the code review sessions, the first thing that I comment on is how a piece of code can be converted to a single line of **regular expression** (or **RegEx**). If you study popular JavaScript libraries, you will be surprised to see how ubiquitous RegEx are. Most seasoned engineers rely on RegEx primarily because once you know how to use them, they are concise and easy to test. However, learning RegEx will take a significant amount of effort and time. A regular expression is a way to express a pattern to match strings of text. The expression itself consists of terms and operators that allow us to define these patterns. We'll see what these terms and operators consist of shortly.

In JavaScript, there are two ways to create a regular expression: via a regular expression literal and constructing an instance of a `RegExp` object.

For example, if we wanted to create a RegEx that matches the string test exactly, we could use the following RegEx literal:

```
var pattern = /test/;
```

RegEx literals are delimited using forward slashes. Alternatively, we could construct a `RegExp` instance, passing the RegEx as a string:

```
var pattern = new RegExp("test");
```

Both of these formats result in the same RegEx being created in the variable pattern. In addition to the expression itself, there are three flags that can be associated with a RegEx:

- `i`: This makes the RegEx case-insensitive, so `/test/i` matches not only `test`, but also `Test`, `TEST`, `tEsT`, and so on.
- `g`: This matches all the instances of the pattern as opposed to the default of local, which matches the first occurrence only. More on this later.
- `m`: This allows matches across multiple lines that might be obtained from the value of a `textarea` element.

These flags are appended to the end of the literal (for example, `/test/ig`) or passed in a string as the second parameter to the `RegExp` constructor (`new RegExp("test", "ig")`).

The following example illustrates the various flags and how they affect the pattern match:

```
var pattern = /orange/;
console.log(pattern.test("orange")); // true

var patternIgnoreCase = /orange/i;
console.log(patternIgnoreCase.test("Orange")); // true

var patternGlobal = /orange/ig;
console.log(patternGlobal.test("Orange Juice")); // true
```

It isn't very exciting if we can just test whether the pattern matches a string. Let's see how we can express more complex patterns.

Exact match

Any sequence of characters that's not a special RegEx character or operator represents a character literal:

```
var pattern = /orange/;
```

We mean `o` followed by `r` followed by `a` followed by `n` followed by …—you get the point. We rarely use exact match when using RegEx because that is the same as comparing two strings. Exact match patterns are sometimes called simple patterns.

Match from a class of characters

If you want to match against a set of characters, you can place the set inside `[]`. For example, `[abc]` would mean any character `a`, `b`, or `c`:

```
var pattern = /[abc]/;
console.log(pattern.test('a')); //true
console.log(pattern.test('d')); //false
```

You can specify that you want to match anything but the pattern by adding a `^` (caret sign) at the beginning of the pattern:

```
var pattern = /[^abc]/;
console.log(pattern.test('a')); //false
console.log(pattern.test('d')); //true
```

One critical variation of this pattern is a range of values. If we want to match against a sequential range of characters or numbers, we can use the following pattern:

```
var pattern = /[0-5]/;
console.log(pattern.test(3)); //true
console.log(pattern.test(12345)); //true
console.log(pattern.test(9)); //false
console.log(pattern.test(6789)); //false
console.log(/[0123456789]/.test("This is year 2015")); //true
```

Special characters such as `$` and period (`.`) characters either represent matches to something other than themselves or operators that qualify the preceding term. In fact, we've already seen how `[`, `]`, `-`, and `^` characters are used to represent something other than their literal values.

How do we specify that we want to match a literal `[` or `$` or `^` or some other special character? Within a RegEx, the backslash character escapes whatever character follows it, making it a literal match term. So `\[` specifies a literal match to the `[` character rather than the opening of a character class expression. A double backslash (`\\`) matches a single backslash.

In the preceding examples, we saw the `test()` method that returns **true** or **false** based on the pattern matched. There are times when you want to access occurrences of a particular pattern. The `exec()` method comes in handy in such situations.

The `exec()` method takes a string as an argument and returns an array containing all matches. Consider the following example:

```
var strToMatch = 'A Toyota! Race fast, safe car! A Toyota!';
var regExAt = /Toy/;
var arrMatches = regExAt.exec(strToMatch);
console.log(arrMatches);
```

The output of this snippet would be **['Toy'];** if you want all the instances of the pattern `Toy`, you can use the `g` (global) flag as follows:

```
var strToMatch = 'A Toyota! Race fast, safe car! A Toyota!';
var regExAt = /Toy/g;
var arrMatches = regExAt.exec(strToMatch);
console.log(arrMatches);
```

This will return all the occurrences of the word `oyo` from the original text. The String object contains the `match()` method that has similar functionality of the `exec()` method. The `match()` method is called on a String object and the RegEx is passed to it as a parameter. Consider the following example:

```
var strToMatch = 'A Toyota! Race fast, safe car! A Toyota!';
var regExAt = /Toy/;
var arrMatches = strToMatch.match(regExAt);
console.log(arrMatches);
```

In this example, we are calling the `match()` method on the String object. We pass the RegEx as a parameter to the `match()` method. The results are the same in both these cases.

The other String object method is `replace()`. It replaces all the occurrences of a substring with a different string:

```
var strToMatch = 'Blue is your favorite color ?';
var regExAt = /Blue/;
console.log(strToMatch.replace(regExAt, "Red"));
//Output- "Red is your favorite color ?"
```

It is possible to pass a function as a second parameter of the `replace()` method. The `replace()` function takes the matching text as a parameter and returns the text that is used as a replacement:

```
var strToMatch = 'Blue is your favorite color ?';
var regExAt = /Blue/;
console.log(strToMatch.replace(regExAt, function(matchingText){
  return 'Red';
}));
//Output- "Red is your favorite color ?"
```

The String object's `split()` method also takes a RegEx parameter and returns an array containing all the substrings generated after splitting the original string:

```
var sColor = 'sun,moon,stars';
var reComma = /\,/;
console.log(sColor.split(reComma));
//Output - ["sun", "moon", "stars"]
```

We need to add a backslash before the comma because a comma is treated specially in RegEx and we need to escape it if we want to use it literally.

Using simple character classes, you can match multiple patterns. For example, if you want to match `cat`, `bat`, and `fat`, the following snippet shows you how to use simple character classes:

```
var strToMatch = 'wooden bat, smelly Cat,a fat cat';
var re = /[bcf]at/gi;
var arrMatches = strToMatch.match(re);
console.log(arrMatches);
//["bat", "Cat", "fat", "cat"]
```

As you can see, this variation opens up possibilities to write concise RegEx patterns. Take the following example:

```
var strToMatch = 'i1,i2,i3,i4,i5,i6,i7,i8,i9';
var re = /i[0-5]/gi;
var arrMatches = strToMatch.match(re);
console.log(arrMatches);
//["i1", "i2", "i3", "i4", "i5"]
```

In this example, we are matching the numeric part of the matching string with a range `[0-5]`, hence we get a match from `i0` to `i5`. You can also use the negation class ^ to filter the rest of the matches:

```
var strToMatch = 'i1,i2,i3,i4,i5,i6,i7,i8,i9';
var re = /i[^0-5]/gi;
var arrMatches = strToMatch.match(re);
console.log(arrMatches);
//["i6", "i7", "i8", "i9"]
```

Observe how we are negating only the range clause and not the entire expression.

Several character groups have shortcut notations. For example, the shortcut `\d` means the same thing as `[0-9]`:

Notation	Meaning
`\d`	Any digit character
`\w`	An alphanumeric character (word character)
`\s`	Any whitespace character (space, tab, newline, and similar)
`\D`	A character that is not a digit
`\W`	A non-alphanumeric character
`\S`	A non-whitespace character
`.`	Any character except for newline

These shortcuts are valuable in writing concise RegEx. Consider this example:

```
var strToMatch = '123-456-7890';
var re = /[0-9][0-9][0-9]-[0-9][0-9][0-9]/;
var arrMatches = strToMatch.match(re);
console.log(arrMatches);
//["123-456"]
```

This expression definitely looks a bit strange. We can replace `[0-9]` with `\d` and make this a bit more readable:

```
var strToMatch = '123-456-7890';
var re = /\d\d\d-\d\d\d/;
var arrMatches = strToMatch.match(re);
console.log(arrMatches);
//["123-456"]
```

However, you will soon see that there are even better ways to do something like this.

Repeated occurrences

So far, we saw how we can match fixed characters or numeric patterns. Most often, you want to handle certain repetitive natures of patterns also. For example, if I want to match 4 `a`s, I can write `/aaaa/`, but what if I want to specify a pattern that can match any number of `a`s?

Regular expressions provide you with a wide variety of repetition quantifiers. Repetition quantifiers let us specify how many times a particular pattern can occur. We can specify fixed values (characters should appear *n* times) and variable values (characters can appear at least *n* times till they appear *m* times). The following table lists the various repetition quantifiers:

- `?`: Either 0 or 1 occurrence (marks the occurrence as optional)
- `*`: 0 or more occurrences
- `+`: 1 or more occurrences
- `{n}`: Exactly `n` occurrences
- `{n,m}`: Occurrences between `n` and `m`
- `{n,}`: At least an `n` occurrence
- `{,n}`: 0 to `n` occurrences

In the following example, we create a pattern where the character `u` is optional (has 0 or 1 occurrence):

```
var str = /behaviou?r/;
console.log(str.test("behaviour"));
// true
console.log(str.test("behavior"));
// true
```

It helps to read the `/behaviou?r/` expression as 0 or 1 occurrences of character `u`. The repetition quantifier succeeds the character that we want to repeat. Let's try out some more examples:

```
console.log(/'\d+'/.test("'123'")); // true
```

You should read and interpret the `\d+` expression as `'` is a literal character match, `\d` matches characters `[0-9]`, the `+` quantifier will allow one or more occurrences, and `'` is a literal character match.

You can also group character expressions using `()`. Observe the following example:

```
var heartyLaugh = /Ha+(Ha+)+/i;
console.log(heartyLaugh.test("HaHaHaHaHaHaHaaaaaaaaaaaa"));
//true
```

Let's break the preceding expression into smaller chunks to understand what is going on in here:

- `H`: literal character match
- `a+`: 1 or more occurrences of character `a`
- `(`: start of the expression group
- `H`: literal character match
- `a+`: 1 or more occurrences of character `a`
- `)`: end of expression group
- `+`: 1 or more occurrences of expression group (`Ha+`)

Now it is easier to see how the grouping is done. If we have to interpret the expression, it is sometimes helpful to read out the expression, as shown in the preceding example.

Often, you want to match a sequence of letters or numbers on their own and not just as a substring. This is a fairly common use case when you are matching words that are not just part of any other words. We can specify the word boundaries by using the `\b` pattern. The word boundary with `\b` matches the position where one side is a word character (letter, digit, or underscore) and the other side is not. Consider the following examples.

The following is a simple literal match. This match will also be successful if `cat` is part of a substring:

```
console.log(/cat/.test('a black cat')); //true
```

However, in the following example, we define a word boundary by indicating `\b` before the word `cat`—this means that we want to match only if `cat` is a word and not a substring. The boundary is established before `cat`, and hence a match is found on the text, `a black cat`:

```
console.log(/\bcat/.test('a black cat')); //true
```

When we use the same boundary with the word `tomcat`, we get a failed match because there is no word boundary before `cat` in the word `tomcat`:

```
console.log(/\bcat/.test('tomcat')); //false
```

There is a word boundary after the string `cat` in the word `tomcat`, hence the following is a successful match:

```
console.log(/cat\b/.test('tomcat')); //true
```

In the following example, we define the word boundary before and after the word `cat` to indicate that we want `cat` to be a standalone word with boundaries before and after:

```
console.log(/\bcat\b/.test('a black cat')); //true
```

Based on the same logic, the following match fails because there are no boundaries before and after `cat` in the word `concatenate`:

```
console.log(/\bcat\b/.test("concatenate")); //false
```

The `exec()` method is useful in getting information about the match found because it returns an object with information about the match. The object returned from `exec()` has an `index` property that tells us where the successful match begins in the string. This is useful in many ways:

```
var match = /\d+/.exec("There are 100 ways to do this");
console.log(match);
// ["100"]
console.log(match.index);
// 10
```

Alternatives – OR

Alternatives can be expressed using the `|` (pipe) character. For example, `/a|b/` matches either the `a` or `b` character, and `/(ab)+|(cd)+/` matches one or more occurrences of either `ab` or `cd`.

Beginning and end

Frequently, we may wish to ensure that a pattern matches at the beginning of a string or perhaps at the end of a string. The caret character, when used as the first character of the RegEx, anchors the match at the beginning of the string such that `/^test/` matches only if the test substring appears at the beginning of the string being matched. Similarly, the dollar sign (`$`) signifies that the pattern must appear at the end of the string: `/test$/`.

Using both `^` and `$` indicates that the specified pattern must encompass the entire candidate string: `/^test$/`.

Backreferences

After an expression is evaluated, each group is stored for later use. These values are known as backreferences. Backreferences are created and numbered by the order in which opening parenthesis characters are encountered going from left to right. You can think of backreferences as the portions of a string that are successfully matched against terms in the regular expression.

The notation for a backreference is a backslash followed by the number of the capture to be referenced, beginning with 1, such as `\1`, `\2`, and so on.

An example could be `/^([XYZ])a\1/`, which matches a string that starts with any of the `X`, `Y`, or `Z` characters followed by an `a` and followed by whatever character matched the first capture. This is very different from `/[XYZ] a[XYZ]/`. The character following `a` can't be any of `X`, or `Y`, or `Z`, but must be whichever one of those that triggered the match for the first character. Backreferences are used with String's `replace()` method using the special character sequences, `$1`, `$2`, and so on. Suppose that you want to change the `1234 5678` string to `5678 1234`. The following code accomplishes this:

```
var orig = "1234 5678";
var re = /(\d{4}) (\d{4})/;
var modifiedStr = orig.replace(re, "$2 $1");
console.log(modifiedStr); //outputs "5678 1234"
```

In this example, the regular expression has two groups each with four digits. In the second argument of the `replace()` method, `$2` is equal to `5678` and `$1` is equal to `1234`, corresponding to the order in which they appear in the expression.

Greedy and lazy quantifiers

All the quantifiers that we discussed so far are greedy. A greedy quantifier starts looking at the entire string for a match. If there are no matches, it removes the last character in the string and reattempts the match. If a match is not found again, the last character is again removed and the process is repeated until a match is found or the string is left with no characters.

The `\d+` pattern, for example, will match one or more digits. For example, if your string is `123`, a greedy match would match `1`, `12`, and `123`. Greedy pattern `h.+l` would match `hell` in a string `hello`—which is the longest possible string match. As `\d+` is greedy, it will match as many digits as possible and hence the match would be `123`.

In contrast to greedy quantifiers, a lazy quantifier matches as few of the quantified tokens as possible. You can add a question mark (`?`) to the regular expression to make it lazy. A lazy pattern `h.?l` would match `hel` in the string `hello`—which is the shortest possible string.

The `\w*?X` pattern will match zero or more words and then match an `X`. However, a question mark after `*` indicates that as few characters as possible should be matched. For an `abcXXX` string, the match can be `abcX`, `abcXX`, or `abcXXX`. Which one should be matched? As `*?` is lazy, as few characters as possible are matched and hence the match is `abcX`.

With this necessary information, let's try to solve some common problems using regular expressions.

Removing extra white space from the beginning and end of a string is a very common use case. As a String object did not have the `trim()` method until recently, several JavaScript libraries provide and use an implementation of string trimming for older browsers that don't have the `String.trim()` method. The most commonly used approach looks something like the following code:

```
function trim(str) {
  return (str || "").replace(/^\s+|\s+$/g, "");
}
console.log("--"+trim("   test    ")+"--");
//"--test--"
```

What if we want to replace repeated whitespaces with a single whitespace?

```
re=/\s+/g;
console.log('There are    a lot      of spaces'.replace(re,' '));
//"There are a lot of spaces"
```

In the preceding snippet, we are trying to match one or more space character sequences and replacing them with a single space.

As you can see, regular expressions can prove to be a Swiss army knife in your JavaScript arsenal. Careful study and practice will be extremely rewarding for you in the long run.

Arrays

An array is an ordered set of values. You can refer to the array elements with a name and index. These are the three ways to create arrays in JavaScript:

```
var arr = new Array(1,2,3);
var arr = Array(1,2,3);
var arr = [1,2,3];
```

When these values are specified, the array is initialized with them as the array's elements. An array's `length` property is equal to the number of arguments. The bracket syntax is called an array literal. It's a shorter and preferred way to initialize arrays.

You have to use the array literal syntax if you want to initialize an array with a single element and the element happens to be a number. If you pass a single number value to the `Array()` constructor or function, JavaScript considers this parameter as the length of the array, not as a single element:

```
var arr = [10];
var arr = Array(10); // Creates an array with no element, but with
  arr.length set to 10
// The above code is equivalent to
var arr = [];
arr.length = 10;
```

JavaScript does not have an explicit array data type. However, you can use the predefined `Array` object and its methods to work with arrays in your applications. The `Array` object has methods to manipulate arrays in various ways, such as joining, reversing, and sorting them. It has a property to determine the array length and other properties for use with regular expressions.

You can populate an array by assigning values to its elements:

```
var days = [];
days[0] = "Sunday";
days[1] = "Monday";
```

You can also populate an array when you create it:

```
var arr_generic = new Array("A String", myCustomValue, 3.14);
var fruits = ["Mango", "Apple", "Orange"]
```

In most languages, the elements of an array are all required to be of the same type. JavaScript allows an array to contain any type of values:

```
var arr = [
  'string', 42.0, true, false, null, undefined,
  ['sub', 'array'], {object: true}, NaN
];
```

You can refer to elements of an `Array` using the element's index number. For example, suppose you define the following array:

```
var days = ["Sunday", "Monday", "Tuesday"]
```

You then refer to the first element of the array as `colors[0]` and the second element of the array as `colors[1]`. The index of the elements starts with `0`.

JavaScript internally stores array elements as standard object properties, using the array index as the property name. The `length` property is different. The `length` property always returns the index of the last element plus one. As we discussed, JavaScript array indexes are 0-based: they start at `0`, not `1`. This means that the `length` property will be one more than the highest index stored in the array:

```
var colors = [];
colors[30] = ['Green'];
console.log(colors.length); // 31
```

You can also assign to the `length` property. Writing a value that is shorter than the number of stored items truncates the array; writing `0` empties it entirely:

```
var colors = ['Red', 'Blue', 'Yellow'];
console.log(colors.length); // 3
colors.length = 2;
console.log(colors); // ["Red","Blue"] - Yellow has been removed
colors.length = 0;
console.log(colors); // [] the colors array is empty
colors.length = 3;
console.log(colors); // [undefined, undefined, undefined]
```

If you query a non-existent array index, you get `undefined`.

A common operation is to iterate over the values of an array, processing each one in some way. The simplest way to do this is as follows:

```
var colors = ['red', 'green', 'blue'];
for (var i = 0; i < colors.length; i++) {
  console.log(colors[i]);
}
```

The `forEach()` method provides another way of iterating over an array:

```
var colors = ['red', 'green', 'blue'];
colors.forEach(function(color) {
  console.log(color);
});
```

The function passed to `forEach()` is executed once for every item in the array, with the array item passed as the argument to the function. Unassigned values are not iterated in a `forEach()` loop.

The `Array` object has a bunch of useful methods. These methods allow the manipulation of the data stored in the array.

The `concat()` method joins two arrays and returns a new array:

```
var myArray = new Array("33", "44", "55");
myArray = myArray.concat("3", "2", "1");
console.log(myArray);
// ["33", "44", "55", "3", "2", "1"]
```

The `join()` method joins all the elements of an array into a string. This can be useful while processing a list. The default delimiter is a comma (`,`):

```
var myArray = new Array('Red','Blue','Yellow');
var list = myArray.join(" ~ ");
console.log(list);
//"Red ~ Blue ~ Yellow"
```

The `pop()` method removes the last element from an array and returns that element. This is analogous to the `pop()` method of a stack:

```
var myArray = new Array("1", "2", "3");
var last = myArray.pop();
// myArray = ["1", "2"], last = "3"
```

The `push()` method adds one or more elements to the end of an array and returns the resulting length of the array:

```
var myArray = new Array("1", "2");
myArray.push("3");
// myArray = ["1", "2", "3"]
```

The `shift()` method removes the first element from an array and returns that element:

```
var myArray = new Array ("1", "2", "3");
var first = myArray.shift();
// myArray = ["2", "3"], first = "1"
```

The `unshift()` method adds one or more elements to the front of an array and returns the new length of the array:

```
var myArray = new Array ("1", "2", "3");
myArray.unshift("4", "5");
// myArray = ["4", "5", "1", "2", "3"]
```

The `reverse()` method reverses or transposes the elements of an array—the first array element becomes the last and the last becomes the first:

```
var myArray = new Array ("1", "2", "3");
myArray.reverse();
// transposes the array so that myArray = [ "3", "2", "1" ]
```

The `sort()` method sorts the elements of an array:

```
var myArray = new Array("A", "C", "B");
myArray.sort();
// sorts the array so that myArray = [ "A","B","c" ]
```

The `sort()` method can optionally take a callback function to define how the elements are compared. The function compares two values and returns one of three values. Let us study the following functions:

- `indexOf(searchElement[, fromIndex])`: This searches the array for `searchElement` and returns the index of the first match:

  ```
  var a = ['a', 'b', 'a', 'b', 'a','c','a'];
  console.log(a.indexOf('b')); // 1
  // Now try again, starting from after the last match
  console.log(a.indexOf('b', 2)); // 3
  console.log(a.indexOf('1')); // -1, 'q' is not found
  ```

- `lastIndexOf(searchElement[, fromIndex])`: This works like `indexOf()`, but only searches backwards:

  ```
  var a = ['a', 'b', 'c', 'd', 'a', 'b'];
  console.log(a.lastIndexOf('b')); //  5
  // Now try again, starting from before the last match
  console.log(a.lastIndexOf('b', 4)); //  1
  console.log(a.lastIndexOf('z')); //  -1
  ```

Now that we have covered JavaScript arrays in depth, let me introduce you to a fantastic library called **Underscore.js** (`http://underscorejs.org/`). Underscore.js provides a bunch of exceptionally useful functional programming helpers to make your code even more clear and functional.

We will assume that you are familiar with **Node.js**; in this case, install Underscore.js via npm:

```
npm install underscore
```

As we are installing Underscore as a Node module, we will test all the examples by typing them in a `.js` file and running the file on Node.js. You can install Underscore using **Bower** also.

Like jQuery's `$` module, Underscore comes with a _ module defined. You will call all functions using this module reference.

Type the following code in a text file and name it `test_.js`:

```
var _ = require('underscore');
function print(n){
  console.log(n);
}
_.each([1, 2, 3], print);
//prints 1 2 3
```

This can be written as follows, without using `each()` function from underscore library:

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

What you see here is a powerful functional construct that makes the code much more elegant and concise. You can clearly see that the traditional approach is verbose. Many languages such as Java suffer from this verbosity. They are slowly embracing functional paradigms. As JavaScript programmers, it is important for us to incorporate these ideas into our code as much as possible.

The `each()` function we saw in the preceding example iterates over a list of elements, yielding each to an iteratee function in turn. Each invocation of iteratee is called with three arguments (element, index, and list). In the preceding example, the `each()` function iterates over the array `[1,2,3]`, and for each element in the array, the `print` function is called with the array element as the parameter. This is a convenient alternative to the traditional looping mechanism to access all the elements in an array.

The `range()` function creates lists of integers. The start value, if omitted, defaults to `0` and step defaults to `1`. If you'd like a negative range, use a negative step:

```
var _ = require('underscore');
console.log(_.range(10));
// [0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
console.log(_.range(1, 11));
//[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
console.log(_.range(0, 30, 5));
//[ 0, 5, 10, 15, 20, 25 ]
console.log(_.range(0, -10, -1));
//[ 0, -1, -2, -3, -4, -5, -6, -7, -8, -9 ]
console.log(_.range(0));
//[]
```

By default, `range()` populates the array with integers, but with a little trick, you can populate other data types also:

```
console.log(_.range(3).map(function () { return 'a' }) );
[ 'a', 'a', 'a' ]
```

This is a fast and convenient way to create and initialize an array with values. We frequently do this by traditional loops.

The `map()` function produces a new array of values by mapping each value in the list through a transformation function. Consider the following example:

```
var _ = require('underscore');
console.log(_.map([1, 2, 3], function(num){ return num * 3; }));
//[3,6,9]
```

The `reduce()` function reduces a list of values to a single value. The initial state is passed by the iteratee function and each successive step is returned by the iteratee. The following example shows the usage:

```
var _ = require('underscore');
var sum = _.reduce([1, 2, 3], function(memo, num){
  console.log(memo,num);return memo + num; }, 0);
console.log(sum);
```

In this example, the line, `console.log(memo,num);`, is just to make the idea clear. The output will be as follows:

```
0 1
1 2
3 3
6
```

The final output is a sum of *1+2+3=6*. As you can see, two values are passed to the iteratee function. On the first iteration, we call the iteratee function with two values `(0,1)`—the value of the `memo` is defaulted in the call to the `reduce()` function and `1` is the first element of the list. In the function, we sum `memo` and `num` and return the intermediate `sum`, which will be used by the `iterate()` function as a `memo` parameter—eventually, the `memo` will have the accumulated `sum`. This concept is important to understand how the intermediate states are used to calculate eventual results.

The `filter()` function iterates through the entire list and returns an array of all the elements that pass the condition. Take a look at the following example:

```
var _ = require('underscore');
var evens = _.filter([1, 2, 3, 4, 5, 6], function(num){
  return num % 2 == 0; });
console.log(evens);
```

The `filter()` function's iteratee function should return a truth value. The resulting `evens` array contains all the elements that satisfy the truth test.

The opposite of the `filter()` function is `reject()`. As the name suggests, it iterates through the list and ignores elements that satisfy the truth test:

```
var _ = require('underscore');
var odds = _.reject([1, 2, 3, 4, 5, 6], function(num){
  return num % 2 == 0; });
console.log(odds);
//[ 1, 3, 5 ]
```

We are using the same code as the previous example but using the `reject()` method instead of `filter()`—the result is exactly the opposite.

The `contains()` function is a useful little function that returns `true` if the value is present in the list; otherwise, returns `false`:

```
var _ = require('underscore');
console.log(_.contains([1, 2, 3], 3));
//true
```

One very useful function that I have grown fond of is `invoke()`. It calls a specific function on each element in the list. I can't tell you how many times I have used it since I stumbled upon it. Let us study the following example:

```
var _ = require('underscore');
console.log(_.invoke([[5, 1, 7], [3, 2, 1]], 'sort'));
//[ [ 1, 5, 7 ], [ 1, 2, 3 ] ]
```

In this example, the `sort()` method of the `Array` object is called for each element in the array. Note that this would fail:

```
var _ = require('underscore');
console.log(_.invoke(["new","old","cat"], 'sort'));
//[ undefined, undefined, undefined ]
```

This is because the `sort` method is not part of the String object. This, however, would work perfectly:

```
var _ = require('underscore');
console.log(_.invoke(["new","old","cat"], 'toUpperCase'));
//[ 'NEW', 'OLD', 'CAT' ]
```

This is because `toUpperCase()` is a String object method and all elements of the list are of the String type.

The `uniq()` function returns the array after removing all duplicates from the original one:

```
var _ = require('underscore');
var uniqArray = _.uniq([1,1,2,2,3]);
console.log(uniqArray);
//[1,2,3]
```

The `partition()` function splits the array into two; one whose elements satisfy the predicate and the other whose elements don't satisfy the predicate:

```
var _ = require('underscore');
function isOdd(n){
  return n%2==0;
}
console.log(_.partition([0, 1, 2, 3, 4, 5], isOdd));
//[ [ 0, 2, 4 ], [ 1, 3, 5 ] ]
```

The `compact()` function returns a copy of the array without all falsy values (false, null, 0, "", undefined, and NaN):

```
console.log(_.compact([0, 1, false, 2, '', 3]));
```

This snippet will remove all falsy values and return a new array with elements `[1,2,3]`—this is a helpful method to eliminate any value from a list that can cause runtime exceptions.

The `without()` function returns a copy of the array with all instances of the specific values removed:

```
var _ = require('underscore');
console.log(_.without([1,2,3,4,5,6,7,8,9,0,1,2,0,0,1,1],0,1,2));
//[ 3, 4, 5, 6, 7, 8, 9 ]
```

Maps

ECMAScript 6 introduces maps. A map is a simple key-value map and can iterate its elements in the order of their insertion. The following snippet shows some methods of the `Map` type and their usage:

```
var founders = new Map();
founders.set("facebook", "mark");
founders.set("google", "larry");
founders.size; // 2
founders.get("twitter"); // undefined
founders.has("yahoo"); // false

for (var [key, value] of founders) {
  console.log(key + " founded by " + value);
}
// "facebook founded by mark"
// "google founded by larry"
```

Sets

ECMAScript 6 introduces sets. Sets are collections of values and can be iterated in the order of the insertion of their elements. An important characteristic about sets is that a value can occur only once in a set.

The following snippet shows some basic operations on sets:

```
var mySet = new Set();
mySet.add(1);
mySet.add("Howdy");
mySet.add("foo");

mySet.has(1); // true
mySet.delete("foo");
mySet.size; // 2

for (let item of mySet) console.log(item);
// 1
// "Howdy"
```

We discussed briefly that JavaScript arrays are not really arrays in a traditional sense. In JavaScript, arrays are objects that have the following characteristics:

- The `length` property
- The functions that inherit from `Array.prototype` (we will discuss this in the next chapter)
- Special handling for keys that are numeric keys

When we write an array index as numbers, they get converted to strings—`arr[0]` internally becomes `arr["0"]`. Due to this, there are a few things that we need to be aware of when we use JavaScript arrays:

- Accessing array elements by an index is not a constant time operation as it is in, say, C. As arrays are actually key-value maps, the access will depend on the layout of the map and other factors (collisions and others).
- JavaScript arrays are sparse (most of the elements have the default value), which means that the array can have gaps in it. To understand this, look at the following snippet:

  ```
  var testArr=new Array(3);
  console.log(testArr);
  ```

  You will see the output as `[undefined, undefined, undefined]`—`undefined` is the default value stored on the array element.

Consider the following example:

```
var testArr=[];
testArr[3] = 10;
testArr[10] = 3;
console.log(testArr);
// [undefined, undefined, undefined, 10, undefined, undefined,
  undefined, undefined, undefined, undefined, 3]
```

You can see that there are gaps in this array. Only two elements have elements and the rest are gaps with the default value. Knowing this helps you in a couple of things. Using the `for...in` loop to iterate an array can result in unexpected results. Consider the following example:

```
var a = [];
a[5] = 5;
for (var i=0; i<a.length; i++) {
  console.log(a[i]);
}
```

```
// Iterates over numeric indexes from 0 to 5
// [undefined,undefined,undefined,undefined,undefined,5]

for (var x in a) {
  console.log(x);
}
// Shows only the explicitly set index of "5", and ignores 0-4
```

A matter of style

Like the previous chapters, we will spend some time discussing the style considerations while creating arrays.

- Use the literal syntax for array creation:

```
// bad
const items = new Array();
// good
const items = [];
```

- Use `Array#push` instead of a direct assignment to add items to an array:

```
const stack = [];
// bad
stack[stack.length] = 'pushme';
// good
stack.push('pushme');
```

Summary

As JavaScript matures as a language, its tool chain also becomes more robust and effective. It is rare to see seasoned programmers staying away from libraries such as Underscore.js. As we see more advanced topics, we will continue to explore more such versatile libraries that can make your code compact, more readable, and performant. We looked at regular expressions—they are first-class objects in JavaScript. Once you start understanding `RegExp`, you will soon find yourself using more of them to make your code concise. In the next chapter, we will look at JavaScript Object notation and how JavaScript prototypal inheritance is a new way of looking at object-oriented programming.

4
Object-Oriented JavaScript

JavaScript's most fundamental data type is the Object data type. JavaScript objects can be seen as mutable key-value-based collections. In JavaScript, arrays, functions, and RegExp are objects while numbers, strings, and Booleans are object-like constructs that are immutable but have methods. In this chapter, you will learn the following topics:

- Understanding objects
- Instance properties versus prototype properties
- Inheritance
- Getters and setters

Understanding objects

Before we start looking at how JavaScript treats objects, we should spend some time on an object-oriented paradigm. Like most programming paradigms, **object-oriented programming** (**OOP**) also emerged from the need to manage complexity. The main idea is to divide the entire system into smaller pieces that are isolated from each other. If these small pieces can hide as many implementation details as possible, they become easy to use. A classic car analogy will help you understand this very important point about OOP.

When you drive a car, you operate on the interface—the steering, clutch, brake, and accelerator. Your view of using the car is limited by this interface, which makes it possible for us to drive the car. This interface is essentially hiding all the complex systems that really drive the car, such as the internal functioning of its engine, its electronic system, and so on. As a driver, you don't bother about these complexities. A similar idea is the primary driver of OOP. An object hides the complexities of how to implement a particular functionality and exposes a limited interface to the outside world. All other systems can use this interface without really bothering about the internal complexity that is hidden from view. Additionally, an object usually hides its internal state from other objects and prevents its direct modification. This is an important aspect of OOP.

In a large system where a lot of objects call other objects' interfaces, things can go really bad if you allow them to modify the internal state of such objects. OOP operates on the idea that the state of an object is inherently hidden from the outside world and it can be changed only via controlled interface operations.

OOP was an important idea and a definite step forward from the traditional structured programming. However, many feel that OOP is overdone. Most OOP systems define complex and unnecessary class and type hierarchies. Another big drawback was that in the pursuit of hiding the state, OOP considered the object state almost immaterial. Though hugely popular, OOP was clearly flawed in many areas. Still, OOP did have some very good ideas, especially hiding the complexity and exposing only the interface to the outside world. JavaScript picked up a few good ideas and built its object model around them. Luckily, this makes JavaScript objects very versatile. In their seminal work, *Design Patterns: Elements of Reusable Object-Oriented Software*, the *Gang of Four* gave two fundamental principles of a better object-oriented design:

- Program to an interface and not to an implementation
- Object composition over class inheritance

These two ideas are really against how classical OOP operates. The classical style of inheritance operates on inheritance that exposes parent classes to all child classes. Classical inheritance tightly couples children to its parents. There are mechanisms in classical inheritance to solve this problem to a certain extent. If you are using classical inheritance in a language such as Java, it is generally advisable to *program to an interface, not an implementation*. In Java, you can write a decoupled code using interfaces:

```
//programming to an interface 'List' and not implementation
  'ArrayList'
List theList = new ArrayList();
```

Instead of programming to an implementation, you can perform the following:

```
ArrayList theList = new ArrayList();
```

How does programming to an interface help? When you program to the `List` interface, you can call methods only available to the `List` interface and nothing specific to `ArrayList` can be called. Programming to an interface gives you the liberty to change your code and use any other specific child of the `List` interface. For example, I can change my implementation and use `LinkedList` instead of `ArrayList`. You can change your variable to use `LinkedList` instead:

```
List theList = new LinkedList();
```

The beauty of this approach is that if you are using the `List` at 100 places in your program, you don't have to worry about changing the implementation at all these places. As you were programming to the interface and not to the implementation, you were able to write a loosely coupled code. This is an important principle when you are using classical inheritance.

Classical inheritance also has a limitation where you can only enhance the child class within the limit of the parent classes. You can't fundamentally differ from what you have got from the ancestors. This inhibits reuse. Classical inheritance has several other problems as follows:

- Inheritance introduces tight coupling. Child classes have knowledge about their ancestors. This tightly couples a child class with its parent.
- When you subclass from a parent, you don't have a choice to select what you want to inherit and what you don't. *Joe Armstrong* (the inventor of **Erlang**) explains this situation very well—his now famous quote:

  > *"The problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle."*

Behavior of JavaScript objects

With this background, let's explore how JavaScript objects behave. In broad terms, an object contains properties, defined as a key-value pair. A property key (name) can be a string and the value can be any valid JavaScript value. You can create objects using object literals. The following snippet shows you how object literals are created:

```
var nothing = {};
var author = {
```

```
  "firstname": "Douglas",
  "lastname": "Crockford"
}
```

A property's name can be any string or an empty string. You can omit quotes around the property name if the name is a legal JavaScript name. So quotes are required around `first-name` but are optional around `firstname`. Commas are used to separate the pairs. You can nest objects as follows:

```
var author = {
  firstname : "Douglas",
  lastname : "Crockford",
  book : {
    title:"JavaScript- The Good Parts",
    pages:"172"
  }
};
```

Properties of an object can be accessed by using two notations: the array-like notation and dot notation. According to the array-like notation, you can retrieve the value from an object by wrapping a string expression in `[]`. If the expression is a valid JavaScript name, you can use the dot notation using `.` instead. Using `.` is a preferred method of retrieving values from an object:

```
console.log(author['firstname']); //Douglas
console.log(author.lastname);     //Crockford
console.log(author.book.title);   // JavaScript- The Good Parts
```

You will get an `undefined` error if you attempt to retrieve a non-existent value. The following would return `undefined`:

```
console.log(author.age);
```

A useful trick is to use the `||` operator to fill in default values in this case:

```
console.log(author.age || "No Age Found");
```

You can update values of an object by assigning a new value to the property:

```
author.book.pages = 190;
console.log(author.book.pages); //190
```

If you observe closely, you will realize that the object literal syntax that you see is very similar to the JSON format.

Methods are properties of an object that can hold function values as follows:

```
var meetingRoom = {};
meetingRoom.book = function(roomId){
  console.log("booked meeting room -"+roomId);
}
meetingRoom.book("VL");
```

Prototypes

Apart from the properties that we add to an object, there is one default property for almost all objects, called a **prototype**. When an object does not have a requested property, JavaScript goes to its prototype to look for it. The `Object.getPrototypeOf()` function returns the prototype of an object.

Many programmers consider prototypes closely related to objects' inheritance—they are indeed a way of defining object types—but fundamentally, they are closely associated with functions.

Prototypes are used as a way to define properties and functions that will be applied to instances of objects. The prototype's properties eventually become properties of the instantiated objects. Prototypes can be seen as blueprints for object creation. They can be seen as analogous to classes in object-oriented languages. Prototypes in JavaScript are used to write a classical style object-oriented code and mimic classical inheritance. Let's revisit our earlier example:

```
var author = {};
author.firstname = 'Douglas';
author.lastname = 'Crockford';
```

In the preceding code snippet, we are creating an empty object and assigning individual properties. You will soon realize that this is not a very standard way of building objects. If you know OOP already, you will immediately see that there is no encapsulation and the usual class structure. JavaScript provides a way around this. You can use the `new` operator to instantiate an object via constructors. However, there is no concept of a class in JavaScript, and it is important to note that the `new` operator is applied to the constructor function. To clearly understand this, let's look at the following example:

```
//A function that returns nothing and creates nothing
function Player() {}
```

```
//Add a function to the prototype property of the function
Player.prototype.usesBat = function() {
  return true;
}

//We call player() as a function and prove that nothing happens
var crazyBob = Player();
if(crazyBob === undefined){
  console.log("CrazyBob is not defined");
}

//Now we call player() as a constructor along with 'new'
//1. The instance is created
//2. method usesBat() is derived from the prototype of the function
var swingJay = new Player();
if(swingJay && swingJay.usesBat && swingJay.usesBat()){
  console.log("SwingJay exists and can use bat");
}
```

In the preceding example, we have a `player()` function that does nothing. We invoke it in two different ways. The first call of the function is as a normal function and second call is as a constructor—note the use of the `new()` operator in this call. Once the function is defined, we add a `usesBat()` method to it. When this function is called as a normal function, the object is not instantiated and we see `undefined` assigned to `crazyBob`. However, when we call this function with the `new` operator, we get a fully instantiated object, `swingJay`.

Instance properties versus prototype properties

Instance properties are the properties that are part of the object instance itself, as shown in the following example:

```
function Player() {
  this.isAvailable = function() {
    return "Instance method says - he is hired";
  };
}
Player.prototype.isAvailable = function() {
  return "Prototype method says - he is Not hired";
};
var crazyBob = new Player();
console.log(crazyBob.isAvailable());
```

When you run this example, you will see that **Instance method says - he is hired** is printed. The `isAvailable()` function defined in the `Player()` function is called an instance of `Player`. This means that apart from attaching properties via the prototype, you can use the this keyword to initialize properties in a constructor. When we have the same functions defined as an instance property and also as a prototype, the instance property takes precedence. The rules governing the precedence of the initialization are as follows:

- Properties are tied to the object instance from the prototype
- Properties are tied to the object instance in the constructor function

This example brings us to the use of the `this` keyword. It is easy to get confused by the `this` keyword because it behaves differently in JavaScript. In other OO languages such as Java, the `this` keyword refers to the current instance of the class. In JavaScript, the value of `this` is determined by the invocation context of a function and where it is called. Let's see how this behavior needs to be carefully understood:

- When `this` is used in a global context: When `this` is called in a global context, it is bound to the global context. For example, in the case of a browser, the global context is usually `window`. This is true for functions also. If you use `this` in a function that is defined in the global context, it is still bound to the global context because the function is part of the global context:

```
function globalAlias(){
  return this;
}
console.log(globalAlias()); //[object Window]
```

- When `this` is used in an object method: In this case, `this` is assigned or bound to the enclosing object. Note that the enclosing object is the immediate parent if you are nesting the objects:

```
var f = {
  name: "f",
  func: function () {
    return this;
  }
};
console.log(f.func());
//prints -
//[object Object] {
//  func: function () {
//    return this;
//  },
//  name: "f"
//}
```

- When there is no context: A function, when invoked without any object, does not get any context. By default, it is bound to the global context. When you use `this` in such a function, it is also bound to the global context.
- When `this` is used in a constructor function: As we saw earlier, when a function is called with a `new` keyword, it acts as a constructor. In the case of a constructor, `this` points to the object being constructed. In the following example, `f()` is used as a constructor (because it's invoked with a `new` keyword) and hence, `this` is pointing to the new object being created. So when we say `this.member = "f"`, the new member is added to the object being created, in this case, that object happens to be `o`:

```
var member = "global";
function f()
{
  this.member = "f";
}
var o= new f();
console.log(o.member); // f
```

We saw that instance properties take precedence when the same property is defined both as an instance property and prototype property. It is easy to visualize that when a new object is created, the properties of the constructor's prototype are copied over. However, this is not a correct assumption. What actually happens is that the prototype is attached to the object and referred when any property of this object is referred. Essentially, when a property is referenced on an object, either of the following occur:

- The object is checked for the existence of the property. If it's found, the property is returned.
- The associated prototype is checked. If the property is found, it is returned; otherwise, an `undefined` error is returned.

This is an important understanding because, in JavaScript, the following code actually works perfectly:

```
function Player() {
  isAvailable=false;
}
var crazyBob = new Player();
Player.prototype.isAvailable = function() {
  return isAvailable;
};
console.log(crazyBob.isAvailable()); //false
```

This code is a slight variation of the earlier example. We are creating the object first and then attaching the function to its prototype. When you eventually call the `isAvailable()` method on the object, JavaScript goes to its prototype to search for it if it's not found in the particular object (`crazyBob`, in this case). Think of this as *hot code loading*—when used properly, this ability can give you incredible power to extend the basic object framework even after the object is created.

If you are familiar with OOP already, you must be wondering whether we can control the visibility and access of the members of an object. As we discussed earlier, JavaScript does not have classes. In programming languages such as Java, you have access modifiers such as `private` and `public` that let you control the visibility of the class members. In JavaScript, we can achieve something similar using the function scope as follows:

- You can declare private variables using the `var` keyword in a function. They can be accessed by private functions or privileged methods.
- Private functions may be declared in an object's constructor and can be called by privileged methods.
- Privileged methods can be declared with `this.method=function() {}`.
- Public methods are declared with `Class.prototype.method=function() {}`.
- Public properties can be declared with `this.property` and accessed from outside the object.

The following example shows several ways of doing this:

```
function Player(name,sport,age,country){

  this.constructor.noOfPlayers++;

  // Private Properties and Functions
  // Can only be viewed, edited or invoked by privileged members
  var retirementAge = 40;
  var available=true;
  var playerAge = age?age:18;
  function isAvailable(){ return available &&
(playerAge<retirementAge); }
  var playerName=name ? name :"Unknown";
  var playerSport = sport ? sport : "Unknown";
```

```
    // Privileged Methods
    // Can be invoked from outside and can access private members
    // Can be replaced with public counterparts
    this.book=function(){
      if (!isAvailable()){
        this.available=false;
      } else {
        console.log("Player is unavailable");
      }
    };
    this.getSport=function(){ return playerSport; };
    // Public properties, modifiable from anywhere
    this.batPreference="Lefty";
    this.hasCelebGirlfriend=false;
    this.endorses="Super Brand";
  }

  // Public methods - can be read or written by anyone
  // Can only access public and prototype properties
  Player.prototype.switchHands = function(){ this.
  batPreference="righty"; };
  Player.prototype.dateCeleb = function(){ this.hasCelebGirlfriend=true;
  } ;
  Player.prototype.fixEyes = function(){ this.wearGlasses=false; };

  // Prototype Properties - can be read or written by anyone (or
  overridden)
  Player.prototype.wearsGlasses=true;

  // Static Properties - anyone can read or write
  Player.noOfPlayers = 0;


  (function PlayerTest(){
    //New instance of the Player object created.
    var cricketer=new Player("Vivian","Cricket",23,"England");
    var golfer =new Player("Pete","Golf",32,"USA");
    console.log("So far there are " + Player.noOfPlayers + " in the
  guild");
```

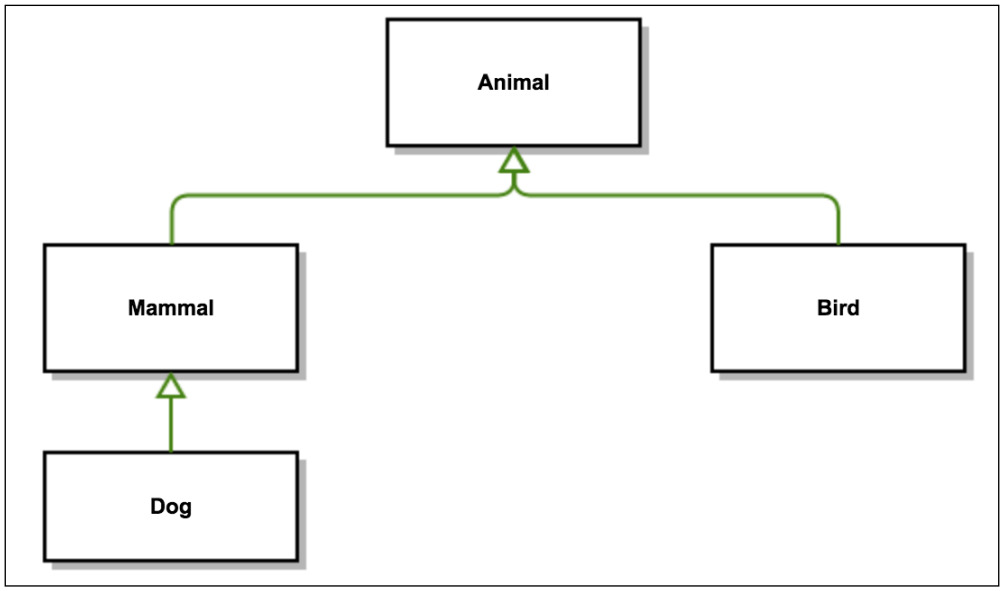
```
  //Both these functions share the common 'Player.prototype.
wearsGlasses' variable
  cricketer.fixEyes();
  golfer.fixEyes();


  cricketer.endorses="Other Brand";//public variable can be updated

  //Both Player's public method is now changed via their prototype
  Player.prototype.fixEyes=function(){
    this.wearGlasses=true;
  };
  //Only Cricketer's function is changed
  cricketer.switchHands=function(){
    this.batPreference="undecided";
  };

})();
```

Let's understand a few important concepts from this example:

- The `retirementAge` variable is a private variable that has no privileged method to get or set its value.
- The `country` variable is a private variable created as a constructor argument. Constructor arguments are available as private variables to the object.
- When we called `cricketer.switchHands()`, it was only applied to the `cricketer` and not to both the players, although it's a prototype function of the `Player` itself.
- Private functions and privileged methods are instantiated with each new object created. In our example, new copies of `isAvailable()` and `book()` would be created for each new player instance that we create. On the other hand, only one copy of public methods is created and shared between all instances. This can mean a bit of performance gain. If you don't *really* need to make something private, think about keeping it public.

Inheritance

Inheritance is an important concept of OOP. It is common to have a bunch of objects implementing the same methods. It is also common to have an almost similar object definition with differences in a few methods. Inheritance is very useful in promoting code reuse. We can look at the following classic example of inheritance relation:



Here, you can see that from the generic **Animal** class, we derive more specific classes such as **Mammal** and **Bird** based on specific characteristics. Both the Mammal and Bird classes do have the same template of an Animal; however, they also define behaviors and attributes specific to them. Eventually, we derive a very specific mammal, **Dog**. A Dog has common attributes and behaviors from an Animal class and Mammal class, while it adds specific attributes and behaviors of a Dog. This can go on to add complex inheritance relationships.

Traditionally, inheritance is used to establish or describe an **IS-A** relationship. For example, a dog IS-A mammal. This is what we know as **classical inheritance**. You would have seen such relationships in object-oriented languages such as C++ and Java. JavaScript has a completely different mechanism to handle inheritance. JavaScript is classless language and uses prototypes for inheritance. Prototypal inheritance is very different in nature and needs thorough understanding. Classical and prototypal inheritance are very different in nature and need careful study.

In classical inheritance, instances inherit from a class blueprint and create subclass relationships. You can't invoke instance methods on a class definition itself. You need to create an instance and then invoke methods on this instance. In prototypal inheritance, on the other hand, instances inherit from other instances.

As far as inheritance is concerned, JavaScript uses only objects. As we discussed earlier, each object has a link to another object called its prototype. This prototype object, in turn, has a prototype of its own, and so on until an object is reached with `null` as its prototype; `null`, by definition, has no prototype, and acts as the final link in this prototype chain.

To understand prototype chains better, let's consider the following example:

```
function Person() {}
Person.prototype.cry = function() {
  console.log("Crying");
}
function Child() {}
Child.prototype = {cry: Person.prototype.cry};
var aChild = new Child();
console.log(aChild instanceof Child);  //true
console.log(aChild instanceof Person); //false
console.log(aChild instanceof Object); //true
```

Here, we define a `Person` and then `Child`—a child IS-A person. We also copy the `cry` property of a `Person` to the `cry` property of `Child`. When we try to see this relationship using `instanceof`, we soon realize that just by copying a behavior, we could not really make `Child` an instance of `Person`; `aChild instanceof Person` fails. This is just copying or masquerading, not inheritance. Even if we copy all the properties of `Person` to `Child`, we won't be inheriting from `Person`. This is usually a bad idea and is shown here only for illustrative purposes. We want to derive a prototype chain—an IS-A relationship, a real inheritance where we can say that child IS-A person. We want to create a chain: a child IS-A person IS-A mammal IS-A animal IS-A object. In JavaScript, this is done using an instance of an object as a prototype as follows:

```
SubClass.prototype = new SuperClass();
Child.prototype = new Person();
```

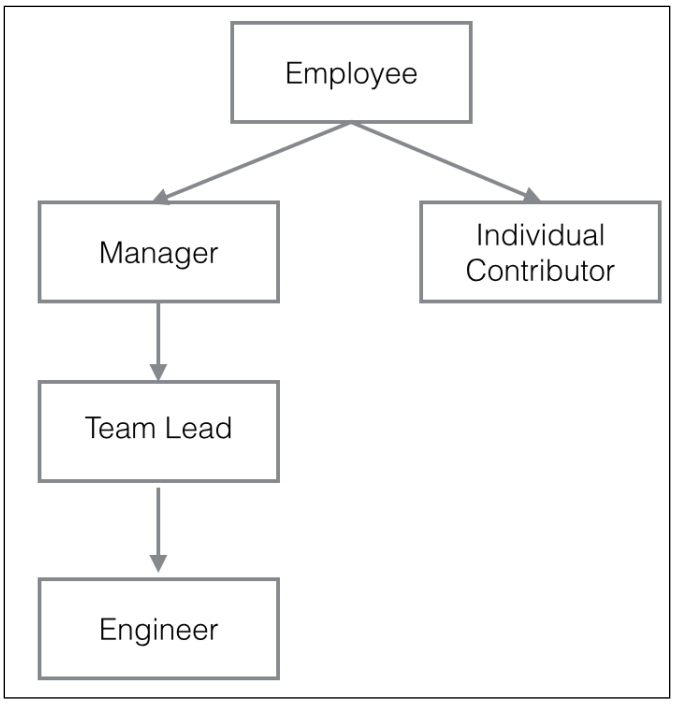
Let's modify the earlier example:

```
function Person() {}
Person.prototype.cry = function() {
  console.log("Crying");
}
```

```
function Child() {}
Child.prototype = new Person();
var aChild = new Child();
console.log(aChild instanceof Child);  //true
console.log(aChild instanceof Person); //true
console.log(aChild instanceof Object); //true
```

The changed line uses an instance of `Person` as the prototype of `Child`. This is an important distinction from the earlier method. Here we are declaring that child IS-A person.

We discussed about how JavaScript looks for a property up the prototype chain till it reaches `Object.prototype`. Let's discuss the concept of prototype chains in detail and try to design the following employee hierarchy:



This is a typical pattern of inheritance. A manager IS-A(n) employee. **Manager** has common properties inherited from an **Employee**. It can have an array of reportees. An **Individual Contributor** is also based on an employee but he does not have any reportees. A **Team Lead** is derived from a Manager with a few functions that are different from a Manager. What we are doing essentially is that each child is deriving properties from its parent (Manager being the parent and Team Lead being the child).

Let's see how we can create this hierarchy in JavaScript. Let's define our `Employee` type:

```
function Employee() {
  this.name = '';
  this.dept = 'None';
  this.salary = 0.00;
}
```

There is nothing special about these definitions. The `Employee` object contains three properties—name, salary, and department. Next, we define `Manager`. This definition shows you how to specify the next object in the inheritance chain:

```
function Manager() {
  Employee.call(this);
  this.reports = [];
}
Manager.prototype = Object.create(Employee.prototype);
```

In JavaScript, you can add a prototypical instance as the value of the prototype property of the constructor function. You can do so at any time after you define the constructor. In this example, there are two ideas that we have not explored earlier. First, we are calling `Employee.call(this)`. If you come from a Java background, this is analogous to the `super()` method call in the constructor. The `call()` method calls a function with a specific object as its context (in this case, it is the given the `this` value), in other words, call allows to specify which object will be referenced by the `this` keyword when the function will be executed. Like `super()` in Java, calling `parentObject.call(this)` is necessary to correctly initialize the object being created.

The other thing we see is `Object.create()` instead of calling `new`. `Object.create()` creates an object with a specified prototype. When we do `new Parent()`, the constructor logic of the parent is called. In most cases, what we want is for `Child.prototype` to be an object that is linked via its prototype to `Parent.prototype`. If the parent constructor contains additional logic specific to the parent, we don't want to run this while creating the child object. This can cause very difficult-to-find bugs. `Object.create()` creates the same prototypal link between the child and parent as the `new` operator without calling the parent constructor.

To have a side effect-free and accurate inheritance mechanism, we have to make sure that we perform the following:

- Setting the prototype to an instance of the parent initializes the prototype chain (inheritance); this is done only once (as the prototype object is shared)
- Calling the parent's constructor initializes the object itself; this is done with every instantiation (you can pass different parameters each time you construct it)

With this understanding in place, let's define the rest of the objects:

```
function IndividualContributor() {
  Employee.call(this);
  this.active_projects = [];
}
IndividualContributor.prototype = Object.create(Employee.prototype);

function TeamLead() {
  Manager.call(this);
  this.dept = "Software";
  this.salary = 100000;
}
TeamLead.prototype = Object.create(Manager.prototype);

function Engineer() {
  TeamLead.call(this);
  this.dept = "JavaScript";
  this.desktop_id = "8822" ;
  this.salary = 80000;
}
Engineer.prototype = Object.create(TeamLead.prototype);
```

Based on this hierarchy, we can instantiate these objects:

```
var genericEmployee = new Employee();
console.log(genericEmployee);
```

You can see the following output for the preceding code snippet:

```
[object Object] {
  dept: "None",
  name: "",
  salary: 0
}
```

A generic `Employee` has a department assigned to `None` (as specified in the default value) and the rest of the properties are also assigned as the default ones.

Next, we instantiate a manager; we can provide specific values as follows:

```
var karen = new Manager();
karen.name = "Karen";
karen.reports = [1,2,3];
console.log(karen);
```

You will see the following output:

```
[object Object] {
  dept: "None",
  name: "Karen",
  reports: [1, 2, 3],
  salary: 0
}
```

For `TeamLead`, the `reports` property is derived from the base class (Manager in this case):

```
var jason = new TeamLead();
jason.name = "Json";
console.log(jason);
```

You will see the following output:

```
[object Object] {
  dept: "Software",
  name: "Json",
  reports: [],
  salary: 100000
}
```

When JavaScript processes the new operator, it creates a new object and passes this object as the value of `this` to the parent—the `TeamLead` constructor. The constructor function sets the value of the `projects` property and implicitly sets the value of the internal `__proto__` property to the value of `TeamLead.prototype`. The `__proto__` property determines the prototype chain used to return property values. This process does not set values for properties inherited from the prototype chain in the `jason` object. When the value of a property is read, JavaScript first checks to see whether the value exists in that object. If the value does exist, this value is returned. If the value is not there, JavaScript checks the prototype chain using the `__proto__` property. Having said this, what happens when you do the following:

```
Employee.prototype.name = "Undefined";
```

It does not propagate to all the instances of `Employee`. This is because when you create an instance of the `Employee` object, this instance gets a local value for the name. When you set the `TeamLead` prototype by creating a new `Employee` object, `TeamLead.prototype` has a local value for the `name` property. Therefore, when JavaScript looks up the `name` property of the `jason` object, which is an instance of `TeamLead`), it finds the local value for this property in `TeamLead.prototype`. It does not try to do further lookups up the chain to `Employee.prototype`.

If you want the value of a property changed at runtime and have the new value be inherited by all the descendants of the object, you cannot define the property in the object's constructor function. To achieve this, you need to add it to the constructor's prototype. For example, let's revisit the earlier example but with a slight change:

```
function Employee() {
  this.dept = 'None';
  this.salary = 0.00;
}
Employee.prototype.name = '';
function Manager() {
  this.reports = [];
}
Manager.prototype = new Employee();
var sandy = new Manager();
var karen = new Manager();

Employee.prototype.name = "Junk";

console.log(sandy.name);
console.log(karen.name);
```

You will see that the `name` property of both sandy and karen has changed to `Junk`. This is because the `name` property is declared outside the constructor function. So, when you change the value of `name` in the Employee's prototype, it propagates to all the descendants. In this example, we are modifying Employee's prototype after the `sandy` and `karen` objects are created. If you changed the prototype before the `sandy` and `karen` objects were created, the value would still have changed to `Junk`.

All native JavaScript objects—Object, Array, String, Number, RegExp, and Function—have prototype properties that can be extended. This means that we can extend the functionality of the language itself. For example, the following snippet extends the `String` object to add a `reverse()` method to reverse a string. This method does not exist in the native String object but by manipulating String's prototype, we add this method to String:

```
String.prototype.reverse = function() {
  return Array.prototype.reverse.apply(this.split('')).join('');
};
var str = 'JavaScript';
console.log(str.reverse()); //"tpircSavaJ"
```

Though this is a very powerful technique, care should be taken not to overuse it. Refer to `http://perfectionkills.com/extending-native-builtins/` to understand the pitfalls of extending native built-ins and what care should be taken if you intend to do so.

Getters and setters

Getters are convenient methods to get the value of specific properties; as the name suggests, **setters** are methods that set the value of a property. Often, you may want to derive a value based on some other values. Traditionally, getters and setters used to be functions such as the following:

```
var person = {
  firstname: "Albert",
  lastname: "Einstein",
  setLastName: function(_lastname){
    this.lastname= _lastname;
  },
  setFirstName: function (_firstname){
    this.firstname= _firstname;
  },
  getFullName: function (){
    return this.firstname + ' '+ this.lastname;
  }
};
person.setLastName('Newton');
person.setFirstName('Issac');
console.log(person.getFullName());
```

As you can see, `setLastName()`, `setFirstName()`, and `getFullName()` are functions used to do *get* and *set* of properties. `Fullname` is a derived property by concatenating the `firstname` and `lastname` properties. This is a very common use case and ECMAScript 5 now provides you with a default syntax for getters and setters.

The following example shows you how getters and setters are created using the object literal syntax in ECMAScript 5:

```
var person = {
  firstname: "Albert",
  lastname: "Einstein",
  get fullname() {
    return this.firstname +" "+this.lastname;
  },
  set fullname(_name){
    var words = _name.toString().split(' ');
    this.firstname = words[0];
    this.lastname = words[1];
  }
};
person.fullname = "Issac Newton";
console.log(person.firstname); //"Issac"
console.log(person.lastname);  //"Newton"
console.log(person.fullname);  //"Issac Newton"
```

Another way of declaring getters and setters is using the `Object.defineProperty()` method:

```
var person = {
  firstname: "Albert",
  lastname: "Einstein",
};
Object.defineProperty(person, 'fullname', {
  get: function() {
    return this.firstname + ' ' + this.lastname;
  },
  set: function(name) {
    var words = name.split(' ');
    this.firstname = words[0];
    this.lastname = words[1];
  }
});
person.fullname = "Issac Newton";
console.log(person.firstname); //"Issac"
console.log(person.lastname);  //"Newton"
console.log(person.fullname);  //"Issac Newton"
```

In this method, you can call `Object.defineProperty()` even after the object is created.

Now that you have tasted the object-oriented flavor of JavaScript, we will go through a bunch of very useful utility methods provided by **Underscore.js**. We discussed the installation and basic usage of Underscore.js in the previous chapter. These methods will make common operations on objects very easy:

- `keys()`: This method retrieves the names of an object's own enumerable properties. Note that this function does not traverse up the prototype chain:

```
var _ = require('underscore');
var testobj = {
  name: 'Albert',
  age : 90,
  profession: 'Physicist'
};
console.log(_.keys(testobj));
//[ 'name', 'age', 'profession' ]
```

- `allKeys()`: This method retrieves the names of an object's own and inherited properties:

```
var _ = require('underscore');
function Scientist() {
  this.name = 'Albert';
}
Scientist.prototype.married = true;
aScientist = new Scientist();
console.log(_.keys(aScientist)); //[ 'name' ]
console.log(_.allKeys(aScientist));//[ 'name', 'married' ]
```

- `values()`: This method retrieves the values of an object's own properties:

```
var _ = require('underscore');
function Scientist() {
  this.name = 'Albert';
}
Scientist.prototype.married = true;
aScientist = new Scientist();
console.log(_.values(aScientist)); //[ 'Albert' ]
```

- `mapObject()`: This method transforms the value of each property in the object:

```
var _ = require('underscore');
function Scientist() {
  this.name = 'Albert';
  this.age = 90;
}
```

```
aScientist = new Scientist();
var lst = _.mapObject(aScientist, function(val,key){
  if(key==="age"){
    return val + 10;
  } else {
    return val;
  }
});
console.log(lst); //{ name: 'Albert', age: 100 }
```

- `functions()`: This returns a sorted list of the names of every method in an object—the name of every function property of the object.
- `pick()`: This function returns a copy of the object, filtered to just the values of the keys provided:

```
var _ = require('underscore');
var testobj = {
  name: 'Albert',
  age : 90,
  profession: 'Physicist'
};
console.log(_.pick(testobj, 'name','age')); //{ name: 'Albert',
age: 90 }
console.log(_.pick(testobj, function(val,key,object){
  return _.isNumber(val);
})); //{ age: 90 }
```

- `omit()`: This function is an invert of `pick()`—it returns a copy of the object, filtered to omit the values for the specified keys.

Summary

JavaScript applications can improve in clarity and quality by allowing for the greater degree of control and structure that object-orientation can bring to the code. JavaScript object-orientation is based on the function prototypes and prototypal inheritance. These two ideas can provide an incredible amount of wealth to developers.

In this chapter, we saw basic object creation and manipulation. We looked at how constructor functions are used to create objects. We dived into prototype chains and how inheritance operates on the idea of prototype chains. These foundations will be used to build your knowledge of JavaScript patterns that we will explore in the next module. We will discuss various testing and debugging techniques in the next chapter.

5
Testing and Debugging

As you write JavaScript applications, you will soon realize that having a sound testing strategy is indispensable. In fact, not writing enough tests is almost always a bad idea. It is essential to cover all the non-trivial functionality of your code to make sure of the following points:

- The existing code behaves as per the specifications
- Any new code does not break the behavior defined by the specifications

Both these points are very important. Many engineers consider only the first point the sole reason to cover your code with enough tests. The most obvious advantage of test coverage is to really make sure that the code being pushed to the production system is mostly error-free. Writing test cases to smartly cover the maximum functional areas of the code generally gives you a good indication about the overall quality of the code. There should be no arguments or compromises around this point. It is unfortunate though that many production systems are still bereft of adequate code coverage. It is very important to build an engineering culture where developers think about writing tests as much as they think about writing code.

The second point is even more important. Legacy systems are usually very difficult to manage. When you are working on code written either by someone else or a large distributed team, it is fairly easy to introduce bugs and break things. Even the best engineers make mistakes. When you are working on a large code base that you are unfamiliar with and if there is no sound test coverage to help you, you will introduce bugs. As you won't have the confidence in the changes that you are making (because there are no test cases to confirm your changes), your code releases will be shaky, slow, and obviously full of hidden bugs.

You will refrain from refactoring or optimizing your code because you won't really be sure what changes to the code base would potentially break something (again, because there are no test cases to confirm your changes)—all this is a vicious circle. It's like a civil engineer saying, "though I have constructed this bridge, I have no confidence in the quality of the construction. It may collapse immediately or never." Though this may sound like an exaggeration, I have seen a lot of high impact production code being pushed with no test coverage. This is risky and should be avoided. When you are writing enough test cases to cover majority of your functional code and when you make a change to these pieces, you immediately realize if there is a problem with this new change. If your changes make the test case fail, you realize the problem. If your refactoring breaks the test scenario, you realize the problem—all this happens much before the code is pushed to production.

In recent years, ideas such as test-driven development and self-testing code are gaining prominence, especially in **agile methodology**. These are fundamentally sound ideas and will help you write robust code—code that you are confident of. We will discuss all these ideas in this chapter. You will understand how to write good test cases in modern JavaScript. We will also look at several tools and methods to debug your code. JavaScript has been traditionally a bit difficult to test and debug primarily due to lack of tools, but modern tools make both of these easy and natural.

Unit testing

When we talk about test cases, we mostly mean **unit tests**. It is incorrect to assume that the unit that we want to test is always a function. The unit (or unit of work) is a logical unit that constitutes a single behavior. This unit should be able to be invoked via a public interface and should be testable independently.

Thus, a unit test performs the following functions:

- It tests a single logical function
- It can be run without a specific order of execution
- It takes care of its own dependencies and mock data
- It always returns the same result for the same input
- It should be self-explanatory, maintainable, and readable



Martin Fowler advocates the **test pyramid** (`http://martinfowler.com/bliki/TestPyramid.html`) strategy to make sure that we have a high number of unit tests to ensure maximum code coverage. The test pyramid says that you should write many more low-level unit tests than higher level integration and UI tests.

There are two important testing strategies that we will discuss in this chapter.

Test-driven development

Test-driven development (**TDD**) has gained a lot of prominence in the last few years. The concept was first proposed as part of the **Extreme Programming** methodology. The idea is to have short repetitive development cycles where the focus is on writing the test cases first. The cycle looks as follows:

1. Add a test case as per the specifications for a specific unit of code.
2. Run the existing suite of test cases to see if the new test case that you wrote fails—it should (because there is no code for this unit yet). This step ensures that the current test harness works well.
3. Write the code that serves mainly to confirm the test case. This code is not optimized or refactored or even entirely correct. However, this is fine at the moment.
4. Rerun the tests and see if all the test cases pass. After this step, you will be confident that the new code is not breaking anything.
5. Refactor the code to make sure that you are optimizing the unit and handling all corner cases.

These steps are repeated for all the new code that you add. This is an elegant strategy that works really well for the agile methodology. TDD will be successful only if the testable units of code are small and confirm only to the test case and nothing more. It is important to write small, modular, and precise code units that have input and output confirming the test case.

Behavior-driven development

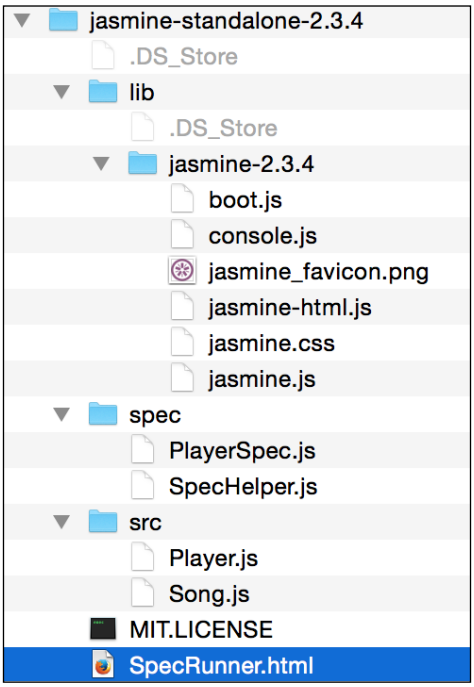
A very common problem while trying to follow TDD is vocabulary and the definition of *correctness*. BDD tries to introduce a *ubiquitous language* while writing the test cases when you are following TDD. This language makes sure that both the business and engineering teams are talking about the same thing.

We will use **Jasmine** as the primary BDD framework and explore various testing strategies.



You can install Jasmine by downloading the standalone package from `https://github.com/jasmine/jasmine/releases/download/v2.3.4/jasmine-standalone-2.3.4.zip`.

When you unzip this package, you will have the following directory structure:



The `lib` directory contains the JavaScript files that you need in your project to start writing Jasmine test cases. If you open `SpecRunner.html`, you will find the following JavaScript files included in it:

```
<script src="lib/jasmine-2.3.4/jasmine.js"></script>
<script src="lib/jasmine-2.3.4/jasmine-html.js"></script>
<script src="lib/jasmine-2.3.4/boot.js"></script>

<!-- include source files here... -->
<script src="src/Player.js"></script>
```

```
<script src="src/Song.js"></script>
<!-- include spec files here... -->
<script src="spec/SpecHelper.js"></script>
<script src="spec/PlayerSpec.js"></script>
```

The first three are Jasmine's own framework files. The next section includes the source files that we want to test and the actual test specifications.

Let's experiment with Jasmine with a very ordinary example. Create a `bigfatjavascriptcode.js` file and place it in the `src/` directory. We will test the following function:

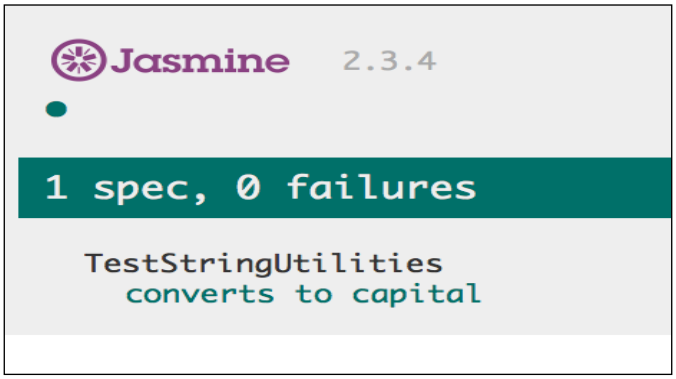
```
function capitalizeName(name){
  return name.toUpperCase();
}
```

This is a simple function that does one single thing. It receives a string and returns a capitalized string. We will test various scenarios around this function. This is the unit of code that we discussed earlier.

Next, create the test specifications. Create one JavaScript file, `test.spec.js`, and place it in the `spec/` directory. The file should contain the following. You will need to add the following two lines to `SpecRunner.html`:

```
<script src="src/bigfatjavascriptcode.js"></script>
<script src="spec/test.spec.js"></script>
```

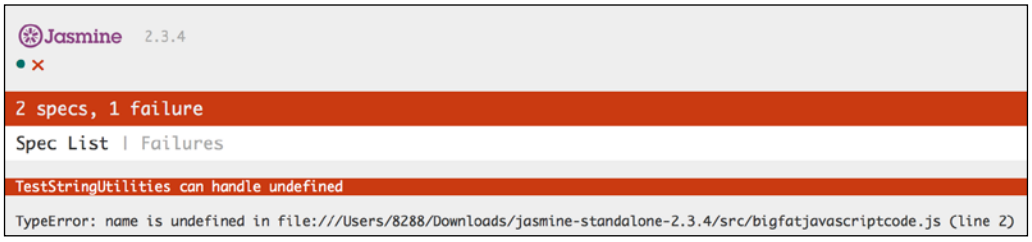
The order of this inclusion does not matter. When we run `SpecRunner.html`, you will see something as follows:



This is the Jasmine report that shows the details about the number of tests that were executed and the count of failures and successes. Now, let's make the test case fail. We want to test a case where an undefined variable is passed to the function. Add one more test case as follows:

```
it("can handle undefined", function() {
  var str= undefined;
  expect(capitalizeName(str)).toEqual(undefined);
});
```

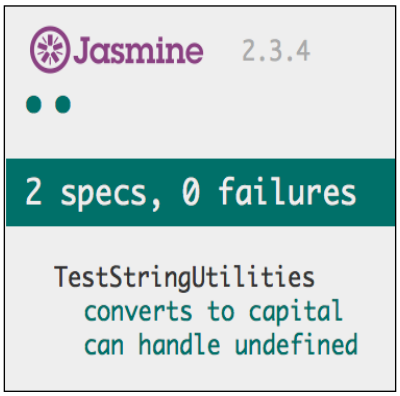
Now, when you run `SpecRunner.html`, you will see the following result:



As you can see, the failure is displayed for this test case in a detailed error stack. Now, we go about fixing this. In your original JavaScript code, we can handle an undefined condition as follows:

```
function capitalizeName(name){
  if(name){
    return name.toUpperCase();
  }
}
```

With this change, your test case will pass and you will see the following in the Jasmine report:



This is very similar to what a test-driven development would look. You write test cases, you then fill in the necessary code to confirm to the specifications, and rerun the test suite. Let's understand the structure of the Jasmine tests.

Our test specification looks as follows:

```
describe("TestStringUtilities", function() {
  it("converts to capital", function() {
    var str = "albert";
    expect(capitalizeName(str)).toEqual("ALBERT");
  });
  it("can handle undefined", function() {
    var str= undefined;
    expect(capitalizeName(str)).toEqual(undefined);
  });
});
```

The `describe("TestStringUtilities"` is a test suite. The name of the test suite should describe the unit of code that we are testing—this can be a function or group of related functionality. In the specifications, you call the global Jasmine `it` function to which you pass the title of the specification and test function used by the test case. This function is the actual test case. You can catch one or more assertions or the general expectations using the `expect` function. When all expectations are `true`, your specification is passed. You can write any valid JavaScript code in the `describe` and `it` functions. The values that you verify as part of the expectations are matched using a matcher. In our example, `toEqual()` is the matcher that matches two values for equality. Jasmine contains a rich set of matches to suit most of the common use cases. Some common matchers supported by Jasmine are as follows:

- `toBe()`: This matcher checks whether two objects being compared are equal. This is the same as the `===` comparison, as shown in the following code:

```
var a = { value: 1};
var b = { value: 1 };

expect(a).toEqual(b);  // success, same as == comparison
expect(b).toBe(b);     // failure, same as === comparison
expect(a).toBe(a);     // success, same as === comparison
```

- `not`: You can negate a matcher with a `not` prefix. For example, `expect(1).not.toEqual(2);` will negate the match made by `toEqual()`.

- `toContain()`: This checks whether an element is part of an array. This is not an exact object match as `toBe()`. For example, look at the following code:

```
expect([1, 2, 3]).toContain(3);
expect("astronomy is a science").toContain("science");
```

- `toBeDefined()` and `toBeUndefined()`: These two matches are handy to check whether a variable is undefined (or not).
- `toBeNull()`: This checks whether a variable's value is `null`.
- `toBeGreaterThan()` and `toBeLessThan()`: These matchers perform numeric comparisons (they work on strings too):

```
expect(2).toBeGreaterThan(1);
expect(1).toBeLessThan(2);
expect("a").toBeLessThan("b");
```

One interesting feature of Jasmine is the **spies**. When you are writing a large system, it is not possible to make sure that all systems are always available and correct. At the same time, you don't want your unit tests to fail due to a dependency that may be broken or unavailable. To simulate a situation where all dependencies are available for a unit of code that we want to test, we mock these dependencies to always give the response that we expect. Mocking is an important aspect of testing and most testing frameworks provide support for the mocking. Jasmine allows mocking using a feature called a spy. Jasmine spies essentially stub the functions that we may not have ready; at the time of writing the test case but as part of the functionality, we need to track that we are executing these dependencies and not ignoring them. Consider the following example:

```
describe("mocking configurator", function() {
  var configurator = null;
  var responseJSON = {};

  beforeEach(function() {
    configurator = {
      submitPOSTRequest: function(payload) {
        //This is a mock service that will eventually be replaced
        //by a real service
        console.log(payload);
        return {"status": "200"};
      }
    };
    spyOn(configurator,
      'submitPOSTRequest').and.returnValue({"status": "200"});
    configurator.submitPOSTRequest({
```

```
        "port":"8000",
        "client-encoding":"UTF-8"
      });
    });

    it("the spy was called", function() {
      expect(configurator.submitPOSTRequest).toHaveBeenCalled();
    });

    it("the arguments of the spy's call are tracked", function() {
      expect(configurator.submitPOSTRequest).toHaveBeenCalledWith({"port
":"8000","client-encoding":"UTF-8"});
    });
  });
```

In this example, while we are writing this test case, we either don't have the real implementation of the `configurator.submitPOSTRequest()` dependency or someone is fixing this particular dependency. In any case, we don't have it available. For our test to work, we need to mock it. Jasmine spies allow us to replace a function with its mock and track its execution.

In this case, we need to ensure that we called the dependency. When the actual dependency is ready, we will revisit this test case to make sure that it fits the specifications, but at this time, all that we need to ensure is that the dependency is called. The Jasmine `tohaveBeenCalled()` function lets us track the execution of a function, which may be a mock. We can use `toHaveBeenCalledWith()` that allows us to determine if the stub function was called with the correct parameters. There are several other interesting scenarios that you can create using Jasmine spies. The scope of this chapter won't permit us to cover them all, but I would encourage you to discover these areas on your own.



You can refer to the user manual for Jasmine for more information on Jasmine spies at `http://jasmine.github.io/2.0/introduction.html`.



Mocha, Chai, and Sinon

Though Jasmine is the most prominent JavaScript testing framework, **Mocha** and **Chai** are gaining prominence in the Node.js environment. Mocha is the testing framework used to describe and run test cases. Chai is the assertion library supported by Mocha. **Sinon.JS** comes in handy while creating mocks and stubs for your tests. We won't discuss these frameworks in this book, but experience on Jasmine will be handy if you want to experiment with these frameworks.

JavaScript debugging

If you are not a completely new programmer, I am sure you must have spent some amount of time debugging your or someone else's code. Debugging is almost like an art form. Every language has different methods and challenges around debugging. JavaScript has traditionally been a difficult language to debug. I have personally spent days and nights of misery trying to debug badly-written JavaScript code using `alert()` functions. Fortunately, modern browsers such as Mozilla Firefox and Google Chrome have excellent developer tools to help debug JavaScript in the browser. There are IDEs like **IntelliJ WebStorm** with great debugging support for JavaScript and Node.js. In this chapter, we will focus primarily on Google Chrome's built-in developer tool. Firefox also supports the Firebug extension and has excellent built-in developer tools, but as they behave more or less the same as Google Chrome's **Developer Tools** (**DevTools**), we will discuss common debugging approaches that work in both of these tools.

Before we talk about the specific debugging techniques, let's understand the type of errors that we would be interested in while we try to debug our code.

Syntax errors

When your code has something that does not confirm to the JavaScript language grammar, the interpreter rejects this piece of code. These are easy to catch if your IDE is helping you with syntax checking. Most modern IDEs help with these errors. Earlier, we discussed the usefulness of the tools such as **JSLint** and **JSHint** around catching syntax issues with your code. They analyze the code and flag errors in the syntax. JSHint output can be very illuminating. For example, the following output shows up so many things that we can change in the code. This snippet is from one of my existing projects:

```
temp git:(dev_branch)   jshint test.js
test.js: line 1, col 1, Use the function form of "use strict".
test.js: line 4, col 1, 'destructuring expression' is available in
  ES6 (use esnext option) or Mozilla JS extensions (use moz).
test.js: line 44, col 70, 'arrow function syntax (=>)' is only
  available in ES6 (use esnext option).
test.js: line 61, col 33, 'arrow function syntax (=>)' is only
  available in ES6 (use esnext option).
test.js: line 200, col 29, Expected ')' to match '(' from line 200
  and instead saw ':'.
test.js: line 200, col 29, 'function closure expressions' is only
  available in Mozilla JavaScript extensions (use moz option).
test.js: line 200, col 37, Expected '}' to match '{' from line 36
  and instead saw ')'.
```

```
test.js: line 200, col 39, Expected ')' and instead saw '{'.
test.js: line 200, col 40, Missing semicolon.
```

Using strict

We briefly discussed the **strict** mode in earlier chapters. The strict mode in JavaScript flags or eliminates some of the JavaScript silent errors. Rather than silently failing, the strict mode makes these failures throw errors instead. The strict mode also helps in converting mistakes to actual errors. There are two ways of enforcing the strict mode. If you want the strict mode for the entire script, you can just add the `use strict` statement as the first line of your JavaScript program. If you want a specific function to conform with the strict mode, you can add the directive as the first line of a function:

```
function strictFn(){
// This line makes EVERYTHING under this strict mode
'use strict';
…
function nestedStrictFn() {
//Everything in this function is also nested
…
}
}
```

Runtime exceptions

These errors appear when you execute the code and try to refer to an undefined variable or process a null. When a runtime exception occurs, any code after that particular line (which caused the exception) does not get executed. It is essential to handle such exceptional scenarios correctly in the code. While exception handling can help prevent crashes, they also aid in debugging. You can wrap the code that *may* encounter a runtime exception in a `try{ }` block. When any code in this block generates a runtime exception, a corresponding handler captures it. The handler is defined by a `catch(exception){}` block. Let's clarify this using an example:

```
try {
  var a = doesnotexist; // throws a runtime exception
} catch(e) {
  console.log(e.message);  //handle the exception
  //prints - "doesnotexist is not defined"
}
```

In this example, the `var a = doesnotexist;` line tries to assign an undefined variable, `doesnotexist`, to another variable, `a`. This causes a runtime exception. When we wrap this problematic code in the `try{} catch(){}` block and when the exception occurs (or is thrown), the execution stops in the `try{}` block and goes directly to the `catch() {}` handler. The `catch` handler is responsible for handling the exceptional scenario. In this case, we are displaying the error message on the console for debugging purposes. You can explicitly throw an exception to trigger an unhandled scenario in the code. Consider the following example:

```
function engageGear(gear){
  if(gear==="R"){ console.log ("Reversing");}
  if(gear==="D"){ console.log ("Driving");}
  if(gear==="N"){ console.log ("Neutral/Parking");}
  throw new Error("Invalid Gear State");
}
try
{
  engageGear("R");  //Reversing
  engageGear("P");  //Invalid Gear State
}
catch(e){
  console.log(e.message);
}
```

In this example, we are handling valid states of a gear shift (`R`, `N`, and `D`), but when we receive an invalid state, we are explicitly throwing an exception clearly stating the reason. When we call the function that we think may throw an exception, we wrap the code in the `try{}` block and attach a `catch(){}` handler with it. When the exception is caught by the `catch()` block, we handle the exceptional condition appropriately.

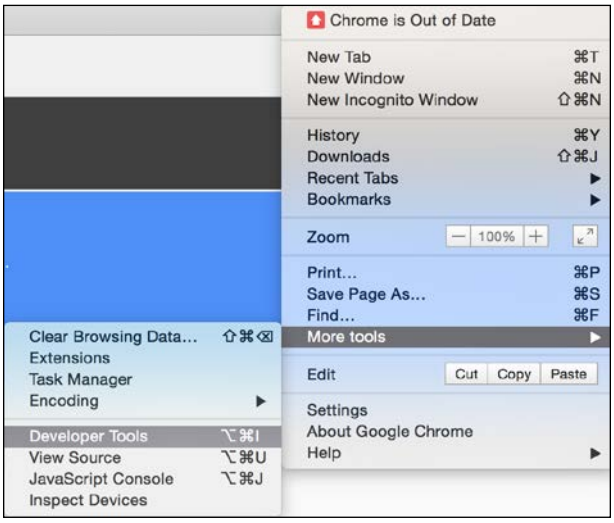
Console.log and asserts

Displaying the state of execution on the console can be very useful while debugging. However, modern developer tools allow you to put breakpoints and halt execution to inspect a particular value during runtime. You can quickly detect small issues by logging some variable state on the console.

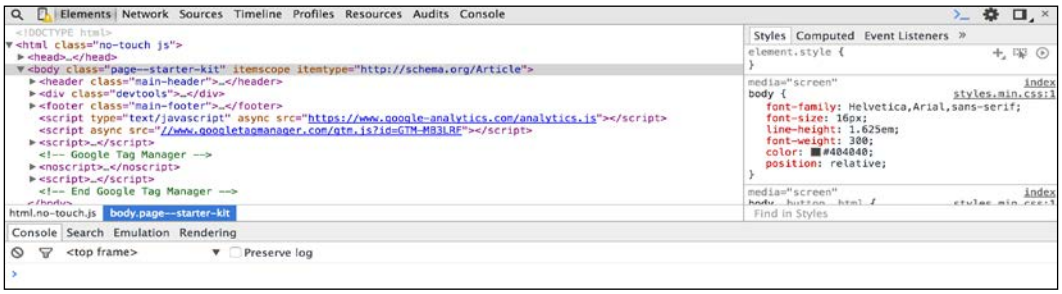
With these concepts, let's see how we can use Chrome Developer Tools to debug JavaScript code.

Chrome DevTools

You can start Chrome DevTools by navigating to menu | **More tools** | **Developer Tools**:



Chrome DevTools opens up on the lower pane of your browser and has a bunch of very useful sections:



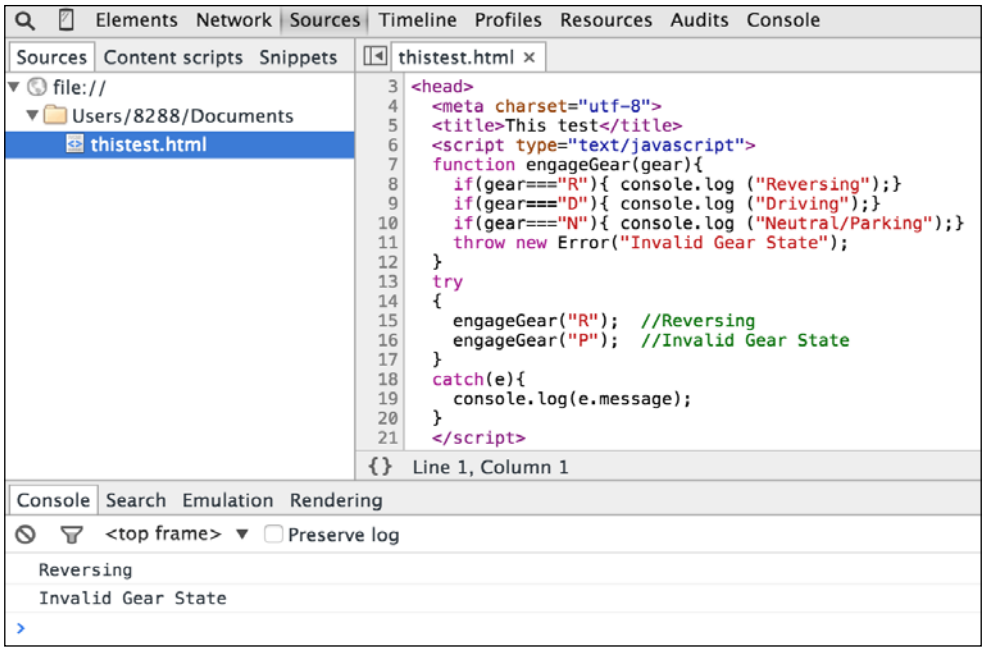
The **Elements** panel helps you inspect and monitor the DOM tree and associated style sheet for each of these components.

The **Network** panel is useful to understand network activity. For example, you can monitor the resources being downloaded over the network in real time.

The most important pane for us is the **Sources** pane. This pane is where the JavaScript source and debugger are displayed. Let's create a sample HTML with the following content:

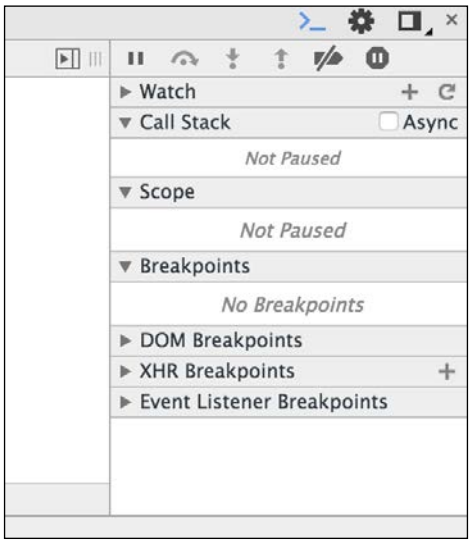
```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>This test</title>
  <script type="text/javascript">
  function engageGear(gear){
    if(gear==="R"){ console.log ("Reversing");}
    if(gear==="D"){ console.log ("Driving");}
    if(gear==="N"){ console.log ("Neutral/Parking");}
    throw new Error("Invalid Gear State");
  }
  try
  {
    engageGear("R");  //Reversing
    engageGear("P");  //Invalid Gear State
  }
  catch(e){
    console.log(e.message);
  }
  </script>
</head>
<body>
</body>
</html>
```

Save this HTML file and open it in Google Chrome. Open DevTools in the browser and you will see the following screen:



This is the view of the **Sources** panel. You can see the HTML and embedded JavaScript source in this panel. You can see the **Console** window as well. You can see that the file is executed and output is displayed in the **Console**.

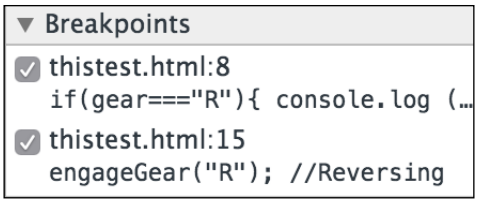
On the right-hand side, you will see the debugger window:



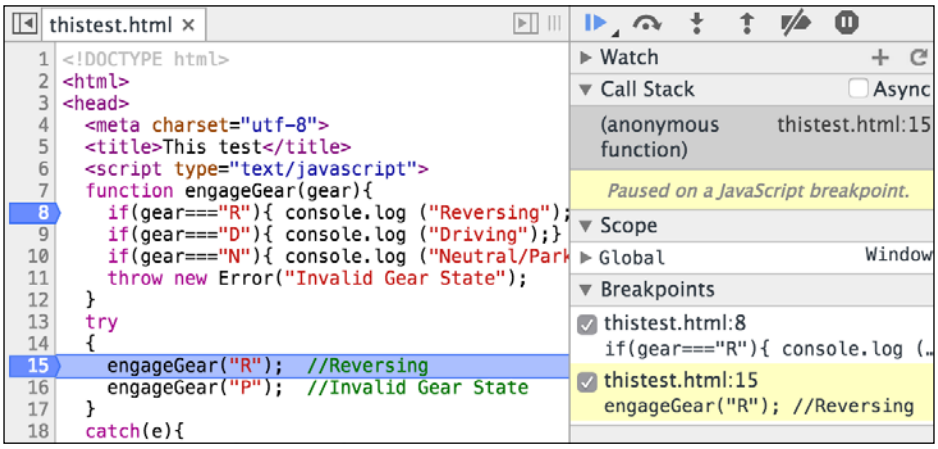
In the **Sources** panel, click on the line numbers **8** and **15** to add a breakpoint. The breakpoints allow you to stop the execution of the script at the specified point:

```
4   <meta charset="utf-8">
5   <title>This test</title>
6   <script type="text/javascript">
7   function engageGear(gear){
8     if(gear==="R"){ console.log ("Reversi
9     if(gear==="D"){ console.log ("Driving
10    if(gear==="N"){ console.log ("Neutral
11    throw new Error("Invalid Gear State")
12  }
13  try
14  {
15    engageGear("R");  //Reversing
16    engageGear("P");  //Invalid Gear Stat
17  }
```

In the debugging pane, you can see all the existing breakpoints:



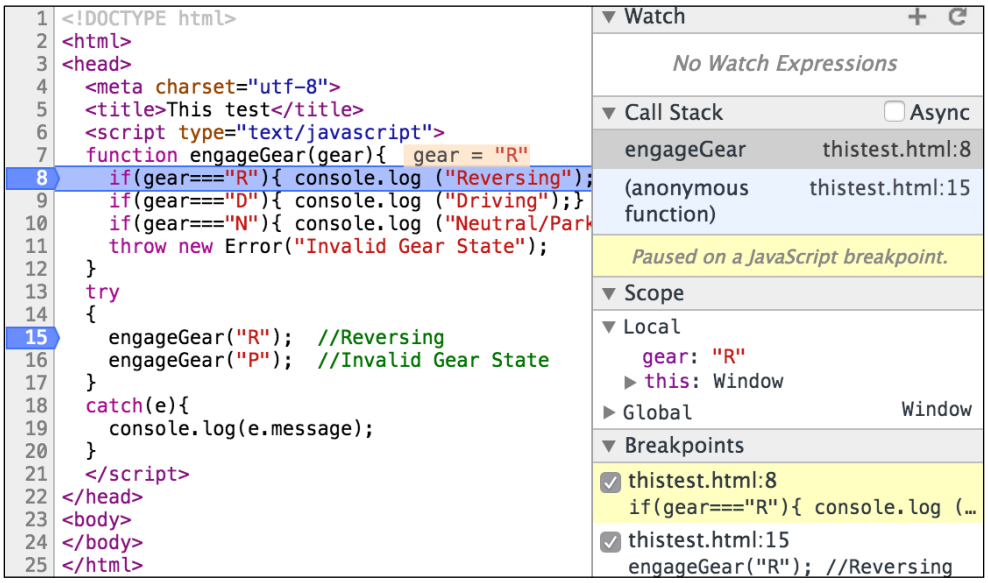
Now, when you rerun the same page, you will see that the execution stops at the debug point. One very useful technique is to inject code during the debugging phase. While the debugger is running, you can add code in order to help you understand the state of the code better:



This window now has all the action. You can see that the execution is paused on line **15**. In the debug window, you can see which breakpoint is being triggered. You can see the **Call Stack** also. You can resume execution in several ways. The debug command window has a bunch of actions:



You can resume execution (which will execute until the next breakpoint) by clicking on the  button. When you do this, the execution continues until the next breakpoint is encountered. In our case, we halt at line **8**:



You can observe that the **Call Stack** window shows you how we arrived at line **8**. The **Scope** panel shows the **Local** scope where you can see the variables in the scope when the breakpoint was arrived at. You can also step into or step over the next function.

There are other very useful mechanisms to debug and profile your code using Chrome DevTools. I would suggest you to go experiment with the tool and make it a part of your regular development flow.

Summary

Both the testing and debugging phases are essential to developing robust JavaScript code. TDD and BDD are approaches closely associated with the agile methodology and are widely embraced by the JavaScript developer community. In this chapter, we reviewed the best practices around TDD and usage of Jasmine as the testing framework. We saw various methods of debugging JavaScript using Chrome DevTools. In the next chapter, we will explore the new and exciting world of ES6, DOM manipulation, and cross-browser strategies.

6
ECMAScript 6

So far, we have taken a detailed tour of the JavaScript programming language. I am sure that you must have gained significant insight into the core of the language. What we saw so far was as per the **ECMAScript 5** (**ES5**) standards. **ECMAScript 6** (**ES6**) or **ECMAScript 2015** (**ES2015**) is the latest version of the ECMAScript standard. This standard is evolving and the last round of modifications was done in June, 2015. ES2015 is significant in its scope and the recommendations of ES2015 are being implemented in most JavaScript engines. This is great news. ES6 introduces a huge number of features that add syntactic forms and helpers that enrich the language significantly. The pace at which ECMAScript standards keep evolving makes it a bit difficult for browsers and JavaScript engines to support new features. It is also a practical reality that most programmers have to write code that can be supported by older browsers. The notorious Internet Explorer 6 was once the most widely used browser in the world. To make sure that your code is compatible with the most number of browsers is a daunting task. So, while you want to jump to the next set of awesome ES6 features, you will have to consider the fact that several ES6 features may not be supported by the most popular of browsers or JavaScript frameworks.

This may look like a dire scenario, but things are not that dark. **Node.js** uses the latest version of the V8 engine that supports majority of ES6 features. Facebook's **React** also supports them. Mozilla Firefox and Google Chrome are two of the most used browsers today and they support a majority of ES6 features.

To avoid such pitfalls and unpredictability, certain solutions have been proposed. The most useful among these are polyfills/shims and transpilers.

Shims or polyfills

Polyfills (also known as shims) are patterns to define behavior from a new version in a compatible form supported by an older version of the environment. There's a great collection of ES6 shims called **ES6 shim** (`https://github.com/paulmillr/es6-shim/`); I would highly recommend a study of these shims. From the ES6 shim collection, consider the following example of a shim.

The `Number.isFinite()` method of the ECMAScript 2015 (ES6) standard determines whether the passed value is a finite number. The equivalent shim for it would look something as follows:

```
var numberIsFinite = Number.isFinite || function isFinite(value) {
  return typeof value === 'number' && globalIsFinite(value);
};
```

The shim first checks if the `Number.isFinite()` method is available; if not, it *fills* it up with an implementation. This is a pretty nifty technique to fill in gaps in specifications. Shims are constantly upgraded with newer features and, hence, it is a sound strategy to keep the most updated shims in your project.



The `endsWith()` polyfill is described in detail at `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/endsWith`. `String.endsWith()` is part of ES6 but can be polyfilled easily for pre-ES6 environments.

Shims, however, cannot polyfill syntactical changes. For this, we can consider transpilers as an option.

Transpilers

Transpiling is a technique that combines both compilation and transformation. The idea is to write ES6-compatible code and use a tool that transpiles this code into a valid and equivalent ES5 code. We will be looking at the most complete and popular transpiler for ES6 called **Babel** (`https://babeljs.io/`).

Babel can be used in various ways. You can install it as a node module and invoke it from the command line or import it as a script in your web page. Babel's setup is exhaustive and well-documented at `https://babeljs.io/docs/setup/`. Babel also has a great **Read-Eval-Print-Loop** (**REPL**). We will Babel REPL for most of the examples in this chapter. An in-depth understanding of various ways in which Babel can be used is out of the scope of this book. However, I would urge you to start using Babel as part of your development workflow.

We will cover the most important part of ES6 specifications in this chapter. You should explore all the features of ES6 if possible and make them part of your development workflow.

ES6 syntax changes

ES6 brings in significant syntactic changes to JavaScript. These changes need careful study and some getting used to. In this section, we will study some of the most important syntax changes and see how you can use Babel to start using these newer constructs in your code right away.

Block scoping

We discussed earlier that the variables in JavaScript are function-scoped. Variables created in a nested scope are available to the entire function. Several programming languages provide you with a default block scope where any variable declared within a block of code (usually delimited by `{}`) is scoped (available) only within this block. To achieve a similar block scope in JavaScript, a prevalent method is to use **immediately-invoked function expressions** (**IIFE**). Consider the following example:

```
var a = 1;
(function blockscope(){
    var a = 2;
    console.log(a);   // 2
})();
console.log(a);       // 1
```

Using the IIFE, we are creating a block scope for the `a` variable. When a variable is declared in the IIFE, its scope is restricted within the function. This is the traditional way of simulating the block scope. ES6 supports block scoping without using IIFEs. In ES6, you can enclose any statement(s) in a block defined by `{}`. Instead of using `var`, you can declare a variable using `let` to define the block scope. The preceding example can be rewritten using ES6 block scopes as follows:

```
"use strict";
var a = 1;
{
  let a = 2;
  console.log( a ); // 2
}
console.log( a ); // 1
```

Using standalone brackets {} may seem unusual in JavaScript, but this convention is fairly common to create a block scope in many languages. The block scope kicks in other constructs such as `if { }` or `for (){ }` as well.

When you use a block scope in this way, it is generally preferred to put the variable declaration on top of the block. One difference between variables declared using `var` and `let` is that variables declared with `var` are attached to the entire function scope, while variables declared using `let` are attached to the block scope and they are not initialized until they appear in the block. Hence, you cannot access a variable declared with `let` earlier than its declaration, whereas with variables declared using `var`, the ordering doesn't matter:

```
function fooey() {
  console.log(foo); // ReferenceError
  let foo = 5000;
}
```

One specific use of `let` is in for loops. When we use a variable declared using `var` in a for loop, it is created in the global or parent scope. We can create a block-scoped variable in the for loop scope by declaring a variable using `let`. Consider the following example:

```
for (let i = 0; i<5; i++) {
  console.log(i);
}
console.log(i); // i is not defined
```

As `i` is created using `let`, it is scoped in the `for` loop. You can see that the variable is not available outside the scope.

One more use of block scopes in ES6 is the ability to create constants. Using the `const` keyword, you can create constants in the block scope. Once the value is set, you cannot change the value of such a constant:

```
if(true){
  const a=1;
  console.log(a);
  a=100;  ///"a" is read-only, you will get a TypeError
}
```

A constant has to be initialized while being declared. The same block scope rules apply to functions also. When a function is declared inside a block, it is available only within that scope.

Default parameters

Defaulting is very common. You always set some default value to parameters passed to a function or variables that you initialize. You may have seen code similar to the following:

```
function sum(a,b){
  a = a || 0;
  b = b || 0;
  return (a+b);
}
console.log(sum(9,9)); //18
console.log(sum(9));   //9
```

Here, we are using `||` (the OR operator) to default variables `a` and `b` to `0` if no value was supplied when this function was invoked. With ES6, you have a standard way of defaulting function arguments. The preceding example can be rewritten as follows:

```
function sum(a=0, b=0){
  return (a+b);
}
console.log(sum(9,9)); //18
console.log(sum(9));   //9
```

You can pass any valid expression or function call as part of the default parameter list.

Spread and rest

ES6 has a new operator, ... Based on how it is used, it is called either `spread` or `rest`. Let's look at a trivial example:

```
function print(a, b){
  console.log(a,b);
}
print(...[1,2]);  //1,2
```

What's happening here is that when you add ... before an array (or an iterable) it *spreads* the element of the array in individual variables in the function parameters. The `a` and `b` function parameters were assigned two values from the array when it was spread out. Extra parameters are ignored while spreading an array:

```
print(...[1,2,3 ]);  //1,2
```

This would still print `1` and `2` because there are only two functional parameters available. Spreads can be used in other places also, such as array assignments:

```
var a = [1,2];
var b = [ 0, ...a, 3 ];
console.log( b ); //[0,1,2,3]
```

There is another use of the ... operator that is the very opposite of the one that we just saw. Instead of spreading the values, the same operator can gather them into one:

```
function print (a,...b){
  console.log(a,b);
}
console.log(print(1,2,3,4,5,6,7));  //1 [2,3,4,5,6,7]
```

In this case, the variable `b` takes the *rest* of the values. The `a` variable took the first value as `1` and `b` took the rest of the values as an array.

Destructuring

If you have worked on a functional language such as **Erlang**, you will relate to the concept of pattern matching. Destructuring in JavaScript is something very similar. Destructuring allows you to bind values to variables using pattern matching. Consider the following example:

```
var [start, end] = [0,5];
for (let i=start; i<end; i++){
  console.log(i);
}
//prints - 0,1,2,3,4
```

We are assigning two variables with the help of array destructuring:

```
var [start, end] = [0,5];
```

As shown in the preceding example, we want the pattern to match when the first value is assigned to the first variable (`start`) and the second value is assigned to the second variable (`end`). Consider the following snippet to see how the destructuring of array elements works:

```
function fn() {
  return [1,2,3];
}
var [a,b,c]=fn();
console.log(a,b,c); //1 2 3
```

```
//We can skip one of them
var [d,,f]=fn();
console.log(d,f);   //1 3
//Rest of the values are not used
var [e,] = fn();
console.log(e);     //1
```

Let's discuss how objects' destructuring works. Let's say that you have a function `f` that returns an object as follows:

```
function f() {
  return {
    a: 'a',
    b: 'b',
    c: 'c'
  };
}
```

When we destructure the object being returned by this function, we can use the similar syntax as we saw earlier; the difference is that we use `{}` instead of `[]`:

```
var { a: a, b: b, c: c } = f();
console.log(a,b,c); //a b c
```

Similar to arrays, we use pattern matching to assign variables to their corresponding values returned by the function. There is an even shorter way of writing this if you are using the same variable as the one being matched. The following example would do just fine:

```
var { a,b,c } = f();
```

However, you would mostly be using a different variable name from the one being returned by the function. It is important to remember that the syntax is *source: destination* and not the usual *destination: source*. Carefully observe the following example:

```
//this is target: source - which is incorrect
var { x: a, x: b, x: c } = f();
console.log(x,y,z); //x is undefined, y is undefined z is undefined
//this is source: target - correct
var { a: x, b: y, c: z } = f();
console.log(x,y,z); // a b c
```

This is the opposite of the *target* = *source* way of assigning values and hence will take some time in getting used to.

Object literals

Object literals are everywhere in JavaScript. You would think that there is no scope of improvement there. However, ES6 wants to improve this too. ES6 introduces several shortcuts to create a concise syntax around object literals:

```
var firstname = "Albert", lastname = "Einstein",
  person = {
    firstname: firstname,
    lastname: lastname
  };
```

If you intend to use the same property name as the variable that you are assigning, you can use the concise property notation of ES6:

```
var firstname = "Albert", lastname = "Einstein",
  person = {
    firstname,
    lastname
  };
```

Similarly, you are assigning functions to properties as follows:

```
var person = {
  getName: function(){
    // ..
  },
  getAge: function(){
    //..
  }
}
```

Instead of the preceding lines, you can say the following:

```
var person = {
  getName(){
    // ..
  },
  getAge(){
    //..
  }
}
```

Template literals

I am sure you have done things such as the following:

```
function SuperLogger(level, clazz, msg){
  console.log(level+": Exception happened in class:"+clazz+" -
    Exception :"+ msg);
}
```

This is a very common way of replacing variable values to form a string literal. ES6 provides you with a new type of string literal using the backtick (`) delimiter. You can use string interpolation to put placeholders in a template string literal. The placeholders will be parsed and evaluated.

The preceding example can be rewritten as follows:

```
function SuperLogger(level, clazz, msg){
  console.log(`${level} : Exception happened in class: ${clazz} -
    Exception : {$msg}`);
}
```

We are using `` around a string literal. Within this literal, any expression of the `${..}` form is parsed immediately. This parsing is called interpolation. While parsing, the variable's value replaces the placeholder within `${}`. The resulting string is just a normal string with the placeholders replaced with actual variable values.

With string interpolation, you can split a string into multiple lines also, as shown in the following code (very similar to Python):

```
var quote =
`Good night, good night!
Parting is such sweet sorrow,
that I shall say good night
till it be morrow.`;
console.log( quote );
```

You can use function calls or valid JavaScript expressions as part of the string interpolation:

```
function sum(a,b){
  console.log(`The sum seems to be ${a + b}`);
}
sum(1,2); //The sum seems to be 3
```

The final variation of the template strings is called **tagged template string**. The idea is to modify the template string using a function. Consider the following example:

```
function emmy(key, ...values){
  console.log(key);
  console.log(values);
}
let category="Best Movie";
let movie="Adventures in ES6";
emmy`And the award for ${category} goes to ${movie}`;

//["And the award for "," goes to ",""]
//["Best Movie","Adventures in ES6"]
```

The strangest part is when we call the `emmy` function with the template literal. It's not a traditional function call syntax. We are not writing `emmy()`; we are just *tagging* the literal with the function. When this function is called, the first argument is an array of all the plain strings (the string between interpolated expressions). The second argument is the array where all the interpolated expressions are evaluated and stored.

Now what this means is that the tag function can actually change the resulting template tag:

```
function priceFilter(s, ...v){
  //Bump up discount
  return s[0]+ (v[0] + 5);
}
let default_discount = 20;
let greeting = priceFilter `Your purchase has a discount of
  ${default_discount} percent`;
console.log(greeting);  //Your purchase has a discount of 25
```

As you can see, we modified the value of the discount in the tag function and returned the modified values.

Maps and Sets

ES6 introduces four new data structures: **Map**, **WeakMap**, **Set**, and **WeakSet**.
We discussed earlier that objects are the usual way of creating key-value pairs in JavaScript. The disadvantage of objects is that you cannot use non-string values as keys. The following snippets demonstrate how Maps are created in ES6:

```
let m = new Map();
let s = { 'seq' : 101 };

m.set('1','Albert');
m.set('MAX', 99);
m.set(s,'Einstein');

console.log(m.has('1')); //true
console.log(m.get(s));   //Einstein
console.log(m.size);     //3
m.delete(s);
m.clear();
```

You can initialize the map while declaring it:

```
let m = new Map([
  [ 1, 'Albert' ],
  [ 2, 'Douglas' ],
  [ 3, 'Clive' ],
]);
```

If you want to iterate over the entries in the Map, you can use the `entries()` function that will return you an iterator. You can iterate through all the keys using the `keys()` function and you can iterate through the values of the Map using the `values()` function:

```
let m2 = new Map([
    [ 1, 'Albert' ],
    [ 2, 'Douglas' ],
    [ 3, 'Clive' ],
]);
for (let a of m2.entries()){
  console.log(a);
}
//[1,"Albert"] [2,"Douglas"][3,"Clive"]
for (let a of m2.keys()){
  console.log(a);
} //1 2 3
for (let a of m2.values()){
  console.log(a);
}
//Albert Douglas Clive
```

A variation of JavaScript Maps is a WeakMap—a WeakMap does not prevent its keys from being garbage-collected. Keys for a WeakMap must be objects and the values can be arbitrary values. While a WeakMap behaves in the same way as a normal Map, you cannot iterate through it and you can't clear it. There are reasons behind these restrictions. As the state of the Map is not guaranteed to remain static (keys may get garbage-collected), you cannot ensure correct iteration.

There are not many cases where you may want to use WeakMap. Most uses of a Map can be written using normal Maps.

While Maps allow you to store arbitrary values, Sets are a collection of unique values. Sets have similar methods as Maps; however, `set()` is replaced with `add()`, and the `get()` method does not exist. The reason that the `get()` method is not there is because a Set has unique values, so you are interested in only checking whether the Set contains a value or not. Consider the following example:

```
let x = {'first': 'Albert'};
let s = new Set([1,2,'Sunday',x]);
//console.log(s.has(x));  //true
s.add(300);
//console.log(s);  //[1,2,"Sunday",{"first":"Albert"},300]

for (let a of s.entries()){
  console.log(a);
}
//[1,1]
//[2,2]
//["Sunday","Sunday"]
//[{"first":"Albert"},{"first":"Albert"}]
//[300,300]
for (let a of s.keys()){
  console.log(a);
}
//1
//2
//Sunday
//{"first":"Albert"}
//300
for (let a of s.values()){
  console.log(a);
}
//1
//2
//Sunday
//{"first":"Albert"}
//300
```

The `keys()` and `values()` iterators both return a list of the unique values in the Set. The `entries()` iterator yields a list of entry arrays, where both items of the array are the unique Set values. The default iterator for a Set is its `values()` iterator.

Symbols

ES6 introduces a new data type called Symbol. A Symbol is guaranteed to be unique and immutable. Symbols are usually used as an identifier for object properties. They can be considered as uniquely generated IDs. You can create Symbols with the `Symbol()` factory method—remember that this is not a constructor and hence you should not use a `new` operator:

```
let s = Symbol();
console.log(typeof s); //symbol
```

Unlike strings, Symbols are guaranteed to be unique and hence help in preventing name clashes. With Symbols, we have an extensibility mechanism that works for everyone. ES6 comes with a number of predefined built-in Symbols that expose various meta behaviors on JavaScript object values.

Iterators

Iterators have been around in other programming languages for quite some time. They give convenience methods to work with collections of data. ES6 introduces iterators for the same use case. ES6 iterators are objects with a specific interface. Iterators have a `next()` method that returns an object. The returning object has two properties—`value` (the next value) and `done` (indicates whether the last result has been reached). ES6 also defines an `Iterable` interface, which describes objects that must be able to produce iterators. Let's look at an array, which is an iterable, and the iterator that it can produce to consume its values:

```
var a = [1,2];
var i = a[Symbol.iterator]();
console.log(i.next());      // { value: 1, done: false }
console.log(i.next());      // { value: 2, done: false }
console.log(i.next());      // { value: undefined, done: true }
```

As you can see, we are accessing the array's iterator via `Symbol.iterator()` and calling the `next()` method on it to get each successive element. Both `value` and `done` are returned by the `next()` method call. When you call `next()` past the last element in the array, you get an undefined value and `done: true`, indicating that you have iterated over the entire array.

For..of loops

ES6 adds a new iteration mechanism in form of the `for..of` loop, which loops over the set of values produced by an iterator.

The value that we iterate over with `for..of` is an iterable.

Let's compare `for..of` to `for..in`:

```
var list = ['Sunday','Monday','Tuesday'];
for (let i in list){
  console.log(i);  //0 1 2
}
for (let i of list){
  console.log(i);  //Sunday Monday Tuesday
}
```

As you can see, using the `for..in` loop, you can iterate over indexes of the `list` array, while the `for..of` loop lets you iterate over the values stored in the `list` array.

Arrow functions

One of the most interesting new parts of ECMAScript 6 is arrow functions. Arrow functions are, as the name suggests, functions defined with a new syntax that uses an *arrow* (`=>`) as part of the syntax. Let's first see how arrow functions look:

```
//Traditional Function
function multiply(a,b) {
  return a*b;
}
//Arrow
var multiply = (a,b) => a*b;
console.log(multiply(1,2)); //2
```

The arrow function definition consists of a parameter list (of zero or more parameters and surrounding `( .. )` if there's not exactly one parameter), followed by the `=>` marker, which is followed by a function body.

The body of the function can be enclosed by { .. } if there's more than one expression in the body. If there's only one expression, and you omit the surrounding { .. }, there's an implied return in front of the expression. There are several variations of how you can write arrow functions. The following are the most commonly used:

```
// single argument, single statement
//arg => expression;
var f1 = x => console.log("Just X");
f1(); //Just X

// multiple arguments, single statement
//(arg1 [, arg2]) => expression;
var f2 = (x,y) => x*y;
console.log(f2(2,2)); //4

// single argument, multiple statements
// arg => {
//     statements;
// }
var f3 = x => {
  if(x>5){
    console.log(x);
  }
  else {
    console.log(x+5);
  }
}
f3(6); //6

// multiple arguments, multiple statements
// ([arg] [, arg]) => {
//   statements
// }
var f4 = (x,y) => {
  if(x!=0 && y!=0){
    return x*y;
  }
}
console.log(f4(2,2));//4
```

```
// with no arguments, single statement
//() => expression;
var f5 = () => 2*2;
console.log(f5()); //4

//IIFE
console.log(( x => x * 3 )( 3 )); // 9
```

It is important to remember that all the characteristics of a normal function parameter are available to arrow functions, including default values, destructuring, and rest parameters.

Arrow functions offer a convenient and short syntax, which gives your code a very *functional programming* flavor. Arrow functions are popular because they offer an attractive promise of writing concise functions by dropping function, return, and { .. } from the code. However, arrow functions are designed to fundamentally solve a particular and common pain point with this-aware coding. In normal ES5 functions, every new function defined its own value of `this` (a new object in case of a constructor, `undefined` in strict mode function calls, context object if the function is called as an *object method*, and so on). JavaScript functions always have their own `this` and this prevents you from accessing the `this` of, for example, a surrounding method from inside a callback. To understand this problem, consider the following example:

```
function CustomStr(str){
  this.str = str;
}
CustomStr.prototype.add = function(s){    // --> 1
  'use strict';
  return s.map(function (a){              // --> 2
    return this.str + a;                  // --> 3
  });
};

var customStr = new CustomStr("Hello");
console.log(customStr.add(["World"]));
//Cannot read property 'str' of undefined
```

On the line marked with `3`, we are trying to get `this.str`, but the anonymous function also has its own `this`, which shadows `this` from the method from line `1`. To fix this in ES5, we can assign `this` to a variable and use the variable instead:

```
function CustomStr(str){
  this.str = str;
}
```

```
CustomStr.prototype.add = function(s){
  'use strict';
  var that = this;                          // --> 1
  return s.map(function (a){                // --> 2
    return that.str + a;                    // --> 3
  });
};

var customStr = new CustomStr("Hello");
console.log(customStr.add(["World"]));
//["HelloWorld]
```

On the line marked with `1`, we are assigning `this` to a variable, `that`, and in the anonymous function we are using the `that` variable, which will have a reference to `this` from the correct context.

ES6 arrow functions have lexical `this`, meaning that the arrow functions capture the `this` value of the enclosing context. We can convert the preceding function to an equivalent arrow function as follows:

```
function CustomStr(str){
  this.str = str;
}
CustomStr.prototype.add = function(s){
  return s.map((a)=> {
    return this.str + a;
  });
};
var customStr = new CustomStr("Hello");
console.log(customStr.add(["World"]));
//["HelloWorld]
```

Summary

In this chapter, we discussed a few important features being added to the language in ES6. It's an exciting collection of new language features and paradigms and, using polyfills and transpilers, you can start with them right away. JavaScript is an ever growing language and it is important to understand what the future holds. ES6 features make JavaScript an even more interesting and mature language. In the next chapter, we will dive deep into manipulating the browser's **Document Object Model** (**DOM**) and events using JavaScript with jQuery.

7
DOM Manipulation and Events

The most important reason for JavaScript's existence is the web. JavaScript is the language for the web and the browser is the raison d'être for JavaScript. JavaScript gives dynamism to otherwise static web pages. In this chapter, we will dive deep into this relationship between the browser and language. We will understand the way in which JavaScript interacts with the components of the web page. We will look at the **Document Object Model** (**DOM**) and JavaScript event model.

DOM

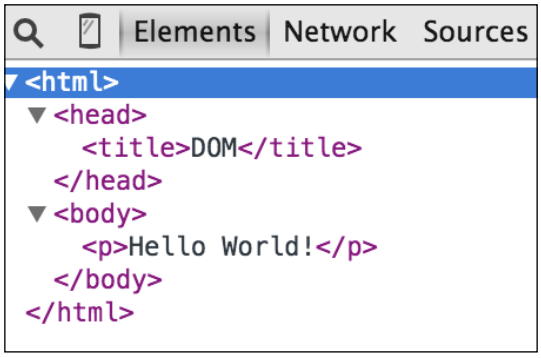
In this chapter, we will look at various aspects of JavaScript with regard to the browser and HTML. HTML, as I am sure you are aware, is the markup language used to define web pages. Various forms of markups exist for different uses. The popular marks are **Extensible Markup Language** (**XML**) and **Standard Generalized Markup Language** (**SGML**). Apart from these generic markup languages, there are very specific markup languages for specific purposes such as text processing and image meta information. **HyperText Markup Language** (**HTML**) is the standard markup language that defines the presentation semantics of a web page. A web page is essentially a document. The DOM provides you with a representation of this document. The DOM also provides you with a means of storing and manipulating this document. The DOM is the programming interface of HTML and allows structural manipulation using scripting languages such as JavaScript. The DOM provides a structural representation of the document. The structure consists of nodes and objects. Nodes have properties and methods on which you can operate in order to manipulate the nodes themselves. The DOM is just a representation and not a programming construct. DOM acts as a model for DOM processing languages such as JavaScript.

Accessing DOM elements

Most of the time, you will be interested in accessing DOM elements to inspect their values or processing these values for some business logic. We will take a detailed look at this particular use case. Let's create a sample HTML file with the following content:

```
<html>
<head>
  <title>DOM</title>
</head>
<body>
  <p>Hello World!</p>
</body>
</html>
```

You can save this file as `sample_dom.html`; when you open this in the Google Chrome browser, you will see the web page displayed with the **Hello World** text displayed. Now, open Google Chrome Developer Tools by navigating to options | **More Tools** | **Developer Tools** (this route may differ on your operating system and browser version). In the **Developer Tools** window, you will see the DOM structure:



Next, we will insert some JavaScript into this HTML page. We will invoke the JavaScript function when the web page is loaded. To do this, we will call a function on `window.onload`. You can place your script in the `<script>` tag located under the `<head>` tag. Your page should look as follows:

```
<html>
  <head>
    <title>DOM</title>
    <script>
```

```
        // run this function when the document is loaded
        window.onload = function() {
          var doc = document.documentElement;
          var body = doc.body;
          var _head = doc.firstChild;
          var _body = doc.lastChild;
          var _head_ = doc.childNodes[0];
          var title = _head.firstChild;
          alert(_head.parentNode === doc); //true
        }
      </script>
    </head>
    <body>
      <p>Hello World!</p>
    </body>
  </html>
```

The anonymous function is executed when the browser loads the page. In the function, we are getting the nodes of the DOM programmatically. The entire HTML document can be accessed using the `document.documentElement` function. We store the document in a variable. Once the document is accessed, we can traverse the nodes using several helper properties of the document. We are accessing the `<body>` element using `doc.body`. You can traverse through the children of an element using the `childNodes` array. The first and last children of a node can be accessed using additional properties—`firstChild` and `lastChild`.



It is not recommended to use render-blocking JavaScript in the `<head>` tag. This slows down the page render dramatically. Modern browsers support the `async` and `defer` attributes to indicate to the browsers that the rendering can go on while the script is being downloaded. You can use these tags in the `<head>` tag without worrying about performance degradation. You can get more information at `http://stackoverflow.com/questions/436411/where-is-the-best-place-to-put-script-tags-in-html-markup`.

Accessing specific nodes

The core DOM defines the `getElementsByTagName()` method to return `NodeList` of all the element objects whose `tagName` property is equal to a specific value. The following line of code returns a list of all the `<p/>` elements in a document:

```
var paragraphs = document.getElementsByTagName('p');
```

The HTML DOM defines `getElementsByName()` to retrieve all the elements that have their name attribute set to a specific value. Consider the following snippet:

```
<html>
  <head>
    <title>DOM</title>
    <script>
      showFeelings = function() {
        var feelings = document.getElementsByName("feeling");
        alert(feelings[0].getAttribute("value"));
        alert(feelings[1].getAttribute("value"));
      }
    </script>
  </head>
  <body>
    <p>Hello World!</p>
    <form method="post" action="/post">
      <fieldset>
        <p>How are you feeling today?</p>
        <input type="radio" name="feeling" value="Happy" />
          Happy<br />
        <input type="radio" name="feeling" value="Sad" />Sad<br />
      </fieldset>
      <input type="button" value="Submit"
        onClick="showFeelings()"/>
    </form>
  </body>
</html>
```

In this example, we are creating a group of radio buttons with the `name` attribute defined as `feeling`. In the `showFeelings` function, we get all the elements with the `name` attribute set to `feeling` and we iterate through all these elements.

The other method defined by the HTML DOM is `getElementById()`. This is a very useful method in accessing a specific element. This method does the lookup based on the `id` associated with an element. The `id` attribute is unique for every element and, hence, this kind of lookup is very fast and should be preferred over `getElementsByName()`. -However, you should be aware that the browser does not guarantee the uniqueness of the `id` attribute. In the following example, we are accessing a specific element using the ID. Element IDs are unique as opposed to tags or name attributes:

```
<html>
  <head>
```

```
    <title>DOM</title>
    <script>
      window.onload= function() {
        var greeting = document.getElementById("greeting");
        alert(greeting.innerHTML); //shows "Hello World" alert
      }
    </script>
  </head>
  <body>
    <p id="greeting">Hello World!</p>
    <p id="identify">Earthlings</p>
  </body>
</html>
```

What we discussed so far was the basics of DOM traversal in JavaScript. When the DOM gets complex and you want sophisticated operations on the DOM, these traversal and access functions seem limiting. With this basic knowledge with us, it's time to get introduced to a fantastic library for DOM traversal (among other things) called jQuery.

jQuery is a lightweight library designed to make common browser operations easier. Common operations such as DOM traversal and manipulation, event handling, animation, and Ajax can be tedious if done using pure JavaScript. jQuery provides you with easy-to-use and shorter helper mechanisms to help you develop these common operations very easily and quickly. jQuery is a feature-rich library, but as far as this chapter goes, we will focus primarily on DOM manipulation and events.

You can add jQuery to your HTML by adding the script directly from a **content delivery network** (**CDN**) or manually downloading the file and adding it to the script tag. The following example shows you how to download jQuery from Google's CDN:

```
<html>
  <head>
    <script src="https://ajax.googleapis.com/ajax/libs/
      jquery/2.1.4/jquery.min.js"></script>
  </head>
  <body>
  </body>
</html>
```

The advantage of a CDN download is that Google's CDN automatically finds the nearest download server for you and keeps an updated stable copy of the jQuery library. If you wish to download and manually host jQuery along with your website, you can add the script as follows:

```
<script src="./lib/jquery.js"></script>
```

In this example, the jQuery library is manually downloaded in the `lib` directory. With the jQuery setup in the HTML page, let's explore the methods of manipulating the DOM elements. Consider the following example:

```
<html>
  <head>
    <script src="https://ajax.googleapis.com/ajax/libs/
      jquery/2.1.4/jquery.min.js"></script>
    <script>
    $(document).ready(function() {
        $('#greeting').html('Hello World Martian');
    });
  </script>
  </head>
  <body>
    <p id="greeting">Hello World Earthling ! </p>
  </body>
</html>
```

After adding jQuery to the HTML page, we write the custom JavaScript that selects the element with a `greeting` ID and changes its value. The strange-looking code within `$()` is the jQuery in action. If you read the jQuery source code (and you should, it's brilliant) you will see the final line:

```
// Expose jQuery to the global object
window.jQuery = window.$ = jQuery;
```

The `$` is just a function. It is an alias for the function called jQuery. The `$` is a syntactic sugar that makes the code concise. In fact, you can use both `$` and `jQuery` interchangeably. For example, both `$('#greeting').html('Hello World Martian');` and `jQuery('#greeting').html('Hello World Martian');` are the same.

You can't use jQuery before the page is completely loaded. As jQuery will need to know all the nodes of the DOM structure, the entire DOM has to be in-memory. To ensure that the page is completely loaded and in a state where it's ready to be manipulated, we can use the `$(document).ready()` function. Here, the IIFE is executed only after the entire documented is *ready*:

```
$(document).ready(function() {
  $('#greeting').html('Hello World Martian');
});
```

This snippet shows you how we can associate a function to jQuery's `.ready()` function. This function will be executed once the document is ready. We are using `$(document)` to create a jQuery object from our page's document. We are calling the `.ready()` function on the jQuery object and passing it the function that we want to execute.

This is a very common thing to do when using jQuery—so much so that it has its own shortcut. You can replace the entire `ready()` call with a short `$()` call:

```
$(function() {
  $('#greeting').html('Hello World Martian');
});
```

The most important function in jQuery is `$()`. This function typically accepts a CSS selector as its sole parameter and returns a new jQuery object pointing to the corresponding elements on the page. The three primary selectors are the tag name, ID, and class. They can be used either on their own or in combination with others. The following simple examples illustrate how these three selectors appear in code:

Selector	CSS Selector	jQuery Selector	Output from the selector
Tag	`p{}`	`$('p')`	This selects all the `p` tags from the document.
Id	`#div_1`	`$('#div_1')`	This selects single elements that have a `div_1` ID. The symbol used to identify the ID is #.
Class	`.bold_fonts`	`$('.bold_fonts')`	This selects all the elements in the document that have the CSS class `bold_fonts`. The symbol used to identify the class match is `"."`.

jQuery works on CSS selectors.



As CSS selectors are not in the scope of this book, I would suggest that you go to `http://www.w3.org/TR/CSS2/selector.html` to get a fair idea of the concept.

We also assume that you are familiar with HTML tags and syntax. The following example covers the fundamental idea of how jQuery selectors work:

```
<html>
  <head>
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/2.1.4/
jquery.min.js"></script>
    <script>
      $(function() {
        $('h1').html(function(index, oldHTML){
          return oldHTML + "Finally?";
        });
        $('h1').addClass('highlight-blue');
        $('#header > h1 ').css('background-color', 'cyan');
        $('ul li:not(.highlight-blue)').addClass(
          'highlight-green');
        $('tr:nth-child(odd)').addClass('zebra');
      });
    </script>
    <style>
      .highlight-blue {
        color: blue;
      }
      .highlight-green{
        color: green;
      }
      .zebra{
        background-color: #666666;
        color: white;
      }
    </style>
  </head>
  <body>
```

```
    <div id=header>
      <h1>Are we there yet ? </h1>
      <span class="highlight">
        <p>Journey to Mars</p>
        <ul>
          <li>First</li>
          <li>Second</li>
          <li class="highlight-blue">Third</li>
        </ul>
      </span>
      <table>
        <tr><th>Id</th><th>First name</th><th>Last Name</th></tr>
        <tr><td>1</td><td>Albert</td><td>Einstein</td></tr>
        <tr><td>2</td><td>Issac</td><td>Newton</td></tr>
        <tr><td>3</td><td>Enrico</td><td>Fermi</td></tr>
        <tr><td>4</td><td>Richard</td><td>Feynman</td></tr>
      </table>
    </div>
  </body>
</html>
```

In this example, we are selecting several DOM elements in the HTML page using selectors. We have an H1 header with the text, `Are we there yet ?`; when the page loads, our jQuery script accesses all H1 headers and appends the text `Finally?` to them:

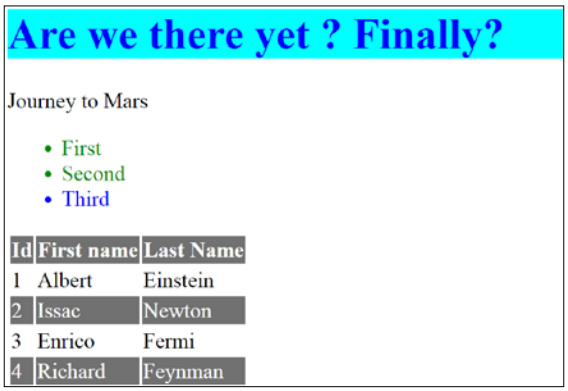
```
$('h1').html(function(index, oldHTML){
  return oldHTML + "Finally ?";
});
```

The `$.html()` function sets the HTML for the target element—an H1 header in this case. Additionally, we select all H1 headers and apply a specific CSS style class, `highlight-blue`, to all of them. The `$('h1').addClass('highlight-blue')` statement selects all the H1 headers and uses the `$.addClass(<CSS class>)` method to apply a CSS class to all the elements selected using the selector.

We use the child combinator (`>`) to custom CSS styles using the `$.css()` function. In effect, the selector in the `$()` function is saying, "Find each header (`h1`) that is a child (`>`) of the element with an ID of header (`#header`)." For each such element, we apply a custom CSS. The next usage is interesting. Consider the following line:

```
$('ul li:not(.highlight-blue)').addClass('highlight-green');
```

We are selecting "For all list elements (`li`) that do not have the class `highlight-blue` applied to them, apply CSS class `highlight-green`. The final line—`$('tr:nth-child(odd)').addClass('zebra')`—can be interpreted as: From all table rows (`tr`), for every odd row, apply CSS style `zebra`. The *n*th-child selector is a custom selector provided by jQuery. The final output looks something similar to the following (Though it shows several jQuery selector types, it is very clear that knowledge of jQuery is not a substitute for bad design taste.):



Once you have made a selection, there are two broad categories of methods that you can call on the selected element. These methods are **getters** and **setters**. Getters retrieve a piece of information from the selection, and setters alter the selection in some way.

Getters usually operate only on the first element in a selection while setters operate on all the elements in a selection. Setters use implicit iteration to automatically iterate over all the elements in the selection.

For example, we want to apply a CSS class to all list items on the page. When we call the `addClass` method on the selector, it is automatically applied to all elements of this particular selection. This is implicit iteration in action:

```
$( 'li' ).addClass( highlighted' );
```

However, sometimes you just don't want to go through all the elements via implicit iteration. You may want to selectively modify only a few of the elements. You can explicitly iterate over the elements using the `.each()` method. In the following code, we are processing elements selectively and using the `index` property of the element:

```
$( 'li' ).each(function( index, element ) {
  if(index % 2 == 0)
    $(elem).prepend( '<b>' + STATUS + '</b>' );
});
```

Chaining

Chaining jQuery methods allows you to call a series of methods on a selection without temporarily storing the intermediate values. This is possible because every setter method that we call returns the selection on which it was called. This is a very powerful feature and you will see it being used by many professional libraries. Consider the following example:

```
$( '#button_submit' )
  .click(function() {
    $( this ).addClass( 'submit_clicked' );
  })
  .find( '#notification' )
    .attr( 'title', 'Message Sent' );x
```

In this snippet, we are chaining `click()`, `find()`, and `attr()` methods on a selector. Here, the `click()` method is executed, and once the execution finishes, the `find()` method locates the element with the `notification` ID and changes its `title` attribute to a string.

Traversal and manipulation

We discussed various methods of element selection using jQuery. We will discuss several DOM traversal and manipulation methods using jQuery in this section. These tasks would be rather tedious to achieve using native DOM manipulation. jQuery makes them intuitive and elegant.

Before we delve into these methods, let's familiarize ourselves with a bit of HTML terminology that we will be using from now on. Consider the following HTML:

```
<ul> <-This is the parent of both 'li' and ancestor of everything
  in
  <li> <-The first (li) is a child of the (ul)
    <span>  <-this is the descendent of the 'ul'
      <i>Hello</i>
    </span>
  </li>
  <li>World</li> <-both 'li' are siblings
</ul>
```

Using jQuery traversal methods, we select the first element and traverse through the DOM in relation to this element. As we traverse the DOM, we alter the original selection and we are either replacing the original selection with the new one or we are modifying the original selection.

For example, you can filter an existing selection to include only elements that match a certain criterion. Consider this example:

```
var list = $( 'li' ); //select all list elements
// filter items that has a class 'highlight' associated
var highlighted = list.filter( '.highlight );
// filter items that doesn't have class 'highlight' associated
var not_highlighted = list.not( '.highlight );
```

jQuery allows you to add and remove classes to elements. If you want to toggle class values for elements, you can use the `toggleClass()` method:

```
$( '#usename' ).addClass( 'hidden' );
$( '#usename' ).removeClass( 'hidden' );
$( '#usename' ).toggleClass( 'hidden' );
```

Most often, you may want to alter the value of elements. You can use the `val()` method to alter the form of element values. For example, the following line alters the value of all the `text` type inputs in the form:

```
$( 'input[type="text"]' ).val( 'Enter usename:' );
```

To modify element attributes, you can use the `attr()` method as follows:

```
$('a').attr( 'title', 'Click' );
```

jQuery has an incredible depth of functionality when it comes to DOM manipulation—the scope of this book restricts a detailed discussion of all the possibilities.

Working with browser events

When are you developing for browsers, you will have to deal with user interactions and events associated to them, for example, text typed in the textbox, scrolling of the page, mouse button press, and others. When the user does something on the page, an event takes place. Some events are not triggered by user interaction, for example, `load` event does not require a user input.

When you are dealing with mouse or keyboard events in the browser, you can't predict when and in which order these events will occur. You will have to constantly look for a key press or mouse move to happen. It's like running an endless background loop listening to some key or mouse event to happen. In traditional programming, this was known as polling. There were many variations of these where the waiting thread used to be optimized using queues; however, polling is still not a great idea in general.

Browsers provide a much better alternative to polling. Browsers provide you with programmatic means to react when an event occurs. These hooks are generally called listeners. You can register a listener that reacts to a particular event and executes an associated callback function when the event is triggered. Consider this example:

```
<script>
  addEventListener("click", function() {
    ...
  });
</script>
```

The `addEventListener` function registers its second argument as a callback function. This callback is executed when the event specified in the first argument is triggered.

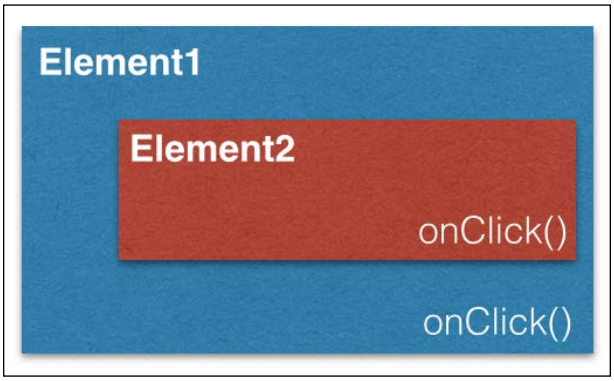
What we saw just now was a generic listener for the `click` event. Similarly, every DOM element has its own `addEventListener` method, which allows you to listen specifically on this element:

```
<button>Submit</button>
<p>No handler here.</p>
<script>
  var button = document.getElementById("#Bigbutton");
  button.addEventListener("click", function() {
    console.log("Button clicked.");
  });
</script>
```

In this example, we are using the reference to a specific element—a button with a `Bigbutton` ID—by calling `getElementById()`. On the reference of the button element, we are calling `addEventListener()` to assign a handler function for the click event. This is perfectly legitimate code that works fine in modern browsers such as Mozilla Firefox or Google Chrome. On Internet Explorer prior to IE9, however, this is not a valid code. This is because Microsoft implements its own custom `attachEvent()` method as opposed to the W3C standard `addEventListener()` prior to Internet Explorer 9. This is very unfortunate because you will have to write very bad hacks to handle browser-specific quirks.

Propagation

At this point, we should ask an important question—if an element and one of its ancestors have a handler on the same event, which handler will be fired first? Consider the following figure:



For example, we have **Element2** as a child of **Element1** and both have the `onClick` handler. When a user clicks on Element2, `onClick` on both Element2 and Element1 is triggered but the question is which one is triggered first. What should the event order be? Well, the answer, unfortunately, is that it depends entirely on the browser. When browsers first arrived, two opinions emerged, naturally, from Netscape and Microsoft.

Netscape decided that the first event triggered should be Element1's `onClick`. This event ordering is known as event capturing.

Microsoft decided that the first event triggered should be Element2's `onClick`. This event ordering is known as event bubbling.

These are two completely opposite views and implementations of how browsers handled events. To end this madness, **World Wide Web Consortium** (**W3C**) decided a wise middle path. In this model, an event is first captured until it reaches the target element and then bubbles up again. In this standard behavior, you can choose in which phase you want to register your event handler—either in the capturing or bubbling phase. If the last argument is true in `addEventListener()`, the event handler is set for the capturing phase, if it is false, the event handler is set for the bubbling phase.

There are times when you don't want the event to be raised by the parents if it was already raised by the child. You can call the `stopPropagation()` method on the event object to prevent handlers further up from receiving the event. Several events have a default action associated with them. For example, if you click on a URL link, you will be taken to the link's target. The JavaScript event handlers are called before the default behavior is performed. You can call the `preventDefault()` method on the event object to stop the default behavior from being triggered.

These are event basics when you are using plain JavaScript on a browser. There is a problem here. Browsers are notorious when it comes to defining event-handling behavior. We will look at jQuery's event handling. To make things easier to manage, jQuery always registers event handlers for the bubbling phase of the model. This means that the most specific elements will get the first opportunity to respond to any event.

jQuery event handling and propagation

jQuery event handling takes care of many of these browser quirks. You can focus on writing code that runs on most supported browsers. jQuery's support for browser events is simple and intuitive. For example, this code listens for a user to click on any button element on the page:

```
$('button').click(function(event) {
  console.log('Mouse button clicked');
});
```

Just like the `click()` method, there are several other helper methods to cover almost all kinds of browser event. The following helpers exist:

- `blur`
- `change`
- `click`
- `dblclick`

- `error`
- `focus`
- `keydown`
- `keypress`
- `keyup`
- `load`
- `mousedown`
- `mousemove`
- `mouseout`
- `mouseover`
- `mouseup`
- `resize`
- `scroll`
- `select`
- `submit`
- `unload`

Alternatively, you can use the `.on()` method. There are a few advantages of using the `on()` method as it gives you a lot more flexibility. The `on()` method allows you to bind a handler to multiple events. Using the `on()` method, you can work on custom events as well.

Event name is passed as the first parameter to the `on()` method just like the other methods that we saw:

```
$('button').on( 'click', function( event ) {
  console.log(' Mouse button clicked');
});
```

Once you've registered an event handler to an element, you can trigger this event as follows:

```
$('button').trigger( 'click' );
```

This event can also be triggered as follows:

```
$('button').click();
```

You can unbind an event using jQuery's `.off()` method. This will remove any event handlers that were bound to the specified event:

```
$('button').off( 'click' );
```

You can add more than one handler to an element:

```
$("#element")
.on("click", firstHandler)
.on("click", secondHandler);
```

When the event is fired, both the handlers will be invoked. If you want to remove only the first handler, you can use the `off()` method with the second parameter indicating the handler that you want to remove:

```
$("#element).off("click",firstHandler);
```

This is possible if you have the reference to the handler. If you are using anonymous functions as handlers, you can't get reference to them. In this case, you can use namespaced events. Consider the following example:

```
$("#element").on("click.firstclick",function() {
  console.log("first click");
});
```

Now that you have a namespaced event handler registered with the element, you can remove it as follows:

```
$("#element).off("click.firstclick");
```

A major advantage of using `.on()` is that you can bind to multiple events at once. The `.on()` method allows you to pass multiple events in a space-separated string. Consider the following example:

```
$('#inputBoxUserName').on('focus blur', function() {
  console.log( Handling Focus or blur event' );
});
```

You can add multiple event handlers for multiple events as follows:

```
$( "#heading" ).on({
  mouseenter: function() {
    console.log( "mouse entered on heading" );
  },
  mouseleave: function() {
    console.log( "mouse left heading" );
  },
  click: function() {
    console.log( "clicked on heading" );
  }
});
```

As of jQuery 1.7, all events are bound via the `on()` method, even if you call helper methods such as `click()`. Internally, jQuery maps these calls to the `on()` method. Due to this, it's generally recommended to use the `on()` method for consistency and faster execution.

Event delegation

Event delegation allows us to attach a single event listener to a parent element. This event will fire for all the descendants matching a selector even if these descendants will be created in the future (after the listener was bound to the element).

We discussed *event bubbling* earlier. Event delegation in jQuery works primarily due to event bubbling. Whenever an event occurs on a page, the event bubbles up from the element that it originated from, up to its parent, then up to the parent's parent, and so on, until it reaches the root element (`window`). Consider the following example:

```
<html>
  <body>
    <div id="container">
      <ul id="list">
        <li><a href="http://google.com">Google</a></li>
        <li><a href="http://myntra.com">Myntra</a></li>
        <li><a href="http://bing.com">Bing</a></li>
      </ul>
    </div>
  </body>
</html>
```

Now let's say that we want to perform some common action on any of the URL clicks. We can add an event handler to all the `a` elements in the list as follows:

```
$( "#list a" ).on( "click", function( event ) {
  console.log( $( this ).text() );
});
```

This works perfectly fine, but this code has a minor bug. What will happen if there is an additional URL added to the list as a result of some dynamic action? Let's say that we have an **Add** button that adds new URLs to this list. So, if the new list item is added with a new URL, the earlier event handler will not be attached to it. For example, if the following link is added to the list dynamically, clicking on it will not trigger the handler that we just added:

```
<li><a href="http://yahoo.com">Yahoo</a></li>
```

This is because such events are registered only when the `on()` method is called. In this case, as this new element did not exist when `.on()` was called, it does not get the event handler. With our understanding of event bubbling, we can visualize how the event will travel up the DOM tree. When any of the URLs are clicked on, the travel will be as follows:

```
a(click)->li->ul#list->div#container->body->html->root
```

We can create a delegated event as follows:

```
$( "#list" ).on( "click", "a", function( event ) {
  console.log( $( this ).text() );
});
```

We moved `a` from the original selector to the second parameter in the `on()` method. This second parameter of the `on()` method tells the handler to listen to this specific event and check whether the triggering element was the second parameter (the `a` in our case). As the second parameter matches, the handler function is executed. With this delegate event, we are attaching a single handler to the entire `ul#list`. This handler will listen to the click event triggered by any descendent of the `ul` element.

The event object

So far, we attached anonymous functions as event handlers. To make our event handlers more generic and useful, we can create named functions and assign them to the events. Consider the following lines:

```
function handlesClicks(event){
  //Handle click event
}
$("#bigButton").on('click', handlesClicks);
```

Here, we are passing a named function instead of an anonymous function to the `on()` method. Let's shift our focus now to the `event` parameter that we pass to the function. jQuery passes an event object with all the event callbacks. An event object contains very useful information about the event being triggered. In cases where we don't want the default behavior of the element to kick in, we can use the `preventDefault()` method of the event object. For example, we want to fire an AJAX request instead of a complete form submission or we want to prevent the default location to be opened when a URL anchor is clicked on. In these cases, you may also want to prevent the event from bubbling up the DOM. You can stop the event propagation by calling the `stopPropagation()` method of the event object. Consider this example:

```
$( "#loginform" ).on( "submit", function( event ) {
  // Prevent the form's default submission.
  event.preventDefault();
  // Prevent event from bubbling up DOM tree, also stops any
    delegation
  event.stopPropagation();
});
```

Apart from the event object, you also get a reference to the DOM object on which the event was fired. This element can be referred by `$(this)`. Consider the following example:

```
$( "a" ).click(function( event ) {
  var anchor = $( this );
  if ( anchor.attr( "href" ).match( "google" ) ) {
    event.preventDefault();
  }
});
```

Summary

This chapter was all about understanding JavaScript in its most important role—that of browser language. JavaScript plays the role of introducing dynamism on the web by facilitating DOM manipulation and event management on the browser. We discussed both of these concepts with and without jQuery. As the demands of the modern web are increasing, using libraries such as jQuery is essential. These libraries significantly improve the code quality and efficiency and, at the same time, give you the freedom to focus on important things.

We will focus on another incarnation of JavaScript—mainly on the server side. Node.js has become a popular JavaScript framework to write scalable server-side applications. We will take a detailed look at how we can best utilize Node.js for server applications.

8
Server-Side JavaScript

We have been focusing so far on the versatility of JavaScript as the language of the browser. It speaks volumes about the brilliance of the language given that JavaScript has gained significant popularity as a language to program scalable server systems. In this chapter, we will look at Node.js. Node.js is one of the most popular JavaScript frameworks used for server-side programming. Node.js is also one of the most watched project on GitHub and has superb community support.

Node uses V8, the virtual machine that powers Google Chrome, for server-side programming. V8 gives a huge performance benefit to Node because it directly compiles the JavaScript into native machine code over executing bytecode or using an interpreter as a middleware.

The versatility of V8 and JavaScript is a wonderful combination—the performance, reach, and overall popularity of JavaScript made Node an overnight success. In this chapter, we will cover the following topics:

- An asynchronous evented-model in a browser and Node.js
- Callbacks
- Timers
- EventEmitters
- Modules and npm

An asynchronous evented-model in a browser

Before we try to understand Node, let's try to understand JavaScript in a browser.

Node relies on event-driven and asynchronous platforms for server-side JavaScript. This is very similar to how browsers handle JavaScript. Both the browser and Node are event-driven and non-blocking when they use I/O.

To dive deeper into the event-driven and asynchronous nature of Node.js, let's first do a comparison of the various kinds of operations and costs associated with them:

L1 cache read	0.5 nanoseconds
L2 cache read	7 nanoseconds
RAM	100 nanoseconds
Read 4 KB randomly from SSD	150,000 ns
Read 1 MB sequentially from SSD	1,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns

These numbers are from `https://gist.github.com/jboner/2841832` and show how costly **Input/Output** (**I/O**) can get. The longest operations taken by a computer program are the I/O operations and these operations slow down the overall program execution if the program keeps waiting on these I/O operations to finish. Let's see an example of such an operation:

```
console.log("1");
var log = fileSystemReader.read("./verybigfile.txt");
console.log("2");
```

When you call `fileSystemReader.read()`, you are reading a file from the filesystem. As we just saw, I/O is the bottleneck here and can take quite a while before the read operation is completed. Depending on the kind of hardware, filesystem, OS, and so on, this operation will block the overall program execution quite a bit. The preceding code does some I/O that will be a blocking operation—the process will be blocked till I/O finishes and the data comes back. This is the traditional I/O model and most of us are familiar with this. However, this is costly and can cause terribly latency. Every process has associated memory and state—both these will be blocked till I/O is complete.

If a program blocks I/O, the Node server will refuse new requests. There are several ways of solving this problem. The most popular traditional approach is to use several threads to process requests—this technique is known as multithreading. If are you familiar with languages such as Java, chances are that you have written multithreaded code. Several languages support threads in various forms—a thread essentially holds its own memory and state. Writing multithreaded applications on a large scale is tough. When multiple threads are accessing a common shared memory or values, maintaining the correct state across these threads is a very difficult task. Threads are also costly when it comes to memory and CPU utilization. Threads that are used on synchronized resources may eventually get blocked.

The browser handles this differently. I/O in the browser happens outside the main execution thread and an event is emitted when I/O finishes. This event is handled by the callback function associated with that event. This type of I/O is non-blocking and asynchronous. As I/O is not blocking the main execution thread, the browser can continue to process other events as they come without waiting on any I/O. This is a powerful idea. Asynchronous I/O allows browsers to respond to several events and allows a high level of interactivity.

Node uses a similar idea for asynchronous processing. Node's event loop runs as a single thread. This means that the application that you write is essentially single-threaded. This does not mean that Node itself is single-threaded. Node uses **libuv** and is multithreaded—fortunately, these details are hidden within Node and you don't need to know them while developing your application.

Every call that involves an I/O call requires you to register a callback. Registering a callback is also asynchronous and returns immediately. As soon as an I/O operation is completed, its callback is pushed on the event loop. It is executed as soon as all the other callbacks that were pushed on the event loop before are executed. All operations are essentially thread-safe, primarily because there is no parallel execution path in the event loop that will require synchronization.

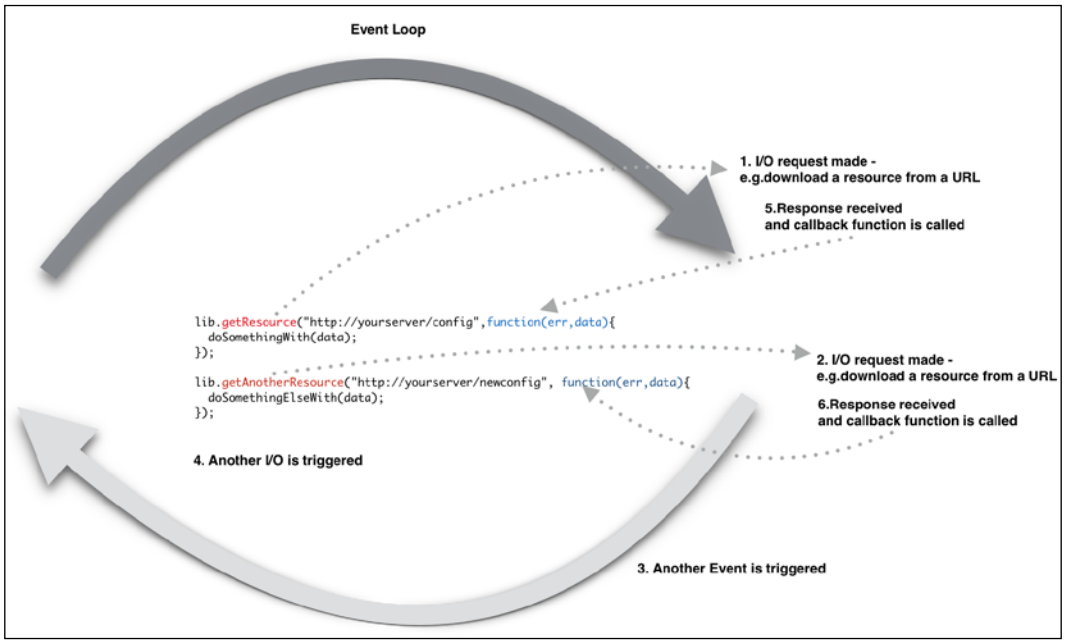
Essentially, there is only one thread running your code and there is no parallel execution; however, everything else except for your code runs in parallel.

Node.js relies on **libev** (`http://software.schmorp.de/pkg/libev.html`) to provide the event loop, which is supplemented by **libeio** (`http://software.schmorp.de/pkg/libeio.html`) that uses pooled threads to provide asynchronous I/O. To learn even more, take a look at the libev documentation at `http://pod.tst.eu/http://cvs.schmorp.de/libev/ev.pod`.

Consider the following example of asynchronous code execution in Node.js:

```
var fs = require('fs');
console.log('1');
fs.readFile('./response.json', function (error, data) {
  if(!error){
    console.log(data);
  });
console.log('2');
```

In this program, we read the `response.json` file from the disk. When the disk I/O is finished, the callback is executed with parameters containing the argument's error, if any error occurred, and data, which is the file data. What you will see in the console is the output of `console.log('1')` and `console.log('2')` one immediately after another:



Node.js does not need any additional server component as it creates its own server process. A Node application is essentially a server running on a designated port. In Node, the server and application are the same.

Here is an example of a Node.js server responding with the **Hello Node** string when the `http://localhost:3000/` URL is run from a browser:

```
var http = require('http');
var server = http.createServer();
server.on('request', function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello Node\n');
});
server.listen(3000);
```

In this example, we are using an `http` module. If you recall our earlier discussions on the JavaScript module, you will realize that this is the CommonJS module implementation. Node has several modules compiled into the binary. The core modules are defined within Node's source. They can be located in the `lib/` folder.

They are loaded first if their identifier is passed to `require()`. For instance, `require('http')` will always return the built-in HTTP module, even if there is a file by this name.

After loading the module to handle HTTP requests, we create a `server` object and use a listener for a `request` event using the `server.on()` function. The callback is called whenever there is a request to this server on port `3000`. The callback receives `request` and `response` parameters. We are also setting the `Content-Type` header and HTTP response code before we send the response back. You can copy the preceding code, save it in a plain text file, and name it `app.js`. You can run the server from the command line using Node.js as follows:

```
$ » node app.js
```

Once the server is started, you can open the `http://localhost:3000` URL in a browser and you will be greeted with unexciting text:



If you want to inspect what's happening internally, you can issue a `curl` command as follows:

```
~ » curl -v http://localhost:3000
* Rebuilt URL to: http://localhost:3000/
*   Trying ::1...
* Connected to localhost (::1) port 3000 (#0)
> GET / HTTP/1.1
> Host: localhost:3000
> User-Agent: curl/7.43.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: text/plain
< Date: Thu, 12 Nov 2015 05:31:44 GMT
< Connection: keep-alive
< Transfer-Encoding: chunked
<
Hello Node
* Connection #0 to host localhost left intact
```

Curl shows a nice request (>) and response (<) dialog including the request and response headers.

Callbacks

Callbacks in JavaScript usually take some time getting used to. If you are coming from some other non-asynchronous programming background, you will need to understand carefully how callbacks work; you may feel like you're learning programming for the first time. As everything is asynchronous in Node, you will be using callbacks for everything without trying to carefully structure them. The most important part of the Node.js project is sometimes the code organization and module management.

Callbacks are functions that are executed asynchronously at a later time. Instead of the code reading top to bottom procedurally, asynchronous programs may execute different functions at different times based on the order and speed that earlier functions such as HTTP requests or filesystem reads happen.

Whether a function execution is sequential or asynchronous depends on the context in which it is executed:

```
var i=0;
function add(num){
  console.log(i);
  i=i+num;
}
add(100);
console.log(i);
```

If you run this program using Node, you will see the following output (assuming that your file is named `app.js`):

```
~/Chapter9 » node app.js
0
100
```

This is what we are all used to. This is traditional synchronous code execution where each line is executed in a sequence. The code here defines a function and then on the next line calls this function, without waiting for anything. This is sequential control flow.

Things will be different if we introduced I/O to this sequence. If we try to read something from the file or call a remote endpoint, Node will execute these operations in an asynchronous fashion. For the next example, we are going to use a Node.js module called `request`. We will use this module to make HTTP calls. You can install the module as follows:

```
npm install request
```

We will discuss the use of npm later in this chapter. Consider the following example:

```
var request = require('request');
var status = undefined;
request('http://google.com', function (error, response, body) {
  if (!error && response.statusCode == 200) {
    status_code = response.statusCode;
  }
});
console.log(status);
```

When you execute this code, you will see that the value of the `status` variable is still `undefined`. In this example, we are making an HTTP call—this is an I/O operation. When we do an I/O operation, the execution becomes asynchronous. In the earlier example, we are doing everything within the memory and there was no I/O involved, hence, the execution was synchronous. When we run this program, all of the functions are immediately defined, but they don't all execute immediately. The `request()` function is called and the execution continues to the next line. If there is nothing to execute, Node will either wait for I/O to finish or it will exit. When the `request()` function finishes its work, it will execute the callback function (an anonymous function as the second parameter to the `request()` function). The reason that we got `undefined` in the preceding example is that nowhere in our code exists the logic that tells the `console.log()` statement to wait until the `request()` function has finished fetching the response from the HTTP call.

Callbacks are functions that get executed at some later time. This changes things in the way you organize your code. The idea around reorganizing the code is as follows:

- Wrapping the asynchronous code in a function
- Passing a callback function to the wrapper function

We will organize our previous example with these two ideas in mind. Consider this modified example:

```
var request = require('request');
var status = undefined;
function getSiteStatus(callback){
  request('http://google.com', function (error, response, body) {
    if (!error && response.statusCode == 200) {
      status_code = response.statusCode;
    }
    callback(status_code);
  });
}
function showStatusCode(status){
  console.log(status);
}
getSiteStatus(showStatusCode);
```

When you run this, you will get the following (correct) output:

```
$node app.js
200
```

What we changed was to wrap the asynchronous code in a `getSiteStatus()` function, pass a function named `callback()` as a parameter to this function, and execute this function on the last line of `getSiteStatus()`. The `showStatusCode()` callback function simply wraps around `console.log()` that we called earlier. The difference, however, is in the way the asynchronous execution works. The most important idea to understand while learning how to program with callbacks is that functions are first-class objects that can be stored in variables and passed around with different names. Giving simple and descriptive names to your variables is important in making your code readable by others. Now that the callback function is called once the HTTP call is completed, the value of the `status_code` variable will have a correct value. There are genuine circumstances where you want an asynchronous task executed only after another asynchronous task is completed. Consider this scenario:

```
http.createServer(function (req, res) {
  getURL(url, function (err, res) {
    getURLContent(res.data, function(err,res) {
      ...
    });
  });
});
```

As you can see, we are nesting one asynchronous function in another. This kind of nesting can result in code that is difficult to read and manage. This style of callback is sometimes known as **callback hell**. To avoid such a scenario, if you have code that has to wait for some other asynchronous code to finish, then you express that dependency by putting your code in functions that get passed around as callbacks. Another important idea is to name your functions instead of relying on anonymous functions as callbacks. We can restructure the preceding example into a more readable one as follows:

```
var urlContentProcessor = function(data){
  ...
}
var urlResponseProcessor = function(data){
  getURLContent(data,urlContentProcessor);
}
var createServer = function(req,res){
  getURL(url,urlResponseProcessor);
};
http.createServer(createServer);
```

This fragment uses two important concepts. First, we are using named functions and using them as callbacks. Second, we are not nesting these asynchronous functions. If you are accessing closure variables within the inner functions, the preceding would be a bit different implementation. In such cases, using inline anonymous functions is even more preferable.

Callbacks are most frequently used in Node. They are usually preferred to define logic for one-off responses. When you need to respond to repeating events, Node provides another mechanism for this. Before going further, we need to understand the function of timers and events in Node.

Timers

Timers are used to schedule the execution of a particular callback after a specific delay. There are two primary methods to set up such delayed execution: `setTimeout` and `setInterval`. The `setTimeout()` function is used to schedule the execution of a specific callback after a delay, while `setInterval` is used to schedule the repeated execution of a callback. The `setTimeout` function is useful to perform tasks that need to be scheduled such as housekeeping. Consider the following example:

```
setTimeout(function() {
  console.log("This is just one time delay");
},1000);
var count=0;
var t = setInterval(function() {
  count++;
  console.log(count);
  if (count> 5){
    clearInterval(t);
  }
}, 2000 );
```

First, we are using `setTimeout()` to execute a callback (the anonymous function) after a delay of 1,000 ms. This is just a one-time schedule for this callback. We scheduled the repeated execution of the callback using `setInterval()`. Note that we are assigning the value returned by `setInterval()` in a variable `t`—we can use this reference in `clearInterval()` to clear this schedule.

EventEmitters

We discussed earlier that callbacks are great for the execution of one-off logic. **EventEmitters** are useful in responding to repeating events. EventEmitters fire events and include the ability to handle these events when triggered. Several important Node APIs are built on EventEmitters.

Events raised by EventEmitters are handled through listeners. A listener is a callback function associated with an event—when the event fires, its associated listener is triggered as well. The `event.EventEmitter` is a class that is used to provide a consistent interface to emit (trigger) and bind callbacks to events.

As a common style convention, event names are represented by a camel-cased string; however, any valid string can be used as an event name.

Use `require('events')` to access the `EventEmitter` class:

```
var EventEmitter = require('events');
```

When an EventEmitter instance encounters an error, it emits an `error` event. Error events are treated as a special case in Node.js. If you don't handle these, the program exits with an exception stack.

All EventEmitters emit the `newListener` event when new listeners are added and `removeListener` when a listener is removed.

To understand the usage of EventEmitters, we will build a simplistic telnet server where different clients can log in and enter certain commands. Based on these commands, our server will respond accordingly:

```
var _net = require('net');
var _events = require ('events');
var _emitter = new events.EventEmitter();
_emitter.on('join', function(id,caller){
  console.log(id+" - joined");
});
_emitter.on('quit', function(id,caller){
  console.log(id+" - left");
});
```

```
var _server = _net.createServer(function(caller) {
  var process_id = caller.remoteAddress + ':' + caller.remotePort;
  _emitter.emit('join',id,caller);
  caller.on('end', function() {
    console.log("disconnected");
    _emitter.emit('quit',id,caller);
  });
});
_server.listen(8124);
```

In this code snippet, we are using the `net` module from Node. The idea here is to create a server and let the client connect to it via a standard `telnet` command. When a client connects, the server displays the client address and port, and when the client quits, the server logs this too.

When a client connects, we are emitting a `join` event, and when the client disconnects, we are emitting a `quit` event. We have listeners for both these events and they log appropriate messages on the server.

You start this program and connect to our server using telnet as follows:

```
telnet 127.0.0.1 8124
```

On the server console, you will see the server logging which client joined the server:

```
» node app.js
::ffff:127.0.0.1:51000 - joined
::ffff:127.0.0.1:51001 – joined
```

If any client quits the session, an appropriate message will appear as well.

Modules

When you are writing a lot of code, you soon reach a point where you have to start thinking about how you want to organize the code. Node modules are CommonJS modules that we discussed earlier when we discussed module patterns. Node modules can be published to the **Node Package Manager** (**npm**) repository. The npm repository is an online collection of Node modules.

Creating modules

Node modules can be either single files or directories containing one or more files. It's usually a good idea to create a separate module directory. The file in the module directory that will be evaluated is normally named `index.js`. A module directory can look as follows:

```
node_project/src/nav
                --- >index.js
```

In your project directory, the `nav` module directory contains the module code. Conventionally, your module code needs to reside in the `index.js` file—you can change this to another file if you want. Consider this trivial module called `geo.js`:

```
exports.area = function (r) {
  return 3.14 * r * r;
};
exports.circumference = function (r) {
  return 3.14 * 3.14 * r;
};
```

You are exporting two functions via `exports`. You can use the module using the `require` function. This function takes the name of the module or system path to the module's code. You can use the module that we created as follows:

```
var geo = require('./geo.js');
console.log(geo.area(2));
```

As we are exporting only two functions to the outside world, everything else remains private. If you recollect, we discussed the module pattern in detail—Node uses CommonJS modules. There is an alternative syntax to create modules as well. You can use `modules.exports` to export your modules. Indeed, `exports` is a helper created for `modules.exports`. When you use `exports`, it attaches the exported properties of a module to `modules.exports`. However, if `modules.exports` already has some properties attached to it, properties attached by `exports` are ignored.

The `geo` module created earlier in this section can be rewritten in order to return a single `Geo` constructor function rather than an object containing functions. We can rewrite the `geo` module and its usage as follows:

```
var Geo = function(PI) {
  this.PI = PI;
}
Geo.prototype.area = function (r) {
  return this.PI * r * r;
};
Geo.prototype.circumference = function (r) {
  return this.PI * this.PI * r;
};
module.exports = Geo;
```

Consider a `config.js` module:

```
var db_config = {
  server: "0.0.0.0",
  port: "3306",
  user: "mysql",
  password: "mysql"
};
module.exports = db_config;
```

If you want to access `db_config` from outside this module, you can use `require()` to include the module and refer the object as follows:

```
var config = require('./config.js');
console.log(config.user);
```

There are three ways to organize modules:

- Using a relative path, for example, `config = require('./lib/config.js')`
- Using an absolute path, for example, `config = require('/nodeproject/lib/config.js')`
- Using a module search, for example, `config = require('config')`

The first two are self-explanatory—they allow Node to look for a module in a particular location in the filesystem.

When you use the third option, you are asking Node to locate the module using the standard look method. To locate the module, Node starts at the current directory and appends `./node_modules/` to it. Node then attempts to load the module from this location. If the module is not found, then the search starts from the parent directory until the root of the filesystem is reached.

For example, if `require('config')` is called in `/projects/node/`, the following locations will be searched until a match a found:

- `/projects/node /node_modules/config.js`
- `/projects/node_modules/config.js`
- `/node_modules/config.js`

For modules downloaded from npm, using this method is relatively simple. As we discussed earlier, you can organize your modules in directories as long as you provide a point of entry for Node.

The easiest way to do this is to create the `./node_modules/supermodule/` directory, and insert an `index.js` file in this directory. The `index.js` file will be loaded by default. Alternatively, you can put a `package.json` file in the `mymodulename` folder, specifying the name and main file of the module:

```
{
  "name": "supermodule",
  "main": "./lib/config.js"
}
```

You have to understand that Node caches modules as objects. If you have two (or more) files requiring a specific module, the first `require` will cache the module in memory so that the second `require` will not have to reload the module source code. However, the second `require` can alter the module functionality if it wishes to. This is commonly called **monkey patching** and is used to modify a module behavior without really modifying or versioning the original module.

npm

The npm is the package manager used by Node to distribute modules. The npm can be used to install, update, and manage modules. Package managers are popular in other languages such as Python. The npm automatically resolves and updates dependencies for a package and hence makes your life easy.

Installing packages

There are two ways to install npm packages: locally or globally. If you want to use the module's functionality only for a specific Node project, you can install it locally relative to the project, which is default behavior of `npm install`. Alternatively, there are several modules that you can use as a command-line tool; in this case, you can install them globally:

```
npm install request
```

The `install` directive with `npm` will install a particular module—`request` in this case. To confirm that `npm install` worked correctly, check to see whether a `node_modules` directory exists and verify that it contains a directory for the package(s) that you installed.

As you start adding modules to your project, it becomes difficult to manage the version/dependency of each module. The best way to manage locally installed packages is to create a `package.json` file in your project.

A `package.json` file can help you in the following ways:

- Defining versions of each module that you want to install. There are times when your project depends on a specific version of a module. In this case, your `package.json` helps you download and maintain the correct version dependency.
- Serving as a documentation of all the modules that your project needs.
- Deploying and packaging your application without worrying about managing dependencies every time you deploy the code.

You can create `package.json` by issuing the following command:

```
npm init
```

After answering basic questions about your project, a blank `package.json` is created with content similar to the following:

```
{
  "name": "chapter9",
  "version": "1.0.0",
  "description": "chapter9 sample project",
  "main": "app.js",
  "dependencies": {
    "request": "^2.65.0"
  },
```

```
  "devDependencies": {},
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "Chapter9",
    "sample",
    "project"
  ],
  "author": "Ved Antani",
  "license": "MIT"
}
```

You can manually edit this file in a text editor. An important part of this file is the `dependencies` tag. To specify the packages that your project depends on, you need to list the packages you'd like to use in your `package.json` file. There are two types of packages that you can list:

- `dependencies`: These packages are required by your application in production
- `devDependencies`: These packages are needed only for development and testing (for example, using the **Jasmine node package**)

In the preceding example, you can see the following dependency:

```
"dependencies": {
  "request": "^2.65.0"
},
```

This means that the project is dependent on the `request` module.



The version of the module is dependent on the semantic versioning rules—`https://docs.npmjs.com/getting-started/semantic-versioning`.

Once your `package.json` file is ready, you can simply use the `npm install` command to install all the modules for your projects automatically.

There is a cool trick that I love to use. While installing modules from the command line, we can add the `--save` flag to add that module's dependency to the `package.json` file automatically:

```
npm install async --save
npm WARN package.json chapter9@1.0.0 No repository field.
npm WARN package.json chapter9@1.0.0 No README data
async@1.5.0 node_modules/async
```

In the preceding command, we installed the `async` module with the normal `npm` command with a `--save` flag. There is a corresponding entry automatically created in `package.json`:

```
"dependencies": {
  "async": "^1.5.0",
  "request": "^2.65.0"
},
```

JavaScript performance

Like any other language, writing correct JavaScript code at scale is an involved task. As the language matures, several of the inherent problems are being taken care of. There are several exceptional libraries that aid in writing good quality code. For most serious systems, *good code = correct code + high performance code*. The demands of new-generation software systems are high on performance. In this section, we will discuss a few tools that you can use to analyze your JavaScript code and understand its performance metrics.

We will discuss the following two ideas in this section:

- Profiling: Timing various functions and operations during script-profiling helps in identifying areas where you can optimize your code
- Network performance: Examining the loading of network resources such as images, stylesheets, and scripts

JavaScript profiling

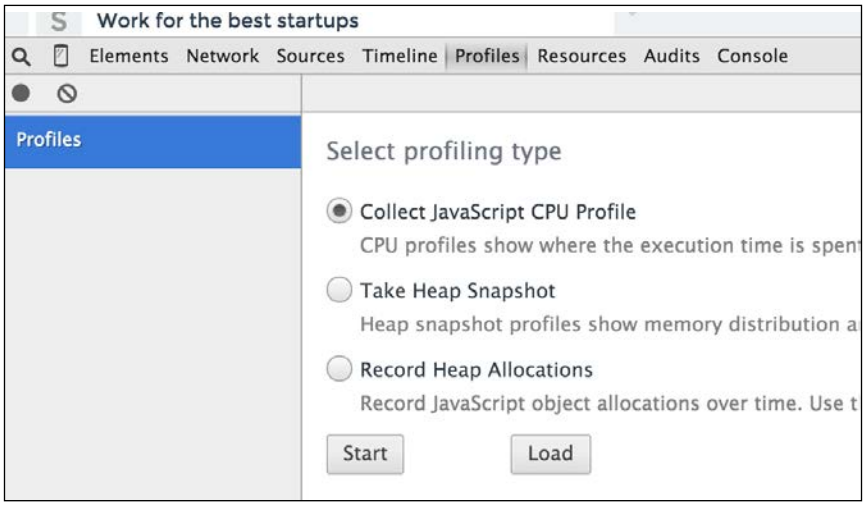
JavaScript profiling is critical to understand performance aspects of various parts of your code. You can observe timings of the functions and operations to understand which operation is taking more time. With this information, you can optimize the performance of time-consuming functions and tune the overall performance of your code. We will be focusing on the profiling options provided by Chrome's Developer Tools. There are comprehensive analysis tools that you can use to understand the performance metrics of your code.

The CPU profile

The CPU profile shows the execution time spent by various parts of your code. We have to inform DevTools to record the CPU profile data. Let's take the profiler for a spin.

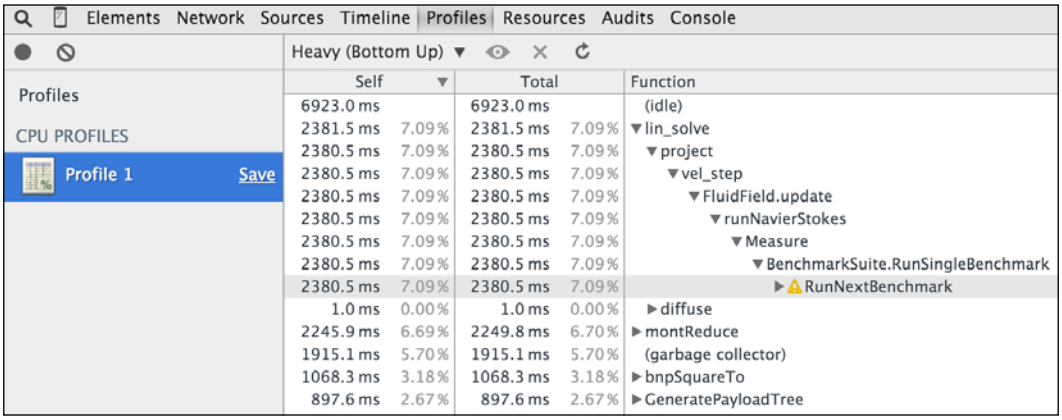
You can enable the CPU profiler in DevTools as follows:

1. Open the Chrome DevTools **Profiles** panel.
2. Verify that **Collect JavaScript CPU Profile** is selected:



For this chapter, we will be using Google's own benchmark page, `http://octane-benchmark.googlecode.com/svn/latest/index.html`. We will use this because it contains sample functions where we can see various performance bottlenecks and benchmarks. To start recording the CPU profile, open DevTools in Chrome, and in the **Profiles** tab, click on the **Start** button or press *Cmd/Ctrl + E*. Refresh the **V8 Benchmark Suite** page. When the page has completed reloading, a score for the benchmark tests is shown. Return to the **Profiles** panel and stop the recording by clicking on the **Stop** button or pressing *Cmd/Ctrl + E* again.

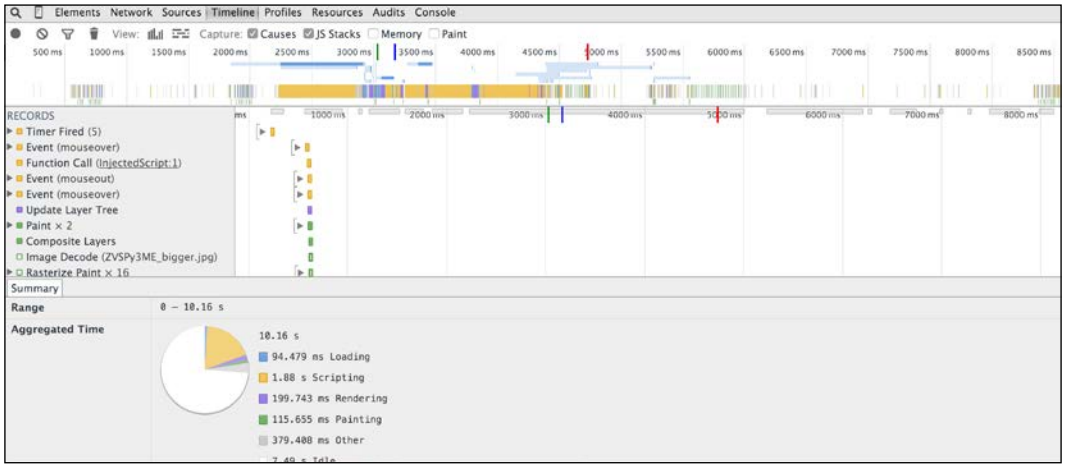
The recorded CPU profile shows you a detailed view of the functions and the execution time taken by them in the bottom-up fashion, as shown in the following image:



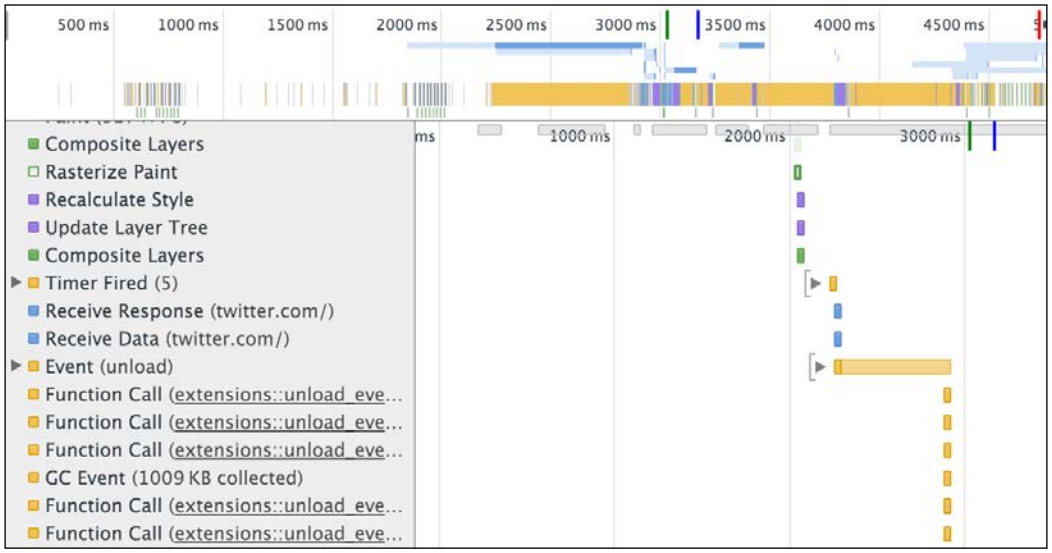
The Timeline view

The Chrome DevTools **Timeline** tool is the first place you can start looking at the overall performance of your code. It lets you record and analyze all the activity in your application as it runs.

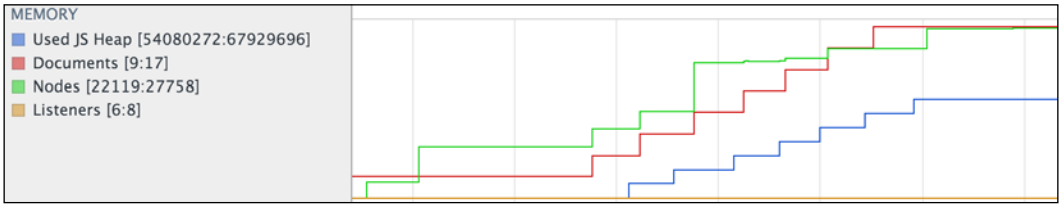
The **Timeline** provides you with a complete overview of where time is spent when loading and using your site. A timeline recording includes a record for each event that occurred and is displayed in a **waterfall** graph:



The preceding screen shows you the timeline view when we try to render `https://twitter.com/` in the browser. The timeline view gives you an overall view of which operation took how much time in execution:



In the preceding screenshot, we can see the progressive execution of various JavaScript functions, network calls, resource downloads, and other operations involved in rendering the Twitter home page. This view gives us a very good idea about which operations may be taking longer. Once we identify such operations, we can optimize them for performance. The **Memory** view is a great tool to understand how the memory is used during the lifetime of your application in the browser. The **Memory** view shows you a graph of the memory used by your application over time and maintains a counter of the number of documents, DOM nodes, and event listeners that are held in the memory. The **Memory** view can help detect memory leaks and give you good enough hints to understand what optimizations are required:



JavaScript performance is a fascinating subject and deserves its own dedicated text. I would urge you to explore Chrome's DevTools and understand how best to use the tools to detect and diagnose performance problems in your code.

Summary

In this chapter, we looked at a different avatar of JavaScript—that of a server-side framework in the form of Node.js.

Node offers an asynchronous evented-model to program scalable and high-performance server applications in JavaScript. We dived deep into some core concepts on Node, such as an event loop, callbacks, modules, and timers. Understanding them is critical to write good Node code. We also discussed several techniques to structure Node code and callbacks in a better way.

With this, we reach the conclusion of our exploration of a brilliant programming language. JavaScript has been instrumental in the evolution of the World Wide Web because of its sheer versatility. The language continues to expand its horizons and improves with each new iteration.

We started our journey with understanding the building blocks of the grammar and syntax of the language. We grasped the fundamental ideas of closures and the functional behavior of JavaScript. These concepts are so essential that most of the JavaScript patterns are based on them. We looked at how we can utilize these patterns to write better code with JavaScript. We studied how JavaScript can operate on a DOM and how to use jQuery to manipulate the DOM effectively. Finally, we looked at the server-side avatar of JavaScript in Node.js.

This book should have enabled you to think differently when you start programming in JavaScript. Not only will you think about common patterns when you code, but also appreciate and use newer language features by ES6.

Module 2

Mastering JavaScript Design Patterns - Second Edition

Write reliable code to create powerful applications by mastering advanced JavaScript design patterns

1
Designing for Fun and Profit

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 topics in this chapter are as follows:

- History of JavaScript
- What is a design pattern?
- Anti-patterns

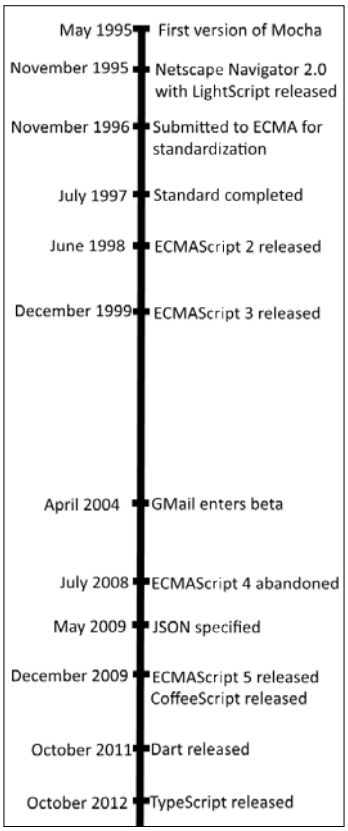
The road to JavaScript

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.

For many years, every JavaScript script would start by attempting to determine the browser on which it was running. This would dictate how to address elements in the DOM, as there were dramatic deviations between each browser. The spaghetti of code that was required to perform simple actions was legendary. I remember reading a year-long 20-part series on developing a **Dynamic HTML** (**DHTML**) drop down menu such that it would work on both Internet Explorer and Netscape Navigator. The same functionally can now be achieved with pure CSS without even having to resort to 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 fluid 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 is 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?

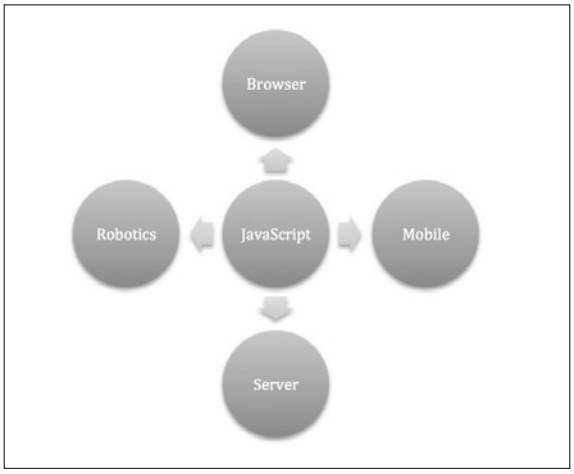
JavaScript everywhere

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 Node.js. Node.js started off as a highly asynchronous method of writing server-side applications. 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 have 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 becoming one of the most important languages in the world. Although language usage statistics are notoriously difficult to calculate, every single source which attempts to develop a ranking puts JavaScript in the top 10:

Language index	**Rank of JavaScript**
Langpop.com	4
Statisticbrain.com	4
Codeval.com	6
TIOBE	8

What is more interesting is that most of of these rankings suggest that the usage of JavaScript is on the rise.

The long and short of it is that JavaScript is going to be a major language in the next few years. 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 Overflow 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 a file 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?

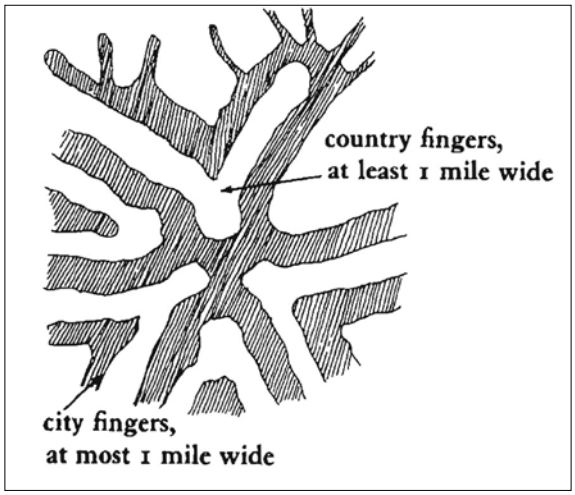
For the most part, ideas are only applicable in one place. Adding peanut butter is really only a great idea in cooking and not in sewing. However, from time to time it is possible to find applicability for a great idea outside of its original purpose. This is the story behind design patterns.

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 domain-driven design. The design patterns book has been so important that it is commonly referred do 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 a 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.

There are some who suggest that design patterns should be emergent. That is to say, that by simply writing software in an intelligent way, one can see the patterns emerge from the implementation. I think that may be an accurate statement, however, it ignores the actual cost of getting to those implementations by trial and error. Those with an awareness of design patterns are much more likely to spot the emergent pattern early on. Teaching junior programmers about patterns is a very useful exercise. Knowing early on which pattern or patterns can be applied acts as a shortcut. The full solution can be arrived at earlier and with fewer missteps.

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 any 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 filed away ready for the next project to make use of. At least, that is the theory. For the most part, the document is filed away and never used again. It is difficult to create lessons that are globally applicable. The lessons learned tend to only be useful 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 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 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.

Part 1

Classical Design Patterns

- Organizing Code
- Creational Patterns
- Structural Patterns
- Behavioral Patterns

2
Organizing Code

In this chapter we'll look at how to organize JavaScript code into reusable, understandable chunks. The language itself doesn't lend itself well to this sort of modularization but a number of methods of organizing JavaScript code have emerged over the years. This chapter will argue for the need to break down code and then work through the methods of creating JavaScript modules.

We will cover the following topics:

- Global scope
- Objects
- Prototype inheritance
- ECMAScript 2015 classes

Chunks of code

The first thing anybody learns to program is the ubiquitous Hello World application. This simple application prints some variation of "hello world" to the screen. Depending on who you ask, the phrase hello world dates back to the early 1970s where it was used to demonstrate the B programming language or even to 1967 where it appears in a BCL programming guide. In such a simple application there is no need to worry about the structure of code. Indeed, in many programming languages, hello world needs no structure at all.

For Ruby, it is as follows:

```
#!/usr/bin/ruby
puts "hello world"
```

For JavaScript (via Node.js), it is as follows:

```
#!/usr/local/bin/node
console.log("Hello world")
```

Programming modern computers was originally done using brutally simplistic techniques. Many of the first computers had problems they were attempting to solve hard-wired into them. They were not general purpose computing machines like the ones we have today. Instead they were built to solve just one problem such as decoding encrypted texts. Stored program computers were first developed in the late 1940s.

The languages used to program these computers were complicated at first, usually very closely tied to the binary. Eventually higher and higher-level abstractions were created to make programming more accessible. As these languages started to take shape through the 50s and 60s it quickly became apparent that there needed to be some way to divide up large blocks of code.

In part this was simply to maintain the sanity of programmers who could not keep an entire, large program in their heads at any one time. However, creating reusable modules also allowed for code to be shared within an application and even between applications. The initial solution was to make use of statements, which jumped the flow control of the program from one place to another. For a number of years these GOTO statements were heavily relied upon. To a modern programmer who has been fed a continual stream of warnings about the use of GOTO statements this seems like insanity. However it was not until some years after the first programming languages emerged that structured programming grew to replace the GOTO syntax.

Structured programming is based on the Böhm-Jacopini theorem, which states that there is a rather large class of problems, the answer to which can be computed using three very simple constructs:

- Sequential execution of sub-programs
- Conditional execution of two sub-programs
- Repeated execution of a sub-program until a condition is true

Astute readers will recognize these constructs as being the normal flow of execution, a branch or `if` statement, and a loop.

Fortran was one of the earliest languages and was initially built without support for structured programming. However structured programming was soon adopted as it helped to avoid spaghetti code.

Code in Fortran was organized into modules. Modules were loosely coupled collections of procedures. For those coming from a modern object oriented language, the closest concept might be that a module was like a class that contains only static methods.

Modules were useful for dividing code into logical groupings. However, it didn't provide for any sort of structure for the actual applications. The structure for object-oriented languages, that is classes and subclasses, can be traced to a 1967 paper written by Ole-Johan Dahl and Kristen Nygaard. This paper would go on to form the basis of Simula-67, the first language with support for object oriented programming.

While Simula-67 was the first language to have classes, the language most talked about in relation to early object oriented programming is Smalltalk. This language was developed behind closed doors at the famous Xerox **Palo Alto Research Center** (**PARC**) during the 1970s. it was released to the public in 1980 as Smalltalk-80 (it seems like all historically relevant programming languages where prefixed with the year of release as a version number). What Smalltalk brought was that everything in the language was an object, even literal numbers like 3 could have operations performed on them.

Almost every modern programming language has some concept of classes to organize code. Often these classes will fall into a higher-level structure commonly called a namespace or module. Through the use of these structures, even very large programs can be divided into manageable and understandable chunks.

Despite the rich history and obvious utility of classes and modules, JavaScript did not support them as first class constructs until just recently. To understand why, one has to simply look back at the history of JavaScript from *Chapter 1*, *Designing For Fun and Profit*, and realize that for its original purpose having such constructs would have been overkill. Classes were a part of the ill-fated ECMAScript 4 standard and they finally became part of the language with the release of the ECMAScript 2015 standard.

In this chapter we'll explore some of the ways to recreate the well worn class structure of other modern programming languages in JavaScript.

What's the matter with global scope anyway?

In browser based JavaScript every object you create is assigned to the global scope. For the browser, this object is simply known as **window**. It is simple to see this behavior in action by opening up the development console in your favorite browser.



Opening the Development Console

Modern browsers have, built into them, some very advanced debugging and auditing tools. To access them there is a menu item, which is located under **Tools** | **Developer Tools in Chrome** | **Tools** | **Web Developer in Firefox**, and directly under the menu as **F12 Developer Tools** in Internet Explorer. Keyboard shortcuts also exist for accessing the tools. On Windows and Linux, *F12* is standard and, on OSX, `Option` + `Command` + `I` is used.

Within the developer tools is a console window that provides direct access to the current page's JavaScript. This is a very handy place to test out small snippets of code or to access the page's JavaScript.

Once you have the console open, enter the following code:

```
> var words = "hello world"
> console.log(window.words);
```

The result of this will be `hello world` printed to the console. By declaring words globally it is automatically attached to the top level container: window.

In Node.js the situation is somewhat different. Assigning a variable in this fashion will actually attach it to the current module. Not including the `var` object will attach the variable to the `global` object.

For years you've likely heard that making use of global variables is a bad thing. This is because globals are very easily polluted by other code.

Consider a very commonly named variable such as `index`. It is likely that in any application of appreciable size that this variable name would be used in several places. When either piece of code makes use of the variable it will cause unexpected results in the other piece of code. It is certainly possible to reuse variables, and it can even be useful in systems with very limited memory such as embedded systems, but in most applications reusing variables to mean different things within a single scope is difficult to understand and a source of errors.

Applications that make use global scoped variables also open themselves up to being attacked on purpose by other code. It is trivial to alter the state of global variables from other code, which could expose secrets like login information to attackers.

Finally global variables add a great deal of complexity to applications. Reducing the scope of variables to a small section of code allows developers to more easily understand the ways in which the variable is used. When the scope is global then changes to that variable may have an effect far outside of the one section of code. A simple change to a variable can cascade into the entire application.

As a general rule global variables should be avoided.

Objects in JavaScript

JavaScript is an object oriented language but most people don't make use of the object oriented features of it except in passing. JavaScript uses a mixed object model in that it has some primitives as well as objects. JavaScript has five primitive types:

- undefined
- null
- boolean
- string
- number

Of these five, only two are what we would expect to be an object anyway. The other three, boolean, string, and number all have wrapped versions, which are objects: Boolean, String, and Number. They are distinguished by starting with uppercase. This is the same sort of model used by Java, a hybrid of objects and primitives.

JavaScript will also box and unbox the primitives as needed.

In this code you can see the boxed and unboxed versions of JavaScript primitives at work:

```
var numberOne = new Number(1);
var numberTwo = 2;
typeof numberOne; //returns 'object'
typeof numberTwo; //returns 'number'
var numberThree = numberOne + numberTwo;
typeof numberThree; //returns 'number'
```

Creating objects in JavaScript is trivial. This can be seen in this code for creating an object in JavaScript:

```
var objectOne = {};
typeof objectOne; //returns 'object'
var objectTwo = new Object();
typeof objectTwo; //returns 'object'
```

Because JavaScript is a dynamic language, adding properties to objects is also quite easy. This can be done even after the object has been created. This code creates the object:

```
var objectOne = { value: 7 };
var objectTwo = {};
objectTwo.value = 7;
```

Objects contain both data and functionality. We've only seen the data part so far. Fortunately in JavaScript, functions are first class objects. Functions can be passed around and functions can be assigned to variables. Let's try adding some functions to the object we're creating in this code:

```
var functionObject = {};
functionObject.doThings = function() {
  console.log("hello world");
}
functionObject.doThings(); //writes "hello world" to the console
```

This syntax is a bit painful, building up objects an assignment at a time. Let's see if we can improve upon the syntax for creating objects:

```
var functionObject = {
  doThings: function() {
    console.log("hello world");
  }
}
functionObject.doThings();//writes "hello world" to the console
```

This syntax seems, at least to me, to be a much cleaner, more traditional way of building objects. Of course it is possible to mix data and functionality in an object in this fashion:

```
var functionObject = {
  greeting: "hello world",
  doThings: function() {
    console.log(this.greeting);
  }
}
functionObject.doThings();//prints hello world
```

There are a couple of things to note in this piece of code. The first is that the different items in the object are separated using a comma and not a semi-colon. Those coming from other languages such as C# or Java are likely to make this mistake.The next item of interest is that we need to make use of the `this` qualifier to address the `greeting` variable from within the `doThings` function. This would also be true if we had a number of functions within the object as shown here:

```
var functionObject = {
  greeting: "hello world",
  doThings: function() {
    console.log(this.greeting);
    this.doOtherThings();
  },
  doOtherThings: function() {
    console.log(this.greeting.split("").reverse().join(""));
  }
}
functionObject.doThings();//prints hello world then dlrow olleh
```

The `this` keyword behaves differently in JavaScript than you might expect coming from other C-syntax languages. `this` is bound to the owner of the function in which it is found. However, the owner of the function is sometimes not what you expect. In the preceding example `this` is bound to the `functionObject` object, however if the function were declared outside of an object this would refer to the global object. In certain circumstances, typically event handlers, this is rebound to the object firing the event.

Let's look at the following code:

```
var target = document.getElementById("someId");
target.addEventListener("click", function() {
  console.log(this);
}, false);
```

`this` takes on the value of target. Getting used to the value of `this` is, perhaps, one of the trickiest things in JavaScript.

ECMAScript-2015 introduces the `let` keyword which can replace the `var` keyword for declaring variables. `let` uses block level scoping which is the scoping you're likely to use from most languages. Let's see an example of how they differ:

```
for(var varScoped =0; varScoped <10; varScoped++)
{
  console.log(varScoped);
}
console.log(varScoped +10);
```

```
for(let letScoped =0; letScoped<10; letScoped++)
{
  console.log(letScoped);
}
console.log(letScoped+10);
```

With the var scoped version you can see that the variable lives on outside of the block. This is because behind the scenes the declaration of `varScoped` is hoisted to the beginning of the code block. With the `let` scoped version of the code `letScoped` is scoped just within the `for` loop so, once we leave the loop, `letScoped` is undefined. When given the option of using `let` or `var` we would tend to err on the side of always using `let`. There are some cases when you actually would want to use var scoping but they are few and far between.

We have built up a pretty complete model of how to build objects within JavaScript. However, objects are not the same thing as classes. Objects are instances of classes. If we want to create multiple instances of our `functionObject` object we're out of luck. Attempting to do so will result in an error. In the case of Node.js the error will be as follows:

```
let obj = new functionObject();
TypeError: object is not a function
  at repl:1:11
  at REPLServer.self.eval (repl.js:110:21)
  at repl.js:249:20
  at REPLServer.self.eval (repl.js:122:7)
  at Interface.<anonymous> (repl.js:239:12)
  at Interface.EventEmitter.emit (events.js:95:17)
  at Interface._onLine (readline.js:202:10)
  at Interface._line (readline.js:531:8)
  at Interface._ttyWrite (readline.js:760:14)
  at ReadStream.onkeypress (readline.js:99:10)
```

The stack trace here shows an error in a module called `repl`. This is the read-execute-print loop that is loaded by default when starting Node.js.

Each time that a new instance is required, the object must be reconstructed. To get around this we can define the object using a function as can be seen here:

```
let ThingDoer = function(){
  this.greeting = "hello world";
  this.doThings = function() {
    console.log(this.greeting);
    this.doOtherThings();
  };
```

```
  this.doOtherThings = function() {
    console.log(this.greeting.split("").reverse().join(""));
  };
}
let instance = new ThingDoer();
instance.doThings(); //prints hello world then dlrow olleh
```

This syntax allows for a constructor to be defined and for new objects to be created from this function. Constructors without return values are functions that are called as an object is created. In JavaScript the constructor actually returns the object created. You can even assign internal properties using the constructor by making them part of the initial function like so:

```
let ThingDoer = function(greeting){
  this.greeting = greeting;
  this.doThings = function() {
    console.log(this.greeting);
  };
}
let instance = new ThingDoer("hello universe");
instance.doThings();
```

Build me a prototype

As previously mentioned, there was, until recently, no support for creating true classes in JavaScript. While ECMAScript-2015 brings some syntactic sugar to classes, the underlying object system is still as it has been in the past, so it remains instructive to see how we would have created objects without this sugar. Objects created using the structure in the previous section have a fairly major drawback: creating multiple objects is not only time consuming but also memory intensive. Each object is completely distinct from other objects created in the same fashion. This means that the memory used to hold the function definitions is not shared between all instances. What is even more fun is that you can redefine individual instances of a class without changing all of the instances. This is demonstrated in this code:

```
let Castle = function(name){
  this.name = name;
  this.build = function() {
    console.log(this.name);
  };
}
let instance1 = new Castle("Winterfell");
let instance2 = new Castle("Harrenhall");
```

```
instance1.build = function(){ console.log("Moat Cailin");}
instance1.build(); //prints "Moat Cailin"
instance2.build(); //prints "Harrenhall" to the console
```

Altering the functionality of a single instance or really of any already defined object in this fashion is known as **monkey patching**. There is some division over whether or not this is a good practice. It can certainly be useful when dealing with library code but it adds great confusion. It is generally considered better practice to extend the existing class.

Without a proper class system JavaScript, of course, has no concept of inheritance. However, it does have a prototype. At the most basic level an object in JavaScript is an associative array of keys and values. Each property or function on an object is simply defined as part of this array. You can even see this in action by accessing members of an object using array syntax as is shown here:

```
let thing = { a: 7};
console.log(thing["a"]);
```



Accessing members of an object using array syntax can be a very handy way to avoid using the eval function. For instance, if I had the name of the function I wanted to call in a string called `funcName` and I wanted to call it on an object, `obj1`, then I could do so by doing `obj1[funcName]()` instead of using a potentially dangerous call to eval. Eval allows for arbitrary code to be executed. Allowing this on a page means that an attacker may be able to enter malicious scripts on other people's browsers.

When an object is created, its definition is inherited from a prototype. Weirdly each prototype is also an object so even prototypes have prototypes. Well, except for the object which is the top-level prototype. The advantage to attaching functions to the prototype is that only a single copy of the function is created; saving on memory. There are some complexities to prototypes but you can certainly survive without knowing about them. To make use of a prototype you need to simply assign functions to it as is shown here:

```
let Castle = function(name){
  this.name = name;
}
Castle.prototype.build = function(){ console.log(this.name);}
let instance1 = new Castle("Winterfell");
instance1.build();
```

One thing to note is that only the functions are assigned to the prototype. Instance variables such as `name` are still assigned to the instance. As these are unique to each instance there is no real impact on the memory usage.

In many ways a prototypical language is more powerful than a class-based inheritance model.

If you make a change to the prototype of an object at a later date then all the objects which share that prototype are updated with the new function. This removes some of the concerns expressed about monkey typing. An example of this behavior is shown here:

```
let Castle = function(name){
  this.name = name;
}
Castle.prototype.build = function(){
  console.log(this.name);
}
let instance1 = new Castle("Winterfell");
Castle.prototype.build = function(){
  console.log(this.name.replace("Winterfell", "Moat Cailin"));
}
instance1.build();//prints "Moat Cailin" to the console
```

When building up objects you should be sure to take advantage of the prototype object whenever possible.

Now we know about prototypes there is an alternative approach to building objects in JavaScript and that is to use the `Object.create` function. This is a new syntax introduced in ECMAScript 5. The syntax is as follows:

```
Object.create(prototype [, propertiesObject ] )
```

The create syntax will build a new object based on the given prototype. You can also pass in a `propertiesObject` object that describes additional fields on the created object. These descriptors consist of a number of optional fields:

- `writable`: This dictates whether the field should be writable
- `configurable`: This dictates whether the files should be removable from the object or support further configuration after creation
- `enumerable`: This dictates whether the property can be listed during an enumeration of the object's properties
- `value`: This dictates the default value of the field

It is also possible to assign a `get` and `set` functions within the descriptor that act as getters and setters for some other internal property.

Using `object.create` for our castle we can build an instance using `Object.create` like so:

```
let instance3 = Object.create(Castle.prototype, {name: { value:
"Winterfell", writable: false}});
instance3.build();
instance3.name="Highgarden";
instance3.build();
```

You'll notice that we explicitly define the `name` field. `Object.create` bypasses the constructor so the initial assignment we described in the preceding code won't be called. You might also notice that writeable is set to `false`. The result of this is that the reassignment of `name` to `Highgarden` has no effect. The output is as follows:

```
Winterfell
Winterfell
```

Inheritance

One of the niceties of objects is that they can be built upon to create increasingly complex objects. This is a common pattern, which is used for any number of things. There is no inheritance in JavaScript because of its prototypical nature. However, you can combine functions from one prototype into another.

Let's say that we have a base class called `Castle` and we want to customize it into a more specific class called `Winterfell`. We can do so by first copying all of the properties from the `Castle` prototype onto the `Winterfell` prototype. This can be done like so:

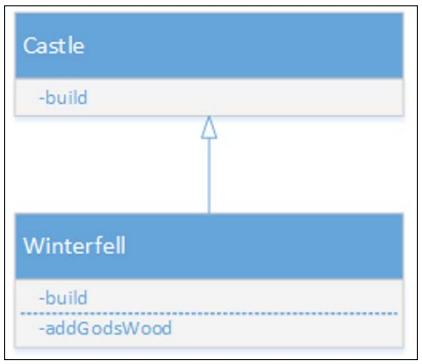
```
let Castle = function(){};
Castle.prototype.build = function(){console.log("Castle built");}

let Winterfell = function(){};
Winterfell.prototype.build = Castle.prototype.build;
Winterfell.prototype.addGodsWood = function(){}
let winterfell = new Winterfell();
winterfell.build(); //prints "Castle built" to the console
```

Of course this is a very painful way to build objects. You're forced to know exactly which functions the base class has to copy them. It can be abstracted in a rather naïve fashion like this:

```
function clone(source, destination) {
```

```
    for(var attr in source.prototype){ destination.prototype[attr] =
      source.prototype[attr];}
}
```

If you are into object diagrams this shows how **Winterfell** extends **Castle** in this diagram:



This can be used quite simply as follows:

```
let Castle = function(){};
Castle.prototype.build = function(){console.log("Castle built");}

let Winterfell = function(){};
clone(Castle, Winterfell);
let winterfell = new Winterfell();
winterfell.build();
```

We say that this is naïve because it fails to take into account a number of potential failure conditions. A fully-fledged implementation is quite extensive. The jQuery library provides a function called `extend` which implements prototype inheritance in a robust fashion. It is about 50 lines long and deals with deep copies and null values. The function is used extensively, internally in jQuery but it can be a very useful function in your own code. We mentioned that prototype inheritance is more powerful than the traditional methods of inheritance. This is because it is possible to mix and match bits from many base classes to create a new class. Most modern languages only support single inheritance: a class can have only one direct parent. There are some languages with multiple inheritance however, it is a practice that adds a great deal of complexity when attempting to decide which version of a method to call at runtime. Prototype inheritance avoids many of these issues by forcing selection of a method at assembly time.

Composing objects in this fashion permits taking properties from two or more different bases. There are many times when this can be useful. For example a class representing a wolf might take some of its properties from a class describing a dog and some from another class describing a quadruped.

By using classes built in this way we can meet pretty much all of the requirements for constructing a system of classes including inheritance. However inheritance is a very strong form of coupling. In almost all cases it is better to avoid inheritance in favor of a looser form of coupling. This will allow for classes to be replaced or altered with a minimum impact on the rest of the system.

Modules

Now that we have a complete class system it would be good to address the global namespace discussed earlier. Again there is no first class support for namespaces but we can easily isolate functionality to the equivalent of a namespace. There are a number of different approaches to creating modules in JavaScript. We'll start with the simplest and add some functionality as we go along.

To start we simply need to attach an object to the global namespace. This object will contain our root namespace. We'll name our namespace `Westeros`; the code simply looks like:

```
Westeros = {}
```

This object is, by default, attached to the top level object so we need not do anything more than that. A typical usage is to first check if the object already exists and use that version instead of reassigning the variable. This allows you to spread your definitions over a number of files. In theory you could define a single class in each file and then bring them all together as part of the build process before delivering them to the client or using them in an application. The short form of this is:

```
Westeros = Westeros || {}
```

Once we have the object, it is simply a question of assigning our classes as properties of that object. If we continue to use the `Castle` object then it would look like:

```
let Westeros = Westeros || {};
Westeros.Castle = function(name){this.name = name}; //constructor
Westeros.Castle.prototype.Build = function(){console.log("Castle
  built: " +  this.name)};
```

If we want to build a hierarchy of namespaces that is more than a single level deep, that too is easily accomplished, as seen in this code:

```
let Westeros = Westeros || {};
Westeros.Structures = Westeros.Structures || {};
Westeros.Structures.Castle = function(name){ this.name = name};
//constructor
Westeros.Structures.Castle.prototype.Build =
  function(){console.log("Castle built: " +  this.name)};
```

This class can be instantiated and used in a similar way to previous examples:

```
let winterfell = new Westeros.Structures.Castle("Winterfell");
winterfell.Build();
```

Of course with JavaScript there is more than one way to build the same code structure. An easy way to structure the preceding code is to make use of the ability to create and immediately execute a function:

```
let Castle = (function () {
  function Castle(name) {
    this.name = name;
  }
  Castle.prototype.Build = function () {
    console.log("Castle built: " + this.name);
  };
  return Castle;
})();
Westros.Structures.Castle = Castle;
```

This code seems to be a bit longer than the previous code sample but I find it easier to follow due to its hierarchical nature. We can create a new castle using them in the same structure as shown in the preceding code:

```
let winterfell = new Westeros.Structures.Castle("Winterfell");
winterfell.Build();
```

Inheritance using this structure is also relatively easily done. If we were to define a `BaseStructure` class which was to be in the ancestor of all structures, then making use of it would look like this:

```
let BaseStructure = (function () {
  function BaseStructure() {
  }
  return BaseStructure;
})();
Structures.BaseStructure = BaseStructure;
```

```
let Castle = (function (_super) {
  __extends(Castle, _super);
  function Castle(name) {
    this.name = name;
    _super.call(this);
  }
  Castle.prototype.Build = function () {
    console.log("Castle built: " + this.name);
  };
  return Castle;
})(BaseStructure);
```

You'll note that the base structure is passed into the `Castle` object when the closure is evaluated. The highlighted line of code makes use of a helper method called `__extends`. This method is responsible for copying the functions over from the base prototype to the derived class. This particular piece of code was generated from a TypeScript compiler which also, helpfully, generated an `extends` method which looks like:

```
let __extends = this.__extends || function (d, b) {
  for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p];
  function __() { this.constructor = d; }
  __.prototype = b.prototype;
  d.prototype = new __();
};
```

We can continue the rather nifty closure syntax we've adopted for a class to implement an entire module. This is shown here:

```
var Westeros;
(function (Westeros) {
  (function (Structures) {
    let Castle = (function () {
      function Castle(name) {
        this.name = name;
      }
       Castle.prototype.Build = function () {
         console.log("Castle built " + this.name);
       };
       return Castle;
     })();
     Structures.Castle = Castle;
  })(Westeros.Structures || (Westeros.Structures = {}));
  var Structures = Westeros.Structures;
})(Westeros || (Westeros = {}));
```

Within this structure you can see the same code for creating modules that we explored earlier. It is also relatively easy to define multiple classes inside a single module. This can be seen in this code:

```
var Westeros;
(function (Westeros) {
  (function (Structures) {
    let Castle = (function () {
      function Castle(name) {
        this.name = name;
      }
      Castle.prototype.Build = function () {
        console.log("Castle built: " + this.name);
        var w = new Wall();
      };
      return Castle;
    })();
    Structures.Castle = Castle;
    var Wall = (function () {
      function Wall() {
        console.log("Wall constructed");
      }
      return Wall;
    })();
    Structures.Wall = Wall;
  })(Westeros.Structures || (Westeros.Structures = {}));
  var Structures = Westeros.Structures;
})(Westeros || (Westeros = {}));
```

The highlighted code creates a second class inside of the module. It is also perfectly permissible to define one class in each file. Because the code checks to get the current value of `Westeros` before blindly reassigning it, we can safely split the module definition across multiple files.

The last line of the highlighted section shows exposing the class outside of the closure. If we want to make private classes that are only available within the module then we only need to exclude that line. This is actually known as the revealing module pattern. We only reveal the classes that need to be globally available. It is a good practice to keep as much functionality out of the global namespace as possible.

ECMAScript 2015 classes and modules

We've seen so far that it is perfectly possible to build classes and even modules in pre ECMAScript -2015 JavaScript. The syntax is, obviously, a bit more involved than in a language such as C# or Java. Fortunately ECMAScript-2015, brings support for some syntactic sugar for making classes:

```
class Castle extends Westeros.Structures.BaseStructure {
  constructor(name, allegience) {
    super(name);
    ...
  }
  Build() {
    ...
    super.Build();
  }
}
```

ECMAScript-2015 also brings a well thought out module system for JavaScript. There's also syntactic sugar for creating modules which looks like this:

```
module 'Westeros' {
  export function Rule(rulerName, house) {
    ...
    return "Long live " + rulerName + " of house " + house;
  }
}
```

As modules can contain functions they can, of course, contain classes. ECMAScript-2015 also defines a module import syntax and support for retrieving modules from remote locations. Importing a module looks like this:

```
import westeros from 'Westeros';
module JSON from 'http://json.org/modules/json2.js';
westeros.Rule("Rob Stark", "Stark");
```

Some of this syntactic sugar is available in any environment which has full ECMAScript-2015 support. At the time of writing, all major browser vendors have very good support for the class portion of ECMAScript-2015 so there is almost no reason not to use it if you don't have to support ancient browsers.

Best practices and troubleshooting

In an ideal world everybody would get to work on greenfield projects where they can put in standards right from the get go. However that isn't the case. Frequently you may find yourself in a situation where you have a bunch of non-modular JavaScript code as part of a legacy system.

In these situations it may be advantageous to simply ignore the non-modular code until there is an actual need to upgrade it. Despite the popularity of JavaScript, much of the tooling for JavaScript is still immature making it difficult to rely on a compiler to find errors introduced by JavaScript refactoring. Automatic refactoring tools are also complicated by the dynamic nature of JavaScript. However, for new code, proper use of modular JavaScript can be very helpful to avoid namespace conflicts and improve testability.

How to arrange JavaScript is an interesting question. From a web perspective I have taken the approach of arranging my JavaScript in line with the web pages. So each page has an associated JavaScript file, which is responsible for the functionality of that page. In addition, components which are common between pages, say a grid control, are placed into a separate file. At compile time all the files are combined into a single JavaScript file. This helps strike a balance between having a small code file with which to work and reducing the number of requests to the server from the browser.

Summary

It has been said that there are only two really hard things in computing science. What those issues are varies depending on who is speaking. Frequently it is some variation of cache invalidation and naming. How to organize your code is a large part of that naming problem.

As a group we seem to have settled quite firmly on the idea of namespaces and classes. As we've seen, there is no direct support for either of these two concepts in JavaScript. However there are myriad ways to work around the problem, some of which actually provide more power than one would get through a traditional namespace/class system.

The primary concern with JavaScript is to avoid polluting the global namespace with a large number of similarly named, unconnected objects. Encapsulating JavaScript into modules is a key step on the road toward writing maintainable and reusable code.

As we move forward we'll see that many of the patterns which are quite complex arrangements of interfaces become far simpler in the land of JavaScript. Prototype-based inheritance, which seems difficult at the outset, is a tremendous tool for aiding in the simplification of design patterns.

3
Creational Patterns

In the last chapter we took a long look at how to fashion a class. In this chapter we'll look at how to create instances of classes. On the surface it seems like a simple concern but how we create instances of a class can be of great importance.

We take great pains in creating our code so that it be as decoupled as possible. Ensuring that classes have minimal dependence on other classes is the key to building a system that can change fluently with the changing needs of those using the software. Allowing classes to be too closely related means that changes ripple through them like, well, ripples.

One ripple isn't a huge problem but, as you throw more and more changes into the mix, the ripples add up and create interference patterns. Soon the once placid surface is an unrecognizable mess of additive and destructive nodes. This same problem occurs in our applications: the changes magnify and interact in unexpected ways. One place where we tend to forget about coupling is in the creation of objects:

```
let Westeros;
(function (Westeros) {
  let Ruler = (function () {
    function Ruler() {
      this.house = new Westeros.Houses.Targaryen();
    }
    return Ruler;
  })();
  Westeros.Ruler = Ruler;
})(Westeros || (Westeros = {}));
```

You can see in this class that the Ruler's house is strongly coupled to the class `Targaryen`. If this were ever to change then this tight coupling would have to change in a great number of places. This chapter discusses a number of patterns, which were originally presented in the gang of four book, *Design Patterns: Elements of Reusable Object-Oriented Software*. The goal of these patterns is to improve the degree of coupling in applications and increase the opportunities for code reuse. The patterns are as follows:

- Abstract factory
- Builder
- Factory method
- Singleton
- Prototype

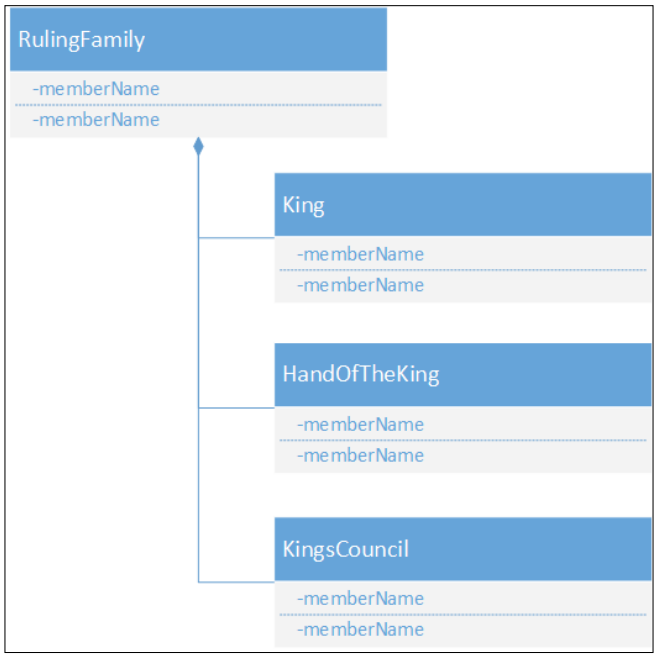
Of course not all of these are applicable to JavaScript, but we'll see all about that as we work through the creational patterns.

Abstract factory

The first pattern presented here is a method for creating kits of objects without knowing the concrete types of the objects. Let's continue with the system presented in the preceding section for ruling a kingdom.

For the kingdom in question the ruling house changes with some degree of frequency. In all likelihood there is a degree of battling and fighting during the change of house but we'll ignore that for the moment. Each house will rule the kingdom differently. Some value peace and tranquility and rule as benevolent leaders, while others rule with an iron fist. The rule of a kingdom is too large for a single individual so the king defers some of his decisions to a second in command known as the hand of the king. The king is also advised on matters by a council, which consists of some of the more savvy lords and ladies of the land.

A diagram of the classes in our description look like this:





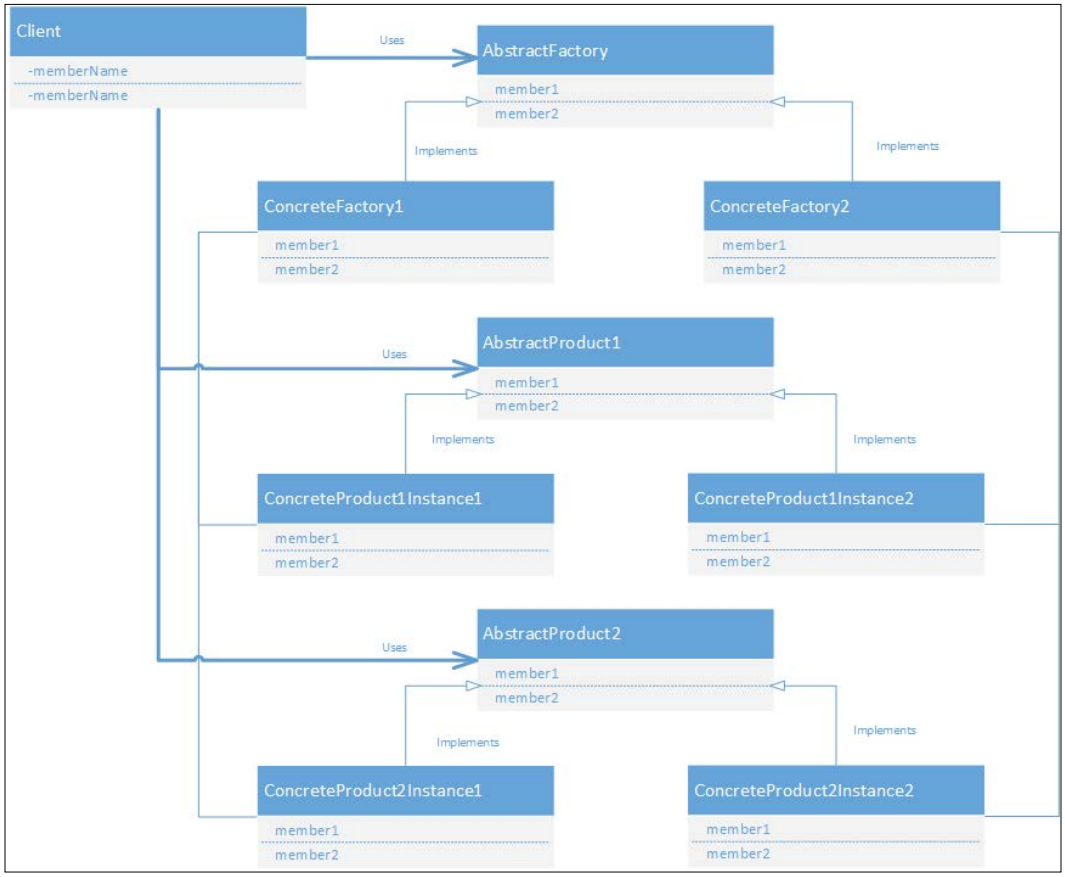
Unified Modeling Language (**UML**) is a standardized language developed by the Object Management Group, which describes computer systems. There is vocabulary in the language for creating user interaction diagrams, sequence diagrams, and state machines, amongst others. For the purposes of this book we're most interested in class diagrams, which describe the relationship between a set of classes.

The entire UML class diagram vocabulary is extensive and is beyond the scope of this book. However, the Wikipedia article available at `https://en.wikipedia.org/wiki/Class_diagram` acts as a great introduction as does Derek Banas' excellent video tutorial on class diagrams available at `https://www.youtube.com/watch?v=3cmzqZzwNDM`.

An issue is that, with the ruling family, and even the member of the ruling family on the throne, changing so frequently, coupling to a concrete family such as Targaryen or Lannister makes our application brittle. Brittle applications do not fare well in an ever-changing world.

An approach to fixing this is to make use of the abstract factory pattern. The abstract factory declares an interface for creating each of the various classes related to the ruling family.

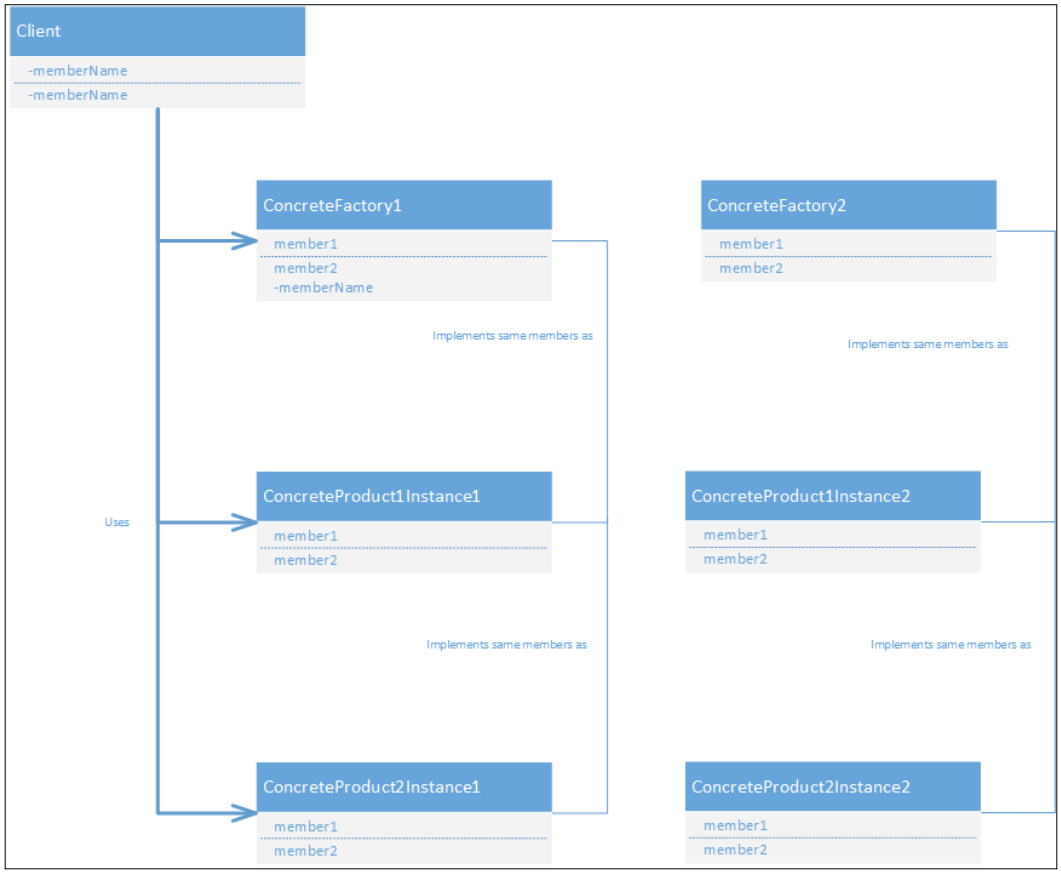
The class diagram of this pattern is rather daunting:



The abstract factory class may have multiple implementations for each of the various ruling families. These are known as concrete factories and each of them will implement the interface provided by the abstract factory. The concrete factories, in return, will return concrete implementations of the various ruling classes. These concrete classes are known as products.

Let's start by looking at the code for the interface for the abstract factory.

No code? Well, actually that is exactly the case. JavaScript's dynamic nature precludes the need for interfaces to describe classes. Instead of having interfaces we'll just create the classes right off the bat:



Instead of interfaces, JavaScript trusts that the class you provide implements all the appropriate methods. At runtime the interpreter will attempt to call the method you request and, if it is found, call it. The interpreter simply assumes that if your class implements the method then it is that class. This is known as **duck typing**.



Duck typing

The name duck typing comes from a 2000 post to the *comp.lang.python* news group by Alex Martelli in which he wrote:

In other words, don't check whether it IS-a duck: check whether it QUACKS-like-a duck, WALKS-like-a duck, and so on, depending on exactly what subset of duck-like behavior you need to play your language-games with.

I enjoy the possibility that Martelli took the term from the witch-hunt sketch from *Monty Python and the Holy Grail*. Although I can find no evidence of that, I think it quite likely as the Python programming language takes its name from Monty Python.

Duck typing is a powerful tool in dynamic languages allowing for much less overhead in implementing a class hierarchy. It does, however, introduce some uncertainty. If two classes implement an identically named method which have radically different meanings then there is no way to know if the one being called is the correct one. Consider for example this code:

```
class Boxer{
  function punch(){}
}
class TicketMachine{
  function punch(){}
}
```

Both classes have a `punch()` method but they clearly have different meanings. The JavaScript interpreter has no idea that they are different classes and will happily call punch on either class, even when one doesn't make sense.

Some dynamic languages support a generic method, which is called whenever an undefined method is called. Ruby, for instance, has `missing_method`, which has proven to be very useful in a number of scenarios. As of writing, there is currently no support for `missing_method` in JavaScript. However, ECMAScript 2016, the follow up to ECMAScript 2015, defines a new construct called `Proxy` which will support dynamically wrapping objects, with this one could implement an equivalent of `missing_method`.

Implementation

To demonstrate an implementation of the Abstract Factory the first thing we'll need is an implementation of the `King` class. This code provides that implementation:

```
let KingJoffery= (function () {
  function KingJoffery() {
```

```
  }
  KingJoffery.prototype.makeDecision = function () {
    …
  };
  KingJoffery.prototype.marry = function () {
    …
  };
  return KingJoffery;
})();
```



This code does not include the module structure suggested in *Chapter 2, Organizing Code*. Including the boiler-plate module code in every example is tedious and you're all smart cookies so you know to put this in modules if you're going to actually use it. The fully modularized code is available in the distributed source code.

This is just a regular concrete class and could really contain any implementation details. We'll also need an implementation of the `HandOfTheKing` class which is equally unexciting:

```
let LordTywin = (function () {
  function LordTywin() {
  }
  LordTywin.prototype.makeDecision = function () {
  };
  return LordTywin;
})();
```

The concrete factory method looks like this:

```
let LannisterFactory = (function () {
  function LannisterFactory() {
  }
  LannisterFactory.prototype.getKing = function () {
    return new KingJoffery();
  };
  LannisterFactory.prototype.getHandOfTheKing = function ()
  {
    return new LordTywin();
  };
  return LannisterFactory;
})();
```

This code simply instantiates new instances of each of the required classes and returns them. An alternative implementation for a different ruling family would follow the same general form and might look like:

```
let TargaryenFactory = (function () {
  function TargaryenFactory() {
  }
  TargaryenFactory.prototype.getKing = function () {
    return new KingAerys();
  };
  TargaryenFactory.prototype.getHandOfTheKing = function () {
    return new LordConnington();
  };
  return TargaryenFactory;
})();
```

The implementation of the Abstract Factory in JavaScript is much easier than in other languages. However the penalty for this is that you lose the compiler checks, which force a full implementation of either the factory or the products. As we proceed through the rest of the patterns, you'll notice that this is a common theme. Patterns that have a great deal of plumbing in statically typed languages are far simpler but create a greater risk of runtime failure. Appropriate unit tests or a JavaScript compiler can ameliorate this situation.

To make use of the Abstract Factory we'll first need a class that requires the use of some ruling family:

```
let CourtSession = (function () {
  function CourtSession(abstractFactory) {
    this.abstractFactory = abstractFactory;
    this.COMPLAINT_THRESHOLD = 10;
  }
  CourtSession.prototype.complaintPresented = function (complaint)
  {
    if (complaint.severity < this.COMPLAINT_THRESHOLD) {
      this.abstractFactory.getHandOfTheKing().makeDecision();
    } else
    this.abstractFactory.getKing().makeDecision();
  };
  return CourtSession;
})();
```

We can now call this `CourtSession` class and inject different functionality depending on which factory we pass in:

```
let courtSession1 = new CourtSession(new TargaryenFactory());
courtSession1.complaintPresented({ severity: 8 });
courtSession1.complaintPresented({ severity: 12 });

let courtSession2 = new CourtSession(new LannisterFactory());
courtSession2.complaintPresented({ severity: 8 });
courtSession2.complaintPresented({ severity: 12 });
```

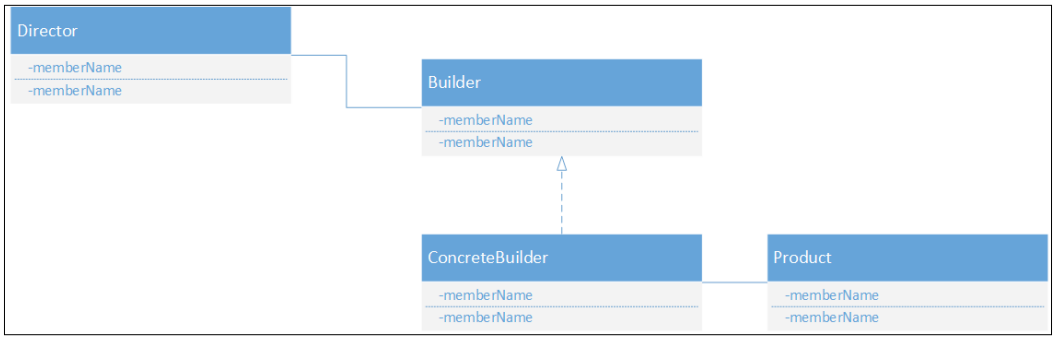
Despite the differences between a static language and JavaScript, this pattern remains applicable and useful in JavaScript applications. Creating a kit of objects, which work together, is useful in a number of situations; any time a group of objects need to collaborate to provide functionality but may need to be replaced wholesale. It may also be a useful pattern when attempting to ensure that a set of objects be used together without substitutions.

Builder

In our fictional world we sometimes have some rather complicated classes, which need to be constructed. The classes contain different implementations of an interface depending on how they are constructed. In order to simplify the building of these classes and encapsulate the knowledge about building the class away from the consumers, a builder may be used. Multiple concrete builders reduce the complexity of the constructor in the implementation. When new builders are required, a constructor does not need to be added, a new builder just needs to be plugged in.

Tournaments are an example of a complicated class. Each tournament has a complicated setup involving the events, the attendees, and the prizes. Much of the setup for these tournaments is similar: each one has a joust, archery, and a melee. Creating a tournament from multiple places in the code means that the responsibility for knowing how to construct a tournament is distributed. If there is a need to change the initiation code then it must be done in a lot of different places.

Employing a builder pattern avoids this issue by centralizing the logic necessary to build the object. Different concrete builders can be plugged into the builder to construct different complicated objects. The relationship between the various classes in the builder pattern is shown here:



Implementation

Let's drop in and look at some of the code. To start with, we'll create a number of utility classes, which will represent the parts of a tournament as shown in the following code:

```
let Event = (function () {
  function Event(name) {
    this.name = name;
  }
  return Event;
})();
Westeros.Event = Event;

let Prize = (function () {
  function Prize(name) {
    this.name = name;
  }
  return Prize;
})();
Westeros.Prize = Prize;

let Attendee = (function () {
  function Attendee(name) {
    this.name = name;
  }
  return Attendee;
})();
Westeros.Attendee = Attendee;
```

The tournament itself is a very simple class as we don't need to assign any of the public properties explicitly:

```
let Tournament = (function () {
  this.Events = [];
  function Tournament() {
  }
  return Tournament;
})();
Westeros.Tournament = Tournament;
```

We'll implement two builders which create different tournaments. This can be seen in the following code:

```
let LannisterTournamentBuilder = (function () {
  function LannisterTournamentBuilder() {
  }
  LannisterTournamentBuilder.prototype.build = function () {
    var tournament = new Tournament();
    tournament.events.push(new Event("Joust"));
    tournament.events.push(new Event("Melee"));
    tournament.attendees.push(new Attendee("Jamie"));
    tournament.prizes.push(new Prize("Gold"));
    tournament.prizes.push(new Prize("More Gold"));
    return tournament;
  };
  return LannisterTournamentBuilder;
})();
Westeros.LannisterTournamentBuilder = LannisterTournamentBuilder;

let BaratheonTournamentBuilder = (function () {
  function BaratheonTournamentBuilder() {
  }
  BaratheonTournamentBuilder.prototype.build = function () {
    let tournament = new Tournament();
    tournament.events.push(new Event("Joust"));
    tournament.events.push(new Event("Melee"));
    tournament.attendees.push(new Attendee("Stannis"));
    tournament.attendees.push(new Attendee("Robert"));
    return tournament;
  };
  return BaratheonTournamentBuilder;
})();
Westeros.BaratheonTournamentBuilder = BaratheonTournamentBuilder;
```

Finally the director, or as we're calling it `TournamentBuilder`, simply takes a builder and executes it:

```
let TournamentBuilder = (function () {
  function TournamentBuilder() {
  }
  TournamentBuilder.prototype.build = function (builder) {
    return builder.build();
  };
  return TournamentBuilder;
})();
Westeros.TournamentBuilder = TournamentBuilder;
```

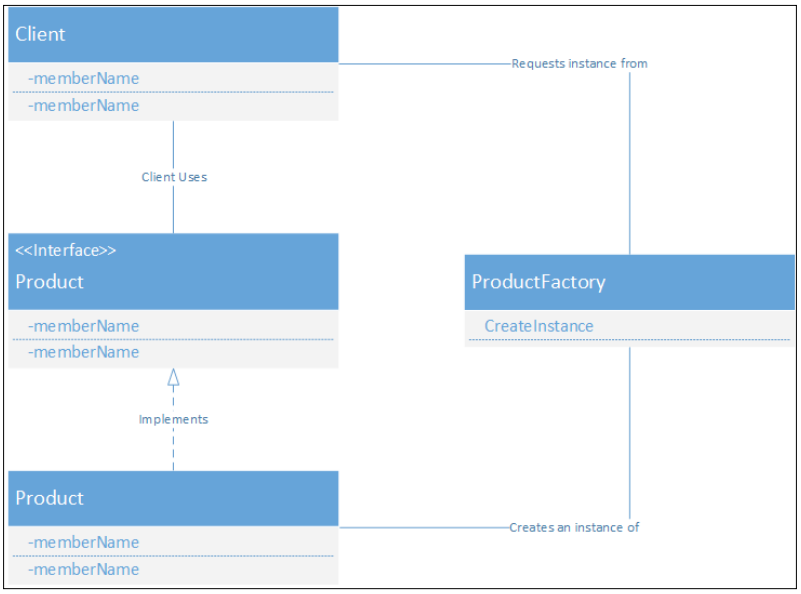
Again you'll see that the JavaScript implementation is far simpler than the traditional implementation due to there being no need for interfaces.

Builders need not return a fully realized object. This means that you can create a builder which partially hydrates an object then allows the object to be passed on to another builder for it to finish. A good real world analogy might be the manufacturing process for a car. Each station along the assembly line builds just a part of the car before passing it onto the next station to build another part. This approach allows for dividing the work of building an object amongst several classes with limited responsibility. In our example above we could have a builder that is responsible for populating the events and another that is responsible for populating the attendees.

Does the builder pattern still make sense in view of JavaScript's prototype extension model? I believe so. There are still cases where a complicated object needs to be created according to different approaches.

Factory method

We've already looked at the Abstract Factory and a builder. The Abstract Factory builds a family of related classes and the builder creates complicated objects using different strategies. The factory method pattern allows a class to request a new instance of an interface without the class making decisions about which implementation of the interface to use. The factory may use some strategy to select which implementation to return:



Sometimes this strategy is simply to take a string parameter or to examine some global setting to act as a switch.

Implementation

In our example world of Westeros there are plenty of times when we would like to defer the choice of implementation to a factory. Just like the real world, Westeros has a vibrant religious culture with dozens of competing religions worshiping a wide variety of gods. When praying in each religion, different rules must be followed. Some religions demand sacrifices while others demand only that a gift be given. The prayer class doesn't want to know about all the different religions and how to construct them.

Let's start with creating a number of different gods to which prayers can be offered. This code creates three gods including a default god to whom prayers fall if no other god is specified:

```
let WateryGod = (function () {
  function WateryGod() {
  }
  WateryGod.prototype.prayTo = function () {
  };
  return WateryGod;
})();
```

```
Religion.WateryGod = WateryGod;
let AncientGods = (function () {
  function AncientGods() {
  }
  AncientGods.prototype.prayTo = function () {
  };
  return AncientGods;
})();
Religion.AncientGods = AncientGods;

let DefaultGod = (function () {
  function DefaultGod() {
  }
  DefaultGod.prototype.prayTo = function () {
  };
  return DefaultGod;
})();
Religion.DefaultGod = DefaultGod;
```

I've avoided any sort of implementation details for each god. You may imagine whatever traditions you want to populate the `prayTo` methods. There is also no need to ensure that each of the gods implements an `IGod` interface. Next we'll need a factory, which is responsible for constructing each of the different gods:

```
let GodFactory = (function () {
  function GodFactory() {
  }
  GodFactory.Build = function (godName) {
    if (godName === "watery")
      return new WateryGod();
    if (godName === "ancient")
      return new AncientGods();
    return new DefaultGod();
  };
  return GodFactory;
})();
```

You can see that in this example we're taking in a simple string to decide how to create a god. It could be done via a global or via a more complicated object. In some polytheistic religions in Westeros, gods have defined roles as gods of courage, beauty, or some other aspect. The god to which one must pray is determined by not just the religion but the purpose of the prayer. We can represent this with a `GodDeterminant` class as is shown here:

```
let GodDeterminant = (function () {
  function GodDeterminant(religionName, prayerPurpose) {
```

```
    this.religionName = religionName;
    this.prayerPurpose = prayerPurpose;
  }
  return GodDeterminant;
})();
```

The factory would be updated to take this class instead of the simple string.

Finally, the last step is to see how this factory would be used. It is quite simple, we just need to pass in a string that denotes which religion we wish to observe and the factory will construct the correct god and return it. This code demonstrates how to call the factory:

```
let Prayer = (function () {
  function Prayer() {
  }
  Prayer.prototype.pray = function (godName) {
  GodFactory.Build(godName).prayTo();
  };
  return Prayer;
})();
```

Once again there is certainly need for a pattern such as this in JavaScript. There are plenty of times where separating the instantiation from the use is useful. Testing the instantiation is also very simple thanks to the separation of concerns and the ability to inject a fake factory to allow testing of `Prayer` is also easy.

Continuing the trend of creating simpler patterns without interfaces, we can ignore the interface portion of the pattern and work directly with the types, thanks to duck typing.

Factory Method is a very useful pattern: it allows classes to defer the selection of the implementation of an instantiation to another class. This pattern is very useful when there are multiple similar implementations such as the strategy pattern (see *Chapter 5*, *Behavioral Patterns*) and is commonly used in conjunction with the Abstract Factory pattern. The Factory Method is used to build the concrete objects within a concrete implementation of the abstract factory. An Abstract Factory pattern may contain a number of Factory Methods. Factory Method is certainly a pattern that remains applicable in the land of JavaScript.

Singleton

The Singleton pattern is perhaps the most overused pattern. It is also a pattern that has fallen out of favor in recent years. To see why people are starting to advise against using Singleton let's take a look at how the pattern works.

Singleton is used when a global variable is desirable, but Singleton provides protection against accidentally creating multiple copies of complex objects. It also allows for the deferral of object instantiation until the first use.

The UML diagram for Singleton looks like the following:



It is clearly a very simple pattern. The Singleton acts as a wrapper around an instance of the class and the singleton itself lives as a global variable. When accessing the instance we simply ask the Singleton for the current instance of the wrapped class. If the class does not yet exist within the Singleton it is common to create a new instance at that time.

Implementation

Within our ongoing example in the world of Westeros, we need to find a case where there can only ever be one of something. Unfortunately, it is a land with frequent conflicts and rivalries, and so my first idea of using the king as the Singleton is simply not going to fly. This split also means that we cannot make use of any of the other obvious candidates (capital city, queen, general, and so on). However, in the far north of Westeros there is a giant wall constructed to keep an ancient enemy at bay. There is only one of these walls and it should pose no issue having it in the global scope.

Let's go ahead and create a singleton in JavaScript:

```
let Westros;
(function (Westeros) {
  var Wall = (function () {
    function Wall() {
      this.height = 0;
      if (Wall._instance)
        return Wall._instance;
      Wall._instance = this;
    }
```

```
    Wall.prototype.setHeight = function (height) {
      this.height = height;
    };
    Wall.prototype.getStatus = function () {
      console.log("Wall is " + this.height + " meters tall");
    };
    Wall.getInstance = function () {
      if (!Wall._instance) {
        Wall._instance = new Wall();
      }
      return Wall._instance;
    };
    Wall._instance = null;
    return Wall;
  })();
  Westeros.Wall = Wall;
})(Westeros || (Westeros = {}));
```

The code creates a lightweight representation of the Wall. The Singleton is demonstrated in the two highlighted sections. In a language like C# or Java we would normally just set the constructor to be private so that it could only be called by the static method `getInstance`. However, we don't have that ability in JavaScript: constructors cannot be private. Thus we do the best we can and return the current instance from the constructor. This may appear strange but in the way we've constructed our classes the constructor is no different from any other method so it is possible to return something from it.

In the second highlighted section we set a static variable, `_instance`, to be a new instance of the Wall when one is not already there. If that `_instance` already exists, we return that. In C# and Java, there would need to be some complicated locking logic in this function to avoid race conditions as two different threads attempted to access the instance at the same time. Fortunately, there is no need to worry about this in JavaScript where the multi-threading story is different.

Disadvantages

Singletons have gained something of a bad reputation in the last few years. They are, in effect, glorified global variables. As we've discussed, global variables are ill conceived and the potential cause of numerous bugs. They are also difficult to test with unit tests as the creation of the instance cannot easily be overridden and any form of parallelism in the test runner can introduce difficult-to-diagnose race conditions. The single largest concern I have with them is that singletons have too much responsibility. They control not just themselves but also their instantiation. This is a clear violation of the single responsibility principle. Almost every problem that can be solved by using a Singleton is better solved using some other mechanism.

JavaScript makes the problem even worse. It isn't possible to create a clean implementation of the Singleton due to the restrictions on the constructor. This, coupled with the general problems around the Singleton, lead me to suggest that the Singleton pattern should be avoided in JavaScript.

Prototype

The final creational pattern in this chapter is the Prototype pattern. Perhaps this name sounds familiar. It certainly should: it is the mechanism through which JavaScript inheritance is supported.

We looked at prototypes for inheritance but the applicability of prototypes need not be limited to inheritance. Copying existing objects can be a very useful pattern. There are numerous cases where being able to duplicate a constructed object is handy. For instance, maintaining a history of the state of an object is easily done by saving previous instances created by leveraging some sort of cloning.

Implementation

In Westeros, we find that members of a family are frequently very similar; as the adage goes: "like father, like son". As each generation is born it is easier to create the new generation through copying and modifying an existing family member than to build one from scratch.

In *Chapter 2*, *Organizing Code*, we looked at how to copy existing objects and presented a very simple piece of code for cloning:

```
function clone(source, destination) {
  for(var attr in source.prototype){
    destination.prototype[attr] = source.prototype[attr];}
}
```

This code can easily be altered to be used inside a class to return a copy of itself:

```
var Westeros;
(function (Westeros) {
  (function (Families) {
    var Lannister = (function () {
      function Lannister() {
      }
      Lannister.prototype.clone = function () {
        var clone = new Lannister();
        for (var attr in this) {
          clone[attr] = this[attr];
```

```
        }
        return clone;
      };
      return Lannister;
    })();
    Families.Lannister = Lannister;
  })(Westeros.Families || (Westeros.Families = {}));
  var Families = Westeros.Families;
})(Westeros || (Westeros = {}));
```

The highlighted section of code is the modified clone method. It can be used as such:

```
let jamie = new Westeros.Families.Lannister();
jamie.swordSkills = 9;
jamie.charm = 6;
jamie.wealth = 10;

let tyrion = jamie.clone();
tyrion.charm = 10;
//tyrion.wealth == 10
//tyrion.swordSkill == 9
```

The Prototype pattern allows for a complex object to be constructed only once and then cloned into any number of objects that vary only slightly. If the source object is not complicated there is little to be gained from taking a cloning approach. Care must be taken when using the prototype approach to think about dependent objects. Should the clone be a deep one?

Prototype is obviously a useful pattern and one that forms an integral part of JavaScript from the get go. As such it is certainly a pattern that will see some use in any JavaScript application of appreciable size.

Tips and tricks

Creational patterns allow for specialized behavior in creating objects. In many cases, such as the factory, they provide extension points into which crosscutting logic can be placed. That is to say logic that applies to a number of different types of objects. If you're looking for a way to inject, say, logging throughout your application, then being able to hook into a factory is of great utility.

For all the utility of these creational patterns they should not be used very frequently. The vast majority of your object instantiations should still be just the normal method of improving the objects. Although it is tempting to treat everything as a nail when you've got a new hammer, the truth is that each situation needs to have a specific strategy. All these patterns are more complicated than simply using `new` and complicated code is more liable to have bugs than simple code. Use `new` whenever possible.

Summary

This chapter presented a number of different strategies for creating objects. These methods provide abstractions over the top of typical methods for creating objects. The Abstract Factory provides a method for building interchangeable kits or collections of related objects. The Builder pattern provides a solution to telescoping parameters issues. It makes the construction of large complicated objects easier. The Factory Method, which is a useful complement to Abstract Factory, allows different implementations to be created though a static factory. Singleton is a pattern for providing a single copy of a class that is available to the entire solution. It is the only pattern we've seen so far which has presented some questions around applicability in modern software. The Prototype pattern is a commonly used pattern in JavaScript for building objects based on other existing objects.

We'll continue our examination of classical design patterns in the next chapter by looking at structural patterns.

4
Structural Patterns

In the previous chapter, we looked at a number of ways to create objects in order to optimize for reuse. In this chapter, we'll take a look at structural patterns; these are patterns that are concerned with easing the design by describing simple ways in which objects can interact.

Again, we will limit ourselves to the patterns described in the GoF book. There are a number of other interesting structural patterns that have been identified since the publication of the GoF and we'll look at those in part 2 of the book.

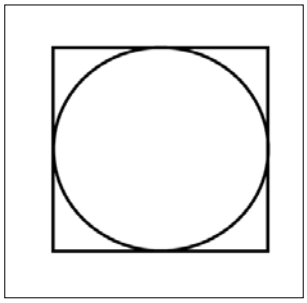
The patterns we'll examine here are:

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

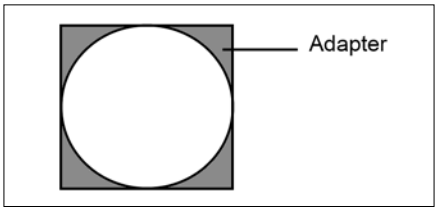
Once again, we'll discuss whether the patterns that were described years ago are still relevant for a different language and a different time.

Adapter

From time to time there is a need to fit a round peg in a square hole. If you've ever played with a child's shape sorting toy then you may have discovered that you can, in fact, put a round peg in a square hole. The hole is not completely filled and getting the peg in there can be difficult:

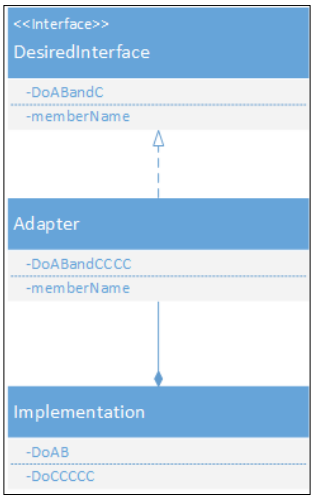


To improve the fit of the peg an adapter can be used. This adapter fills the hole in completely resulting in a perfect fit:



In software a similar approach is often needed. We may need to make use of a class that does not perfectly fit the required interface. The class may be missing methods or may have additional methods we would like to hide. This occurs frequently when dealing with third party code. In order to make it comply with the interface needed in your code, an adapter may be required.

The class diagram for an adapter is very simple as can be seen here:



The interface of the implementation does not look the way we would like it to for use in our code. Normally the solution to this is to simply refactor the implementation so it looks the way we would like it to. However, there are a number of possible reasons that cannot be done. Perhaps the implementation exists inside third party code to which we have no access. It is also possible that the implementation is used elsewhere in the application where the interface is exactly as we would like it to be.

The adapter class is a thin piece of code that implements the required interface. It typically wraps a private copy of the implementation class and proxy calls through to it. The adapter pattern is frequently used to change the abstraction level of the code. Let's take a look at a quick example.

Implementation

In the land of Westeros, much of the trade and travel is done by boat. It is not only more dangerous to travel by ship than to walk or travel by horse, but also riskier due to the constant presence of storms and pirates. These ships are not the sort which might be used by Royal Caribbean to cruise around the Caribbean; they are crude things which might look more at home captained by 15th century European explorers.

While I am aware that ships exist, I have very little knowledge of how they work or how I might go about navigating one. I imagine that many people are in the same (*cough!*) boat as me. If we look at the interface for a Ship in Westeros, it looks intimidating:

```
interface Ship{
  SetRudderAngleTo(angle: number);
  SetSailConfiguration(configuration: SailConfiguration);
  SetSailAngle(sailId: number, sailAngle: number);
  GetCurrentBearing(): number;
  GetCurrentSpeedEstimate(): number;
  ShiftCrewWeightTo(weightToShift: number, locationId: number);
}
```

I would really like a much simpler interface that abstracts away all the fiddly little details. Ideally something like the following:

```
interface SimpleShip{
  TurnLeft();
  TurnRight();
  GoForward();
}
```

This looks like something I could probably figure out even living in a city that is over 1000 kilometers from the nearest ocean. In short, what I'm looking for is a higher-level abstraction around the Ship. In order to transform a Ship into a SimpleShip we need an adapter.

The adapter will have the interface of SimpleShip but it will perform actions on a wrapped instance of Ship. The code might look something like this:

```
let ShipAdapter = (function () {
  function ShipAdapter() {
    this._ship = new Ship();
  }
  ShipAdapter.prototype.TurnLeft = function () {
    this._ship.SetRudderAngleTo(-30);
    this._ship.SetSailAngle(3, 12);
  };
  ShipAdapter.prototype.TurnRight = function () {
    this._ship.SetRudderAngleTo(30);
    this._ship.SetSailAngle(5, -9);
  };
  ShipAdapter.prototype.GoForward = function () {
```

```
    //do something else to the _ship
  };
  return ShipAdapter;
})();
```

In reality these functions would be far more complex, but it should not matter much because we've got a nice simple interface to present to the world. The presented interface can also be set up so as to restrict access to certain methods on the underlying type. When building library code, adapters can be used to mask the internal method and only present the limited functions needed to the end user.

To use this pattern, the code might look like:

```
var ship = new ShipAdapter();
ship.GoForward();
ship.TurnLeft();
```

You would likely not want to use adapter in the name of your client class as it leaks some information about the underlying implementation. Clients should be unaware they are talking to an adapter.

The adapter itself can grow to be quite complex to adjust one interface to another. In order to avoid creating very complex adapters, care must be taken. It is certainly not inconceivable to build several adapters, one atop another. If you find an adapter becoming too large then it is a good idea to stop and examine if the adapter is following the single responsibility principle. That is to say, ensure that each class has only one thing for which it has some responsibility. A class that looks up users from a database should not also contain functionality for sending e-mails to these users. That is too much responsibility. Complex adapters can be replaced with a composite object, which will be explored later in this chapter.

From the testing perspective, adapters can be used to totally wrap third party dependencies. In this scenario they provide a place into which to hook tests. Unit tests should avoid testing libraries but they can certainly test the adapters to ensure that they are proxying through the correct calls.

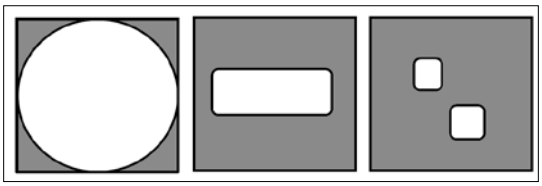
The adapter is a very powerful pattern for simplifying code interfaces. Massaging interfaces to better match a requirement is useful in countless places. The pattern is certainly useful in JavaScript. Applications written in JavaScript tend to make use of a large number of small libraries. By wrapping up these libraries in adapters I'm able to limit the number of places I interact with the libraries directly; this means that the libraries can easily be replaced.

The adapter pattern can be slightly modified to provide consistent interfaces over a number of different implementations. This is usually known as the bridge pattern.

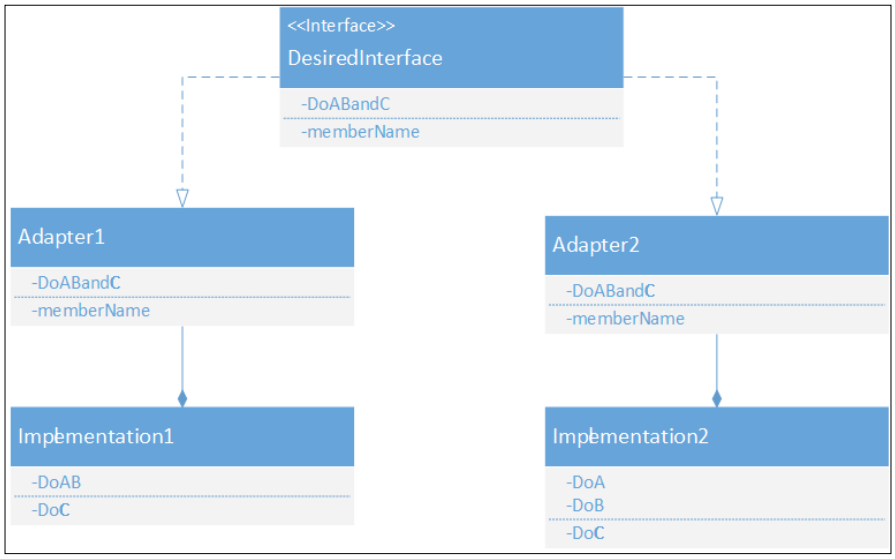
Bridge

The bridge pattern takes the adapter pattern to a new level. Given an interface, we can build multiple adapters, each one of which acts as an intermediary to a different implementation.

An excellent example that I've run across, is dealing with two different services that provide more or less the same functionality and are used in a failover configuration. Neither service provides exactly the interface required by the application and both services provide different APIs. In order to simplify the code, adapters are written to provide a consistent interface. The adapters implement a consistent interface and provide fills so that each API can be called consistently. To expand on the shape sorter metaphor a bit more, we can imagine that we have a variety of different pegs we would like to use to fill the square hole. Each adapter fills in the missing bits and helps us get a good fit:



The bridge is a very useful pattern. Let's take a look at how to implement it:



The adapters shown in the preceding diagram sit between the implementation and the desired interface. They modify the implementation to fit in with the desired interface.

Implementation

We've already discussed that in the land of Westeros the people practice a number of disparate religions. Each one has a different way of praying and making offerings. There is a lot of complexity around making the correct prayers at the correct time and we would like to avoid exposing this complexity. Instead we'll write a series of adapters that can simplify prayers.

The first thing we need is a number of different gods to which we can pray:

```
class OldGods {
  prayTo(sacrifice) {
    console.log("We Old Gods hear your prayer");
  }
}
Religion.OldGods = OldGods;
class DrownedGod {
  prayTo(humanSacrifice) {
    console.log("*BUBBLE* GURGLE");
  }
}
Religion.DrownedGod = DrownedGod;
class SevenGods {
  prayTo(prayerPurpose) {
    console.log("Sorry there are a lot of us, it gets confusing
      here. Did you pray for something?");
  }
}
Religion.SevenGods = SevenGods;
```

These classes should look familiar as they are basically the same classes found in the previous chapter where they were used as examples for the factory method. You may notice, however, that the signature for the `prayTo` method for each religion is slightly different. This proves to be something of an issue when building a consistent interface like the one shown in pseudo code here:

```
interface God
{
  prayTo():void;
}
```

So let's slot in a few adapters to act as a bridge between the classes we have and the signature we would like the following:

```
class OldGodsAdapter {
  constructor() {
    this._oldGods = new OldGods();
  }
  prayTo() {
    let sacrifice = new Sacrifice();
    this._oldGods.prayTo(sacrifice);
  }
}
Religion.OldGodsAdapter = OldGodsAdapter;
class DrownedGodAdapter {
  constructor() {
    this._drownedGod = new DrownedGod();
  }
  prayTo() {
    let sacrifice = new HumanSacrifice();
    this._drownedGod.prayTo(sacrifice);
  }
}
Religion.DrownedGodAdapter = DrownedGodAdapter;
class SevenGodsAdapter {
  constructor() {
    this.prayerPurposeProvider = new PrayerPurposeProvider();
    this._sevenGods = new SevenGods();
  }
  prayTo() {
    this._sevenGods.prayTo(this.prayerPurposeProvider.GetPurpose());
  }
}
Religion.SevenGodsAdapter = SevenGodsAdapter;
class PrayerPurposeProvider {
  GetPurpose() { }
  }
Religion.PrayerPurposeProvider = PrayerPurposeProvider;
```

Each one of these adapters implements the `God` interface we wanted and abstracts away the complexity of dealing with three different interfaces, one for each god:

To use the Bridge pattern, we could write code like so:

```
let god1 = new Religion.SevenGodsAdapter();
let god2 = new Religion.DrownedGodAdapter();
```

```
let god3 = new Religion.OldGodsAdapter();

let gods = [god1, god2, god3];
for(let i =0; i<gods.length; i++){
  gods[i].praryTo();
}
```

This code uses the bridges to provide a consistent interface to the gods such that they can all be treated as equals.

In this case we are simply wrapping the individual gods and proxying method calls through to them. The adapters could each wrap a number of objects and this is another useful place in which to use the adapter. If a complex series of objects needs to be orchestrated, then an adapter can take some responsibility for that orchestration providing a simpler interface to other classes.

You can imagine how useful the bridge pattern is. It can be used well in conjunction with the factory method pattern presented in the previous chapter.

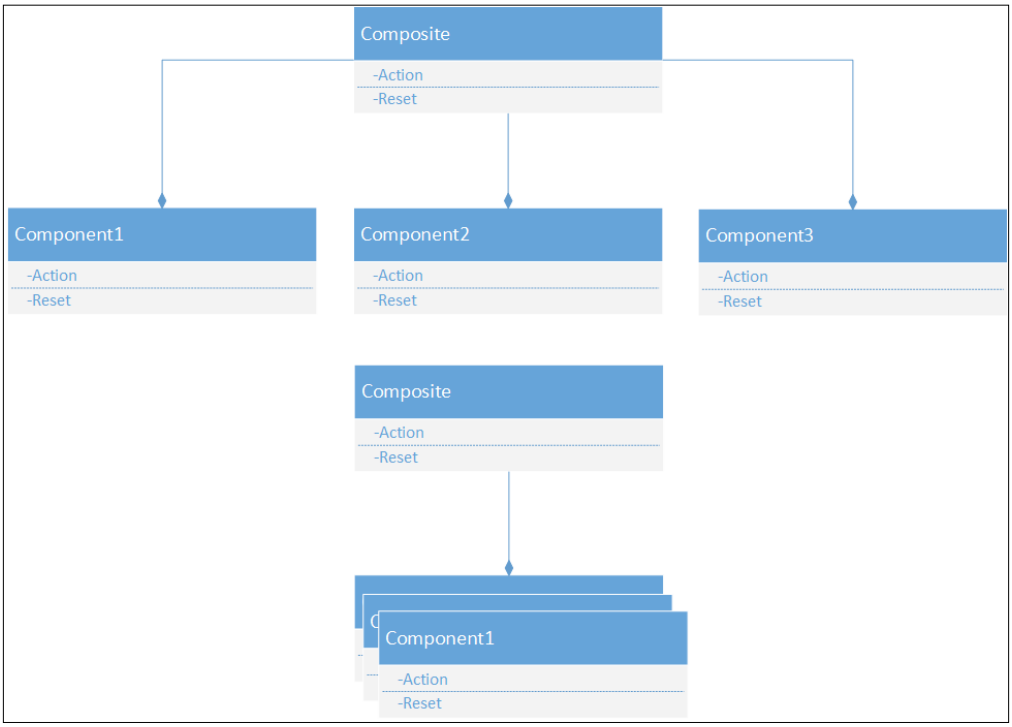
This pattern certainly remains a very useful one for use in JavaScript. As I mentioned at the start of this section, it is handy for dealing with different APIs in a consistent fashion. I have used it for swapping in different third party components such as different graphing libraries or phone system integration points. If you're building applications on a mobile platform using JavaScript, then the bridge pattern is going to be a great friend for you, allowing you to separate your common and platform specific code cleanly. Because there are no interfaces in JavaScript, the bridge pattern is far closer to the adapter in JavaScript than in other languages. In fact, it is basically exactly the same.

A bridge also makes testing easier. We are able to implement a fake bridge and use this to ensure that the calls into the bridge are correct.

Composite

In the previous chapter I mentioned that we would like to avoid coupling our objects together tightly. Inheritance is a very strong form of coupling and I suggested that, instead, composites should be used. The composite pattern is a special case of this in which the composite is treated as interchangeable with the components. Let's explore how the composite pattern works.

The following class diagram contains two different ways to build a composite component:



In the first one, the composite component is built from a fixed number of a variety of components. The second component is constructed from a collection of indeterminate length. In both cases the components contained within the parent composition could be of the same type as the composition. So a composition may contain instances of its own type.

The key feature of the composite pattern is the interchangeability of a component with its children. So, if we have a composite which implements `IComponent`, then all of the components of the composite will also implement `IComponent`. This is, perhaps, best illustrated with an example.

Example

Tree structures are very useful in computing. It turns out that a hierarchical tree can represent many things. A tree is made up of a series of nodes and edges and is a cyclical. In a binary tree, each node contains a left and right child until we get down to the terminal nodes known as leaves.

While life is difficult in Westeros there is an opportunity for taking joy in things like religious holidays or weddings. At these events there is typically a great deal of feasting on delicious foods. The recipes for these foods is much as you would find in your own set of recipes. A simple dish like baked apples contains a list of ingredients:

- Baking apple
- Honey
- Butter
- Nuts

Each one of these ingredients implements an interface which we'll refer to as `IIngredient`. More complex recipes contain more ingredients, but in addition to that, more complex recipes may contain complex ingredients that are themselves made from other ingredients.

A popular dish in a southern part of Westeros is a dessert which is not at all unlike what we would call tiramisu. It is a complex recipe with ingredients such as:

- Custard
- Cake
- Whipped cream
- Coffee

Of course custard itself is made from:

- Milk
- Sugar
- Eggs
- Vanilla

Custard is a composite as is coffee and cake.

Operations on the composite object are typically proxied through to all of the contained objects.

Implementation

A simple ingredient, one which would be a leaf node, is shown in this code:

```
class SimpleIngredient {
  constructor(name, calories, ironContent, vitaminCContent) {
    this.name = name;
    this.calories = calories;
```

```
    this.ironContent = ironContent;
    this.vitaminCContent = vitaminCContent;
  }
  GetName() {
    return this.name;
  }
  GetCalories() {
    return this.calories;
  }
  GetIronContent() {
    return this.ironContent;
  }
  GetVitaminCContent() {
    return this.vitaminCContent;
  }
}
```

It can be used interchangeably with a compound ingredient which has a list of ingredients:

```
class CompoundIngredient {
  constructor(name) {
    this.name = name;
    this.ingredients = new Array();
  }
  AddIngredient(ingredient) {
    this.ingredients.push(ingredient);
  }
  GetName() {
    return this.name;
  }
  GetCalories() {
    let total = 0;
    for (let i = 0; i < this.ingredients.length; i++) {
      total += this.ingredients[i].GetCalories();
    }
    return total;
  }
  GetIronContent() {
    let total = 0;
    for (let i = 0; i < this.ingredients.length; i++) {
      total += this.ingredients[i].GetIronContent();
    }
    return total;
  }
```

```
  GetVitaminCContent() {
    let total = 0;
    for (let i = 0; i < this.ingredients.length; i++) {
      total += this.ingredients[i].GetVitaminCContent();
    }
    return total;
  }
}
```

The composite ingredient loops over its internal ingredients and performs the same operation on each of them. There is, of course, no need to define an interface due to the prototype model.

To make use of this compound ingredient we might do:

```
let egg = new SimpleIngredient("Egg", 155, 6, 0);
let milk = new SimpleIngredient("Milk", 42, 0, 0);
let sugar = new SimpleIngredient("Sugar", 387, 0,0);
let rice = new SimpleIngredient("Rice", 370, 8, 0);

let ricePudding = new CompoundIngredient("Rice Pudding");
ricePudding.AddIngredient(egg);
ricePudding.AddIngredient(rice);
ricePudding.AddIngredient(milk);
ricePudding.AddIngredient(sugar);

console.log("A serving of rice pudding contains:");
console.log(ricePudding.GetCalories() + " calories");
```

Of course this only shows part of the power of the pattern. We could use rice pudding as an ingredient in an even more complicated recipe: rice pudding stuffed buns (they have some strange foods in Westeros). As both the simple and compound version of the ingredient have the same interface, the caller does not need to know that there is any difference between the two ingredient types.

Composite is a heavily used pattern in JavaScript code that deals with HTML elements, as they are a tree structure. For example, the jQuery library provides a common interface if you have selected a single element or a collection of elements. When a function is called it is actually called on all the children, for instance:

```
$("a").hide()
```

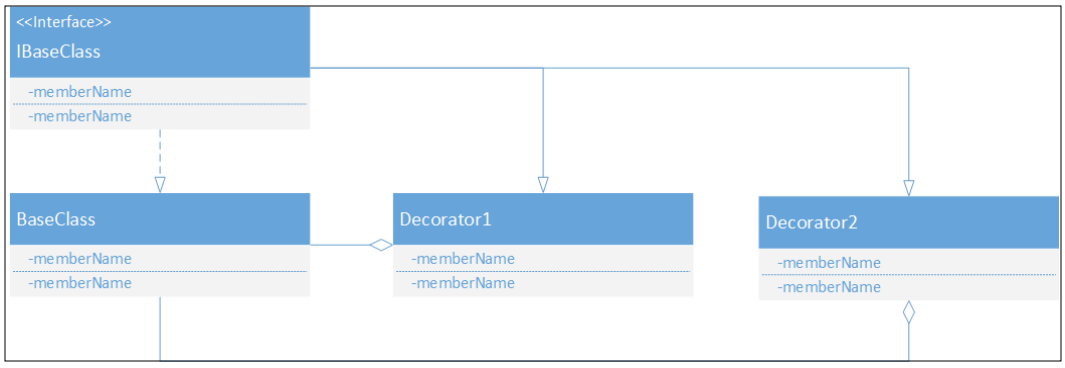
This will hide all the links on a page regardless of how many elements are actually found by calling `$("a")`. The composite is a very useful pattern for JavaScript development.

Decorator

The decorator pattern is used to wrap and augment an existing class. Using a decorator pattern is an alternative to subclassing an existing component. Subclassing is typically a compile time operation and is a tight coupling. This means that once subclassing is performed, there is no way to alter it at runtime. In cases where there are many possible subclassings that can act in combination, the number of combinations of subclassings explodes. Let's look at an example.

The armor worn by knights in Westeros can be quite configurable. Armor can be fabricated in a number of different styles: scale, lamellar, chainmail, and so on. In addition to the style of armor, there is also a variety of different face guards, knee, and elbow joints, and, of course, colors. The behavior of armor made from lamellar and a grille is different from chainmail with a face visor. You can see, however, that there is a large number of possible combinations; far too many combinations to explicitly code.

What we do instead is implement the different styles of armor using the decorator pattern. A decorator works using a similar theory to the adapter and bridge patterns, in that it wraps another instance and proxy calls through. The decorator pattern, however, performs the redirections at runtime by having the instance to wrap passed into it. Typically, a decorator will act as a simple pass through for some methods and for others it will make some modifications. These modifications could be limited to performing an additional action before passing the call off to the wrapped instance or could go so far as to change the parameters passed in. A UML representation of the decorator pattern looks like the following diagram:



This allows for very granular control over which methods are altered by the decorator and which remain as mere pass-through. Let's take a look at an implementation of the pattern in JavaScript.

Implementation

In this code we have a base class, `BasicArmor`, and it is then decorated by the `ChainMail` class:

```
class BasicArmor {
  CalculateDamageFromHit(hit) {
    return hit.Strength * .2;
  }
  GetArmorIntegrity() {
    return 1;
  }
}

class ChainMail {
  constructor(decoratedArmor) {
    this.decoratedArmor = decoratedArmor;
  }
  CalculateDamageFromHit(hit) {
    hit.Strength = hit.Strength * .8;
    return this.decoratedArmor.CalculateDamageFromHit(hit);
  }
  GetArmorIntegrity() {
    return .9 * this.decoratedArmor.GetArmorIntegrity();
  }
}
```

The `ChainMail` armor takes in an instance of armor that complies with an interface, such as:

```
export interface IArmor{
  CalculateDamageFromHit(hit: Hit):number;
  GetArmorIntegrity():number;
}
```

That instance is wrapped and calls proxied through. The method `GetArmorIntegiry` modifies the result from the underlying class while `CalculateDamageFromHit` modifies the arguments that are passed into the decorated class. This `ChainMail` class could, itself, be decorated with several more layers of decorators until a long chain of methods is actually called for each method call. This behavior, of course, remains invisible to outside callers.

To make use of this armor decorator, look at the following code:

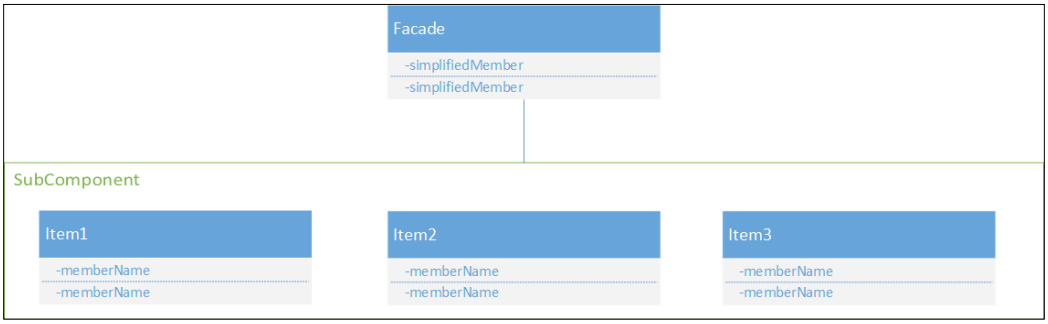
```
let armor = new ChainMail(new Westeros.Armor.BasicArmor());
console.log(armor.CalculateDamageFromHit({Location: "head", Weapon:
  "Sock filled with pennies", Strength: 12}));
```

It is tempting to make use of JavaScript's ability to rewrite individual methods on classes to implement this pattern. Indeed, in an earlier draft of this section I had intended to suggest just that. However, doing so is syntactically messy and not a common way of doing things. One of the most important things to keep in mind when programming is that code must be maintainable, not only by you but also by others. Complexity breeds confusion and confusion breeds bugs.

The decorator pattern is a valuable pattern for scenarios where inheritance is too limiting. These scenarios still exist in JavaScript, so the pattern remains useful.

Façade

The façade pattern is a special case of the Adapter pattern that provides a simplified interface over a collection of classes. I mentioned such a scenario in the section on the adapter pattern but only within the context of a single class, `SimpleShip`. This same idea can be expanded to provide an abstraction around a group of classes or an entire subsystem. The façade pattern in UML form looks like the following diagram:



Implementation

If we take the same `SimpleShip` as before and expand it to an entire fleet, we have a great example of a use for creating a façade. If it was difficult to sail a single ship it would be far more difficult to command an entire fleet of ships. There is a great deal of nuance required, commands to individual ships would have to be made. In addition to the individual ships there must also be a fleet Admiral and a degree of coordination between the ships in order to distribute supplies. All of this can be abstracted away. If we have a collection of classes representing the aspects of a fleet such as these:

```
let Ship = (function () {
  function Ship() {
```

```
    }
    Ship.prototype.TurnLeft = function () {
    };
    Ship.prototype.TurnRight = function () {
    };
    Ship.prototype.GoForward = function () {
    };
    return Ship;
})();
Transportation.Ship = Ship;

let Admiral = (function () {
  function Admiral() {
  }
  return Admiral;
})();
Transportation.Admiral = Admiral;

let SupplyCoordinator = (function () {
  function SupplyCoordinator() {
  }
  return SupplyCoordinator;
})();
Transportation.SupplyCoordinator = SupplyCoordinator;
```

Then we might build a façade as follows:

```
let Fleet = (function () {
   function Fleet() {
  }
  Fleet.prototype.setDestination = function (destination) {
    //pass commands to a series of ships, admirals and whoever else
    needs it
  };

  Fleet.prototype.resupply = function () {
  };

  Fleet.prototype.attack = function (destination) {
    //attack a city
  };
  return Fleet;
})();
```

Façades are very useful abstractions, especially in dealing with APIs. Using a façade around a granular API can create an easier interface. The level of abstraction at which the API works can be raised so that it is more in sync with how your application works. For instance, if you're interacting with the Azure blob storage API you could raise the level of abstraction from working with individual files to working with collections of files. Instead of writing the following:

```
$.ajax({method: "PUT",
url: "https://settings.blob.core.windows.net/container/set1",
data: "setting data 1"});

$.ajax({method: "PUT",
url: "https://settings.blob.core.windows.net/container/set2",
data: "setting data 2"});

$.ajax({method: "PUT",
url: "https://settings.blob.core.windows.net/container/set3",
data: "setting data 3"});
```

A façade could be written which encapsulates all of these calls and provides an interface, like:

```
public interface SettingSaver{
  Save(settings: Settings); //preceding code in this method
  Retrieve():Settings;
}
```

As you can see façades remain useful in JavaScript and should be a pattern that remains in your toolbox.

Flyweight

In boxing there is a light weight division between 49-52 kg known as the flyweight division. It was one of the last divisions to be established and was named, I imagine, for the fact that the fighters in it were tiny, like flies.

The flyweight pattern is used in instances when there are a large number of instances of objects which vary only slightly. I should perhaps pause here to mention that a large number, in this situation, is probably in the order of 10,000 objects rather than 50 objects. However, the cutoff for the number of instances is highly dependent on how expensive the object is to create.

In some cases, the object may be so expensive that only a handful are required before they overload the system. In this case introducing flyweight at a smaller number would be beneficial. Maintaining a full object for each object consumes a lot of memory. It seems that the memory is largely consumed wastefully too, as most of the instances have the same value for their fields. Flyweight offers a way to compress this data by only keeping track of the values that differ from some prototype in each instance.

JavaScript's prototype model is ideal for this scenario. We can simply assign the most common value to the prototype and have individual instances override them as needed. Let's see how that looks with an example.

Implementation

Returning once more to Westeros (aren't you glad I've opted for a single overarching problem domain?) we find that armies are full of ill-equipped fighting people. Within this set of people there is really very little difference from the perspective of the generals. Certainly each person has their own life, ambitions, and dreams but they have all been adapted into simple fighting automatons in the eyes of the general. The general is only concerned with how well the soldiers fight, if they're healthy, and if they're well fed. We can see the simple set of fields in this code:

```
let Soldier = (function () {
  function Soldier() {
    this.Health = 10;
    this.FightingAbility = 5;
    this.Hunger = 0;
  }
  return Soldier;
})();
```

Of course, with an army of 10,000 soldiers, keeping track of all of this requires quite some memory. Let's take a different approach and use a class:

```
class Soldier {
  constructor() {
    this.Health = 10;
    this.FightingAbility = 5;
    this.Hunger = 0;
  }
}
```

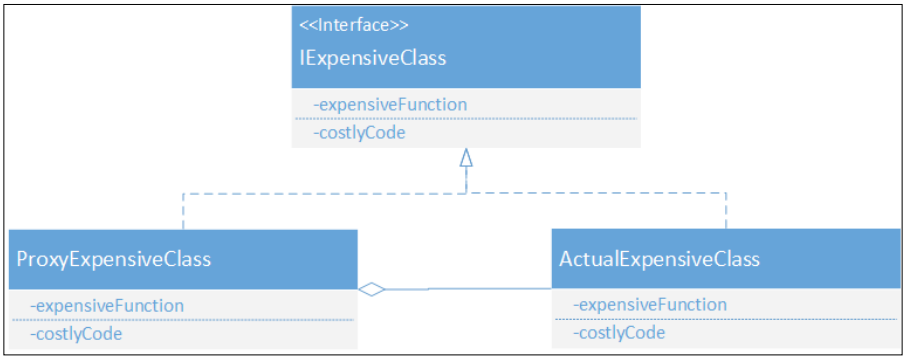
Using this approach, we are able to defer all requests for the soldier's health to the prototype. Setting the value is easy too:

```
let soldier1 = new Soldier();
let soldier2 = new Soldier();
console.log(soldier1.Health); //10
soldier1.Health = 7;
console.log(soldier1.Health); //7
console.log(soldier2.Health); //10
delete soldier1.Health;
console.log(soldier1.Health); //10
```

You'll note that we make a call to delete to remove the property override and return the value back to the parent value.

Proxy

The final pattern presented in this chapter is the proxy. In the previous section I mentioned how it is expensive to create objects and how we would like to avoid creating too many of them. The proxy pattern provides a method of controlling the creation and use of expensive objects. The UML of the proxy pattern looks like the following diagram:



As you can see, the proxy mirrors the interface of the actual instance. It is substituted in for the instance in all the clients and, typically, wraps a private instance of the class. There are a number of places where the proxy pattern can be of use:

- Lazy instantiation of an expensive object
- Protection of secret data
- Stubbing for remote method invocation
- Interposing additional actions before or after method invocation

Often an object is expensive to instantiate and we don't want to have instances created before they're actually used. In this case the proxy can check its internal instance and, if not yet initiated, create it before passing on the method call. This is known as lazy instantiation.

If a class has been designed without any security in mind but now requires some, this can be provided through the use of a proxy. The proxy will check the call and only pass on the method call in cases where the security checks out.

The proxy may be used to simply provide an interface to methods that are invoked somewhere else. In fact, this is exactly how a number of web socket libraries function, proxying calls back to the web server.

Finally, there may be cases where it is useful to interpose some functionality into the method invocation. This could be logging of parameters, validating of parameters, altering results, or any number of things.

Implementation

Let's take a look at a Westeros example where method interposition is needed. As tends to happen, the units of measurement for liquids vary greatly from one side of the land to the other. In the north, one might buy a pint of beer, while in the south, one would buy it by the dragon. This causes no end of confusion and code duplication, but can be solved by wrapping classes that care about measurement in proxies.

For example, this code is for a barrel calculator which estimates the number of barrels needed to ship a quantity of liquid:

```
class BarrelCalculator {
  calculateNumberNeeded(volume) {
    return Math.ceil(volume / 157);
  }
}
```

Although it is not well documented, here this version takes pints as a volume parameter. A proxy is created which deals with the transformation thusly:

```
class DragonBarrelCalculator {
  calculateNumberNeeded(volume) {
    if (this._barrelCalculator == null)
      this._barrelCalculator = new BarrelCalculator();
    return this._barrelCalculator.calculateNumberNeeded(volume *
      .77);
  }
}
```

Equally we might create another proxy for a pint-based barrel calculator:

```
class PintBarrelCalculator {
  calculateNumberNeeded(volume) {
    if (this._barrelCalculator == null)
      this._barrelCalculator = new BarrelCalculator();
    return this._barrelCalculator.calculateNumberNeeded(volume *
      1.2);
  }
}
```

This proxy class does the unit conversion for us and helps alleviate some confusion around units. Some languages, such as F#, support the concept of units of measure. In effect it is a typing system which is overlaid over simple data types such as integers, preventing programmers from making mistakes such as adding a number representing pints to one representing liters. Out of the box in JavaScript there is no such capability. Using a library such as JS-Quantities (`http://gentooboontoo.github.io/js-quantities/`) is an option however. If you look at it, you'll see the syntax is quite painful. This is because JavaScript doesn't permit operator overloading. Having seen how weird adding things such as an empty array to an empty array are (it results in an empty string), I think perhaps we can be thankful that operator overloading isn't supported.

If we wanted to protect against accidentally using the wrong sort of calculator when we have pints and think we have dragons, then we could stop with our primitive obsession and use a type for the quantity, a sort of poor person's units of measure:

```
class PintUnit {
  constructor(unit, quantity) {
    this.quanity = quantity;
  }
}
```

This can then be used as a guard in the proxy:

```
class PintBarrelCalculator {
  calculateNumberNeeded(volume) {
    if(PintUnit.prototype == Object.getPrototypeOf(volume))
      //throw some sort of error or compensate
    if (this._barrelCalculator == null)
      this._barrelCalculator = new BarrelCalculator();
    return this._barrelCalculator.calculateNumberNeeded(volume *
      1.2);
  }
}
```

As you can see, we end up with pretty much what JS-Quantities does but in a more ES6 form.

The proxy is absolutely a useful pattern within JavaScript. I already mentioned that it is used by web socket libraries when generating stubs but it finds itself useful in countless other locations.

Hints and tips

Many of the patterns presented in this chapter provide methods of abstracting functionality and of molding interfaces to look the way you want. Keep in mind that with each layer of abstraction a cost is introduced. Function calls take longer but it is also much more confusing for people who need to understand your code. Tooling can help a little but tracking a function call through nine layers of abstraction is never fun.

Also be wary of doing too much in the façade pattern. It is very easy to turn the façade into a fully-fledged management class and that degrades easily into a God object that is responsible for coordinating and doing everything.

Summary

In this chapter we've looked at a number of patterns used to structure the interaction between objects. Some of them are quite similar to each other but they are all useful in JavaScript, although the bridge is effectively reduced to an adapter. In the next chapter we'll finish our examination of the original GoF patterns by looking at behavioral patterns.

5
Behavioral Patterns

In the last chapter we looked at structural patterns that describe ways in which objects can be constructed to ease interaction.

In this chapter we'll take a look at the final, and largest, grouping of GoF patterns: behavioral patterns. These patterns are ones that provide guidance on how objects share data or, from a different perspective, how data flows between objects.

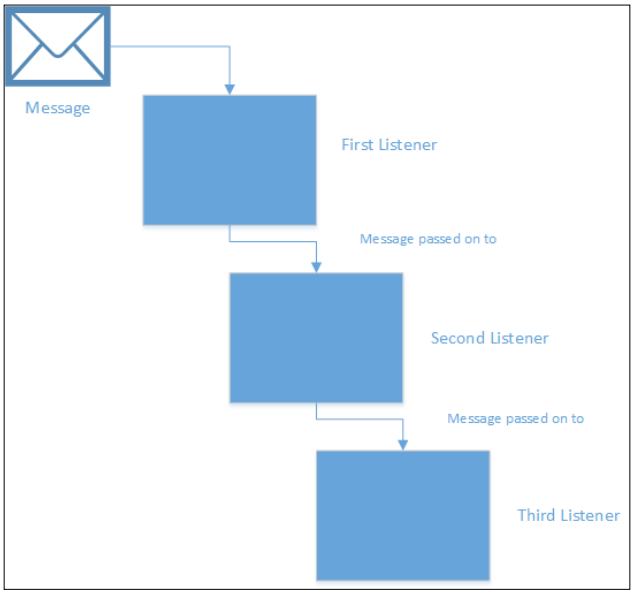
The patterns we'll look at are as follows:

- Chain of responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template method
- Visitor

Once again there are a number of more recently identified patterns that could well be classified as behavioral patterns. We'll defer looking at those until a later chapter, instead keeping to the GoF patterns.

Chain of responsibility

We can think of a function call on an object as sending that object a message. Indeed this message passing mentality is one that dates back to the days of Smalltalk. The chain of responsibility pattern describes an approach in which a message tickles down from one class to another. A class can either act on the message or allow it to be passed on to the next member of the chain. Depending on the implementation there are a few different rules that can be applied to the message passing. In some situations only the first matching link in the chain is permitted to act. In others, every matching link acts on the message. Sometimes the links are permitted to stop processing or even to mutate the message as it continues down the chain:

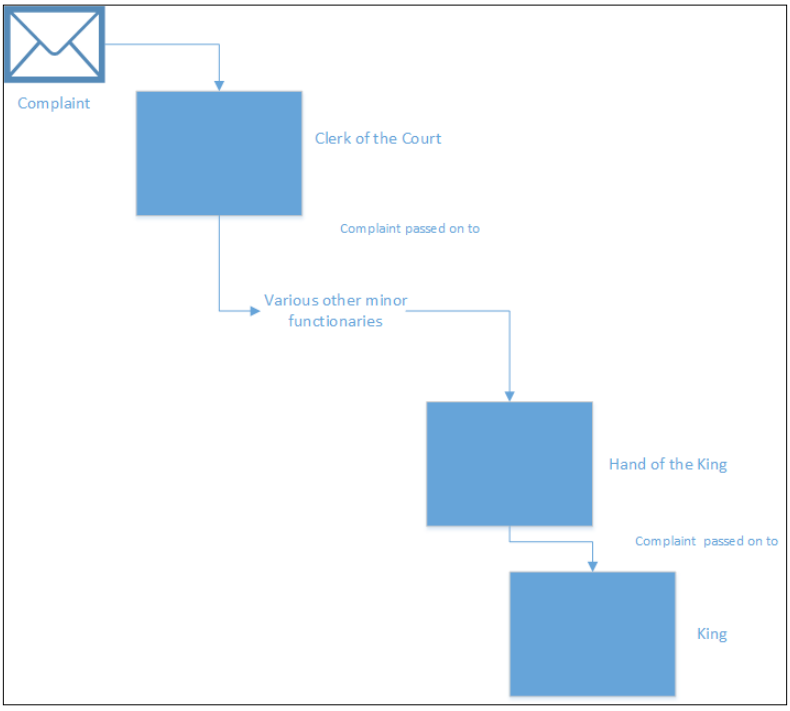


Let's see if we can find a good example of this pattern in our go-to example: the land of Westeros.

Implementation

There is very little in the way of a legal system in Westeros. Certainly there are laws and even city guards who enforce them but the judicial system is scant. The law of the land is really decided by the king and his advisors. Those with the time and money can petition for an audience with the king who will listen to their complaint and pass a ruling. This ruling is law. Of course any king who spent his entire day listening to the complaints of peasants would go mad. For this reason many of the cases are caught and solved by his advisors before they reach his ears.

To represent this in code we'll need to start by thinking about how the chain of responsibility would work. A complaint comes in and it starts with the lowest possible person who can solve it. If that person cannot or will not solve the problem it tickles up to a more senior member of the ruling class. Eventually the problem reaches the king who is the final arbiter of disputes. We can think of him as the default dispute solver who is called upon when all else fails. The chain of responsibility is visible in the following diagram:



We'll start with an interface to describe those who might listen to complaints:

```
export interface ComplaintListener{
  IsAbleToResolveComplaint(complaint: Complaint): boolean;
  ListenToComplaint(complaint: Complaint): string;
}
```

The interface requires two methods. The first is a simple check to see if the class is able to resolve a given complaint. The second listens to and resolves the complaint. Next we'll need to describe what constitutes a complaint:

```
var Complaint = (function () {
  function Complaint() {
    this.ComplainingParty = "";
```

```
      this.ComplaintAbout = "";
      this.Complaint = "";
    }
    return Complaint;
  })();
```

Next we need a couple of different classes which implement `ComplaintListener` and are able to solve complaints:

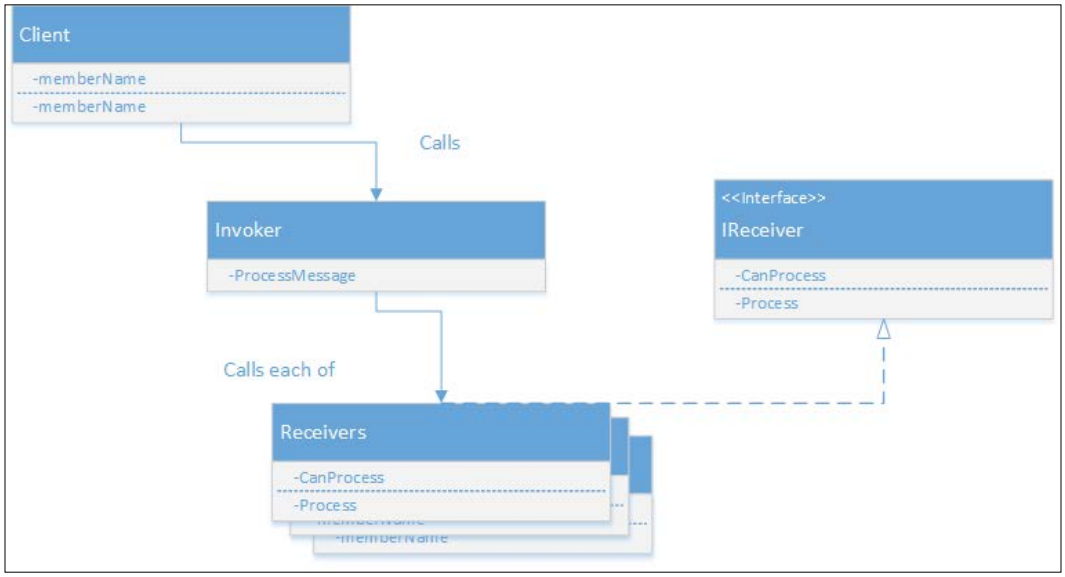
```
class ClerkOfTheCourt {
  IsInterestedInComplaint(complaint) {
    //decide if this is a complaint which can be solved by the clerk
    if(isInterested())
      return true;
    return false;
  }
  ListenToComplaint(complaint) {
    //perform some operation
    //return solution to the complaint
    return "";
  }
}
JudicialSystem.ClerkOfTheCourt = ClerkOfTheCourt;
class King {
  IsInterestedInComplaint(complaint) {
    return true;//king is the final member in the chain so must
      return true
  }
  ListenToComplaint(complaint) {
    //perform some operation
    //return solution to the complaint
    return "";
  }
}
JudicialSystem.King = King;
```

Each one of these classes implements a different approach to solving the complaint. We need to chain them together making sure that the king is in the default position. This can be seen in this code:

```
class ComplaintResolver {
  constructor() {
    this.complaintListeners = new Array();
     this.complaintListeners.push(new ClerkOfTheCourt());
     this.complaintListeners.push(new King());
```

```
    }
    ResolveComplaint(complaint) {
      for (var i = 0; i < this.complaintListeners.length; i++) {
        if
          (this.complaintListeners[i].IsInterestedInComplaint(complaint))
        {
          return
            this.complaintListeners[i].ListenToComplaint(complaint);
        }
      }
    }
  }
```

This code will work its way through each of the listeners until it finds one that is interested in hearing the complaint. In this version the result is returned immediately, halting any further processing. There are variations of this pattern in which multiple listeners could fire, even allowing the listeners to mutate the parameters for the next listener. The following diagram shows multiple listeners configured:

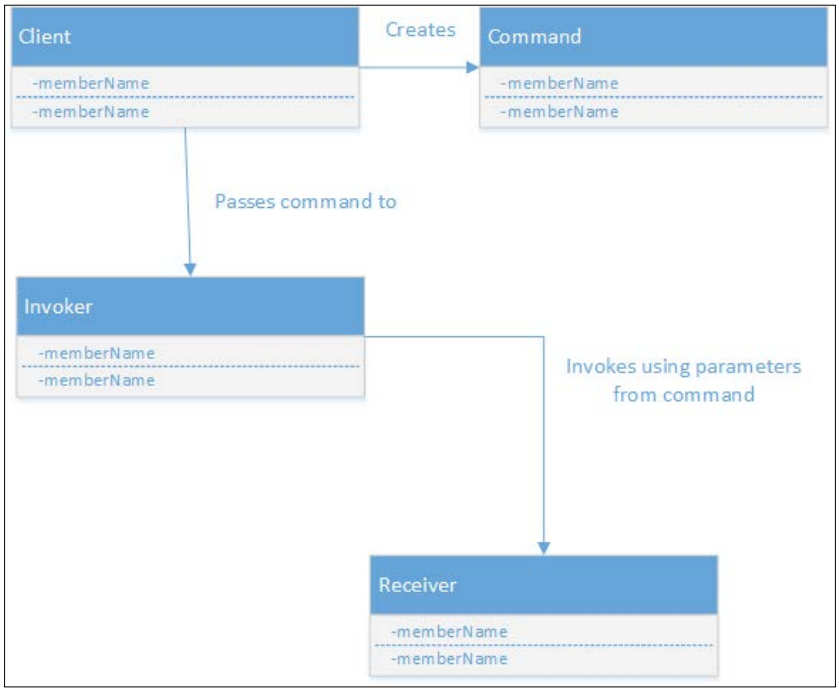


Chain of responsibility is a highly useful pattern in JavaScript. In browser-based JavaScript the events that fire fall through a chain of responsibility. For instance you can attach multiple listeners to the click event on a link and each of them will fire and then, finally, the default navigation listener. It is likely that you're using chain of responsibility in much of your code without even knowing it.

Command

The command pattern is a method of encapsulating both the parameters to a method, as well as the current object state, and which method is to be called. In effect the command pattern packs up everything needed to call a method at a later date into a nice little package. Using this approach one can issue a command and wait until a later date to decide which piece of code will execute the command. This package can then be queued or even serialized for later execution. Having a single point of command execution also allows for easily adding functionality such as undo or command logging.

This pattern can be a bit difficult to imagine so let's break it down into its components:



Command message

The first component of the command pattern is, predictably, the command itself. As I mentioned, the command encapsulates everything needed to invoke a method. This includes the method name, the parameters, and any global state. As you can imagine keeping track of global state in each command is very difficult. What happens if the global state changes after the command has been created? This dilemma is yet another reason why using a global state is problematic and should be avoided.

There are a couple of options for setting up commands. At the simple end of the scale all that is needed is to track a function and a set of parameters. Because functions are first class objects in JavaScript, they can easily be saved into an object. We can also save the parameters to the function into a simple array. Let's build a command using this very simple approach.

The deferred nature of commands suggests an obvious metaphor in the land of Westeros. There are no methods of communicating quickly in Westeros. The best method is to attach small messages to birds and release them. The birds have a tendency to want to return to their homes, so each lord raises a number of birds in their home and, when they come of age, sends them to other lords who might wish to communicate with them. The lords keep an aviary of birds and retain records of which bird will travel to which other lord. The king of Westeros sends many of his commands to his loyal lords through this method.

The commands sent by the king contain all necessary instructions for the lords. The command may be something like bring your troops and the arguments to that command may be a number of troops, a location, and a date by which the command must be carried out.

In JavaScript the simplest way of representing this is through an array:

```
var simpleCommand = new Array();
simpleCommand.push(new LordInstructions().BringTroops);
simpleCommand.push("King's Landing");
simpleCommand.push(500);
simpleCommand.push(new Date());
```

This array can be passed around and invoked at will. To invoke it, a generic function can be used:

```
simpleCommand[0](simpleCommand[1], simpleCommand[2],
simpleCommand[3]);
```

As you can see, this function only works for commands with three arguments. You can, of course, expand this to any number:

```
simpleCommand[0](simpleCommand[1], simpleCommand[2],
simpleCommand[3], simpleCommand[4], simpleCommand[5],
simpleCommand[6]);
```

The additional parameters are undefined, but the function doesn't use them so there are no ill effects. Of course, this is not at all an elegant solution.

It is desirable to build a class for each sort of command. This allows you to ensure the correct arguments have been supplied and easily distinguish the different sorts of commands in a collection. Typically, commands are named using the imperative, as they are instructions. Examples of this are BringTroops, Surrender, SendSupplies, and so on.

Let's transform our ugly simple command into a proper class:

```
class BringTroopsCommand {
  constructor(location, numberOfTroops, when) {
    this._location = location;
    this._numberOfTroops = numberOfTroops;
    this._when = when;
  }
  Execute() {
    var receiver = new LordInstructions();
    receiver.BringTroops(this._location, this._numberOfTroops,
      this._when);
  }
}
```

We may wish to implement some logic to ensure that the parameters passed into the constructor are correct. This will ensure that the command fails on creation instead of on execution. It is easier to debug the issue during creation rather than during execution as execution could be delayed, even for days. The validation won't be perfect, but even if it catches only a small portion of the errors it is helpful.

As mentioned these commands can be saved for later use in memory or even written to disk.

Invoker

The invoker is the part of the command pattern which instructs the command to execute its instructions. The invoker can really be anything: a timed event, a user interaction, or just the next step in the process may all trigger invocation. When we executed the `simpleCommand` command in the preceding section, we were playing at being the invoker. In a more rigorous command the invoker might look something like the following:

```
command.Execute()
```

As you can see, invoking a command is very easy. Commands may be invoked at once or at some later date. One popular approach is to defer the execution of the command to the end of the event loop. This can be done in a node with:

```
process.nextTick(function(){command.Execute();});
```

The function `process.nextTick` defers the execution of a command to the end of the event loop such that, if it is executed next time the process has nothing to do.

Receiver

The final component in the command pattern is the receiver. This is the target of the command execution. In our example we created a receiver called `LordInstructions`:

```
class LordInstructions {
  BringTroops(location, numberOfTroops, when) {
    console.log(`You have been instructed to bring
      ${numberOfTroops} troops to ${location} by ${when}`);
  }
}
```

The receiver knows how to perform the action that the command has deferred. There need not be anything special about the receiver, in fact it may be any class at all.

Together these components make up the command pattern. A client will generate a command, pass it off to an invoker that may delay the command or execute it at once, and the command will act upon a receiver.

In the case of building an undo stack, the commands are special, in that they have both an `Execute` and an `Undo` method. One takes the application state forward and the other takes it backwards. To perform an undo, simply pop the command off the undo stack, execute the `Undo` function, and push it onto a redo stack. For redo, pop from redo, execute `Execute`, and push to the undo stack. Simple as that, although one must make sure all state mutations are performed through commands.

The GoF book outlines a slightly more complicated set of players for the command pattern. This is largely due to the reliance on interfaces that we've avoided in JavaScript. The pattern becomes much simpler thanks to the prototype inheritance model in JavaScript.

The command pattern is a very useful one for deferring the execution of some piece of code. We'll actually explore the command pattern and some useful companion patterns in *Chapter 10*, *Messaging Patterns*.

Interpreter

The interpreter pattern is an interesting pattern as it allows for the creation of your own language. This might sound like something of a crazy idea, we're already writing JavaScript, why would we want to create a new language? Since the publication of the GoF book **Domain specific languages** (**DSLs**) have had something of a renaissance. There are situations where it is quite useful to create a language that is specific to one requirement. For instance the **Structured Query Language** (**SQL**) is very good at describing the querying of relational databases. Equally, regular expressions have proven themselves to be highly effective for the parsing and manipulation of text.

There are many scenarios in which being able to create a simple language is useful. That's really the key: a simple language. Once the language gets more complicated, the advantages are quickly lost to the difficulty of creating what is, in effect, a compiler.

This pattern is different from those we've seen to this point as there is no real class structure that is defined by the pattern. You can design your language interpreter however you wish.

Example

For our example let us define a language which can be used to describe historical battles in the land of Westeros. The language must be simple for clerics to write and easy to read. We'll start by creating a simple grammar:

```
(aggressor -> battle ground <- defender) -> victor
```

Here you can see that we're just writing out a rather nice syntax that will let people describe battles. A battle between Robert Baratheon and RhaegarTargaryen at the river Trident would look like the following:

```
(Robert Baratheon -> River Trident <- RhaegarTargaryen) -> Robert
  Baratheon
```

Using this grammar we would like to build some code which is able to query a list of battles for answers. In order to do this we're going to rely on regular expressions. For most languages this wouldn't be a good approach as the grammar is too complicated. In those cases one might wish to create a lexor and a parser and build up syntax trees, however, by that point you may wish to re-examine if creating a DSL is really a good idea. For our language the syntax is very simple so we can get away with regular expressions.

Implementation

The first thing we establish is a JavaScript data model for the battle like so:

```
class Battle {
  constructor(battleGround, agressor, defender, victor) {
    this.battleGround = battleGround;
    this.agressor = agressor;
    this.defender = defender;
    this.victor = victor;
  }
}
```

Next we need a parser:

```
class Parser {
  constructor(battleText) {
    this.battleText = battleText;
    this.currentIndex = 0;
    this.battleList = battleText.split("\n");
  }
  nextBattle() {
   if (!this.battleList[0])
     return null;
    var segments = this.battleList[0].match(/\((.+?)\s?-
      >\s?(.+?)\s?<-\s?(.+?)\s?->\s?(.+)/);
    return new Battle(segments[2], segments[1], segments[3],
      segments[4]);
  }
}
```

It is likely best that you don't think too much about that regular expression. However, the class does take in a list of battles (one per line) and using `next Battle`, allows one to parse them. To use the class we simply need to do the following:

```
var text = "(Robert Baratheon -> River Trident <-
  RhaegarTargaryen) -> Robert Baratheon";
var p = new Parser(text);
p.nextBattle()
```

This will be the output:

```
{
  battleGround: 'River Trident',
  agressor: 'Robert Baratheon',
  defender: 'RhaegarTargaryen)',
  victor: 'Robert Baratheon'
}
```

This data structure can now be queried like one would for any other structure in JavaScript.

As I mentioned earlier there is no fixed way to implement this pattern, so the implementation done in the preceding code is provided simply as an example. Your implementation will very likely look very different and that is just fine.

Interpreter can be a useful pattern in JavaScript. It is, however, a pretty infrequently used pattern in most situations. The best example of a language interpreted in JavaScript is the less language that is compiled, by JavaScript, to CSS.

Iterator

Traversing collections of objects is an amazingly common problem. So much so that many languages provide for special constructs just for moving through collections. For example C# has a `foreach` loop and Python has `for x in`. These looping constructs are frequently built on top of an iterator. An iterator is a pattern that provides a simple method for selecting, sequentially, the next item in a collection.

The interface for an iterator looks like this:

```
interface Iterator{
  next();
}
```

Implementation

In the land of Westeros there is a well-known sequence of people in line for the throne in the very unlikely event that the king was to die. We can set up a handy iterator over the top of this collection and simply call `next` on it should the ruler die:

```
class KingSuccession {
  constructor(inLineForThrone) {
    this.inLineForThrone = inLineForThrone;
    this.pointer = 0;
  }
  next() {
    return this.inLineForThrone[this.pointer++];
  }
}
```

This is primed with an array and then we can call it:

```
var king = new KingSuccession(["Robert Baratheon"
  ,"JofferyBaratheon", "TommenBaratheon"]);
king.next() //'Robert Baratheon'
king.next() //'JofferyBaratheon'
king.next() //'TommenBaratheon'
```

An interesting application of iterators is to not iterate over a fixed collection. For instance an iterator can be used to generate sequential members of an infinite set like the fibonacci sequence:

```
class FibonacciIterator {
  constructor() {
    this.previous = 1;
    this.beforePrevious = 1;
  }
  next() {
    var current = this.previous + this.beforePrevious;
    this.beforePrevious = this.previous;
    this.previous = current;
    return current;
  }
}
```

This is used like so:

```
var fib = new FibonacciIterator()
fib.next() //2
fib.next() //3
fib.next() //5
fib.next() //8
fib.next() //13
fib.next() //21
```

Iterators are handy constructs allowing for exploring not just arrays but any collection or even any generated list. There are a ton of places where this can be used to great effect.

ECMAScript 2015 iterators

Iterators are so useful that they are actually part of the next generation of JavaScript. The iterator pattern used in ECMAScript 2015 is a single method that returns an object that contains `done` and `value`. `done` is `true` when the iterator is at the end of the collection. What is nice about the ECMAScript 2015 iterators is that the array collection in JavaScript will support the iterator. This opens up a new syntax which can largely replace the `for` loop:

```
var kings = new KingSuccession(["Robert Baratheon"
  ,"JofferyBaratheon", "TommenBaratheon"]);
for(var king of kings){
  //act on members of kings
}
```

Iterators are a syntactic nicety that has long been missing from JavaScript. Another great feature of ECMAScript-2015 are generators. This is, in effect, a built in iterator factory. Our fibonacci sequence could be rewritten like the following:

```
function* FibonacciGenerator (){
  var previous = 1;
  var beforePrevious = 1;
  while(true){
    var current = previous + beforePrevious;
    beforePrevious = previous;
    previous = current;
    yield current;
  }
}
```

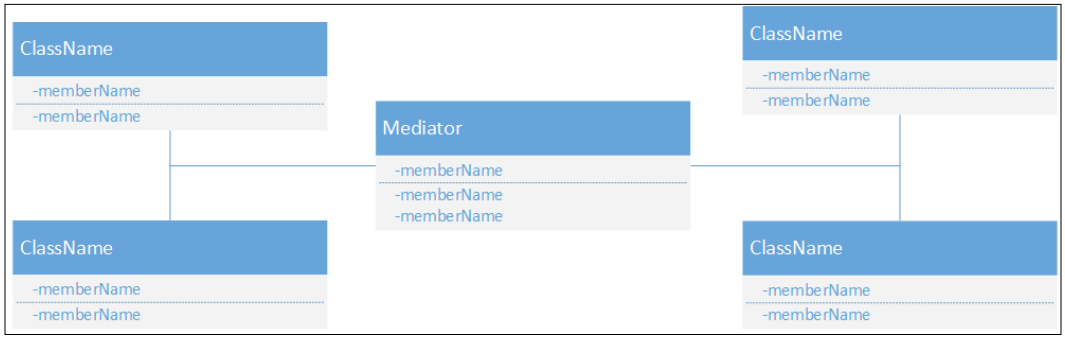
This is used like so:

```
var fib = new FibonacciGenerator()
fib.next().value //2
fib.next().value //3
fib.next().value //5
fib.next().value //8
fib.next().value //13
fib.next().value //21
```

Mediator

Managing many-to-many relationships in classes can be a complicated prospect. Let's consider a form that contains a number of controls, each of which wants to know if other controls on the page are valid before performing their action. Unfortunately, having each control know about each other control creates a maintenance nightmare. Each time a new control is added, each other control needs to be modified.

A mediator will sit between the various components and act as a single place in which message routing changes can be made. By doing so the mediator simplifies the otherwise complex work needed to maintain the code. In the case of controls on a form, the mediator is likely to be the form itself. The mediator acts much like a real life mediator would, clarifying and routing information exchange between a number of parties:



Implementation

In the land of Westeros there are many times when a mediator is needed. Frequently the mediator ends up deceased, but I'm sure that won't be the case with our example.

There are a number of great families in Westeros who own large castles and vast tracts of land. Lesser lords swear themselves to the great houses forming an alliance, frequently supported through marriage.

When coordinating the various houses sworn to them, the great lord will act as a mediator, communicating information back and forth between the lesser lords and resolving any disputes they may have amongst themselves.

In this example we'll greatly simplify the communication between the houses and say that all messages pass through the great lord. In this case we'll use the house of Stark as our great lord. They have a number of other houses which talk with them. Each of the houses looks roughly like the following:

```
class Karstark {
  constructor(greatLord) {
    this.greatLord = greatLord;
  }
  receiveMessage(message) {
  }
  sendMessage(message) {
    this.greatLord.routeMessage(message);
  }
}
```

They have two functions, one of which receives messages from a third party and one of which sends messages out to their great lord, which is set upon instantiation. The `HouseStark` class looks like the following:

```
class HouseStark {
  constructor() {
    this.karstark = new Karstark(this);
    this.bolton = new Bolton(this);
    this.frey = new Frey(this);
    this.umber = new Umber(this);
  }
  routeMessage(message) {
  }
}
```

By passing all messages through the `HouseStark` class the various other houses do not need to concern themselves with how their messages are routed. This responsibility is handed off to `HouseStark` which acts as the mediator.

Mediators are best used when the communication is both complex and well defined. If the communication is not complex then the mediator adds extra complexity. If the communication is ill defined then it becomes difficult to codify the communication rules in a single place.

Simplifying communication between many-to-many objects is certainly useful in JavaScript. I would actually argue that in many ways jQuery acts as a mediator. When acting on a set of items on the page, it serves to simplify communication by abstracting away code's need to know exactly which objects on the page are being changed. For instance:

```
$(".error").slideToggle();
```

Is jQuery shorthand for toggling the visibility of all the elements on the page which have the `error` class?

Memento

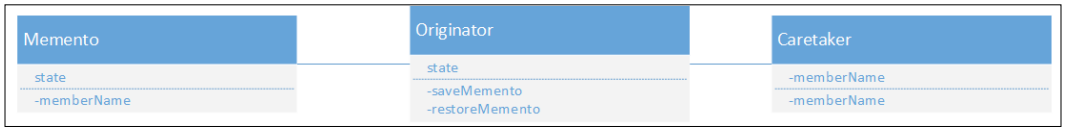
In the section on the command pattern we talked briefly about the ability to undo operations. Creating reversible commands is not always possible. For many operations there is no apparent reversing operation which can restore the original state. For instance, imagine code which squares a number:

```
class SquareCommand {
  constructor(numberToSquare) {
    this.numberToSquare = numberToSquare;
  }
  Execute() {
    this.numberToSquare *= this.numberToSquare;
  }
}
```

Giving this code -9 will result in 81 but giving it 9 will also result in 81. There is no way to reverse this command without additional information.

The memento pattern provides an approach to restore the state of objects to a previous state. The memento keeps a record of the previous values of a variable and provides the functionality to restore them. Keeping a memento around for each command allows for easy restoration of non-reversible commands.

In addition to an undo-stack there are many instances where having the ability to roll back the state of an object is useful. For instance doing what-if analysis requires that you make some hypothetical changes to state and then observe how things change. The changes are generally not permanent so they could be rolled back using the memento pattern or, if the projects are desirable, left in place. A diagram of the memento pattern can be seen here:



A typical memento implementation involves three players:

- **Originator**: The originator holds some form of state and provides an interface for generating new mementos.
- **Caretaker**: This is the client of the pattern, it is what requests that new mementos be taken and governs when they are to be restored.

- **Memento**: This is a representation of the saved state of the originator. This is what can be persisted to storage to allow for rolling back.

It can help to think of the members of the memento pattern as a boss and a secretary taking notes. The boss (caretaker) dictates some memo to the secretary (originator) who writes notes in a notepad (memento). From time to time the boss may request that the secretary cross out what he has just written.

The involvement of the caretaker can be varied slightly with the memento pattern. In some implementation the originator will generate a new memento each time a change is made to its state. This is commonly known as copy on write, as a new copy of the state is created and the change applied to it. The old version can be saved to a memento.

Implementation

In the land of Westeros there are a number of soothsayers, foretellers of the future. They work by using magic to peer into the future and examine how certain changes in the present will play out in the future. Often there is need for numerous foretelling with slightly different starting conditions. When setting their starting conditions, a memento pattern is invaluable.

We start off with a world state which gives information on the state of the world for a certain starting point:

```
class WorldState {
  constructor(numberOfKings, currentKingInKingsLanding, season) {
    this.numberOfKings = numberOfKings;
    this.currentKingInKingsLanding = currentKingInKingsLanding;
    this.season = season;
  }
}
```

This `WorldState` class is responsible for tracking all the conditions that make up the world. It is what is altered by the application every time a change to the starting conditions is made. Because this world state encompasses all the states for the application, it can be used as a memento. We can serialize this object and save it to disk or send it back to some history server somewhere.

The next thing we need is a class which provides the same state as the memento and allows for the creation and restoration of mementos. In our example we've called this as `WorldStateProvider`:

```
class WorldStateProvider {
  saveMemento() {
    return new WorldState(this.numberOfKings,
```

```
        this.currentKingInKingsLanding, this.season);
    }
    restoreMemento(memento) {
      this.numberOfKings = memento.numberOfKings;
      this.currentKingInKingsLanding =
        memento.currentKingInKingsLanding;
      this.season = memento.season;
    }
  }
```

Finally we need a client for the foretelling, which we'll call `Soothsayer`:

```
  class Soothsayer {
    constructor() {
      this.startingPoints = [];
      this.currentState = new WorldStateProvider();
    }
    setInitialConditions(numberOfKings, currentKingInKingsLanding,
      season) {
      this.currentState.numberOfKings = numberOfKings;
      this.currentState.currentKingInKingsLanding =
        currentKingInKingsLanding;
      this.currentState.season = season;
    }
    alterNumberOfKingsAndForetell(numberOfKings) {
      this.startingPoints.push(this.currentState.saveMemento());
      this.currentState.numberOfKings = numberOfKings;
    }
    alterSeasonAndForetell(season) {
      this.startingPoints.push(this.currentState.saveMemento());
      this.currentState.season = season;
    }
    alterCurrentKingInKingsLandingAndForetell(currentKingInKingsLanding)
    {
      this.startingPoints.push(this.currentState.saveMemento());
      this.currentState.currentKingInKingsLanding =
        currentKingInKingsLanding;
      //run some sort of prediction
    }
    tryADifferentChange() {
      this.currentState.restoreMemento(this.startingPoints.pop());
    }
  }
```

This class provides a number of convenience methods which alter the state of the world and then run a foretelling. Each of these methods pushes the previous state into the history array, `startingPoints`. There is also a method, `tryADifferentChange`, which undoes the previous state change ready to run another foretelling. The undo is performed by loading back the memento which is stored in an array.

Despite a great pedigree it is very rare that client side JavaScript applications provide an undo function. I'm sure there are various reasons for this, but for the most part it is likely that people do not expect such functionality. However in most desktop applications, having an undo function is expected. I imagine that, as client side applications continue to grow in their capabilities, undo functionality will become more important. When it does, the memento pattern is a fantastic way of implementing the undo stack.

Observer

The observer pattern is perhaps the most used pattern in the JavaScript world. The pattern is used especially with modern single pages applications; it is a big part of the various libraries that provide **Model View View-Model** (**MVVM**) functionality. We'll explore those patterns in some detail in *Chapter 7*, *Reactive Programming*.

It is frequently useful to know when the value on an object has changed. In order to do so you could wrap up the property of interest with a getter and setter:

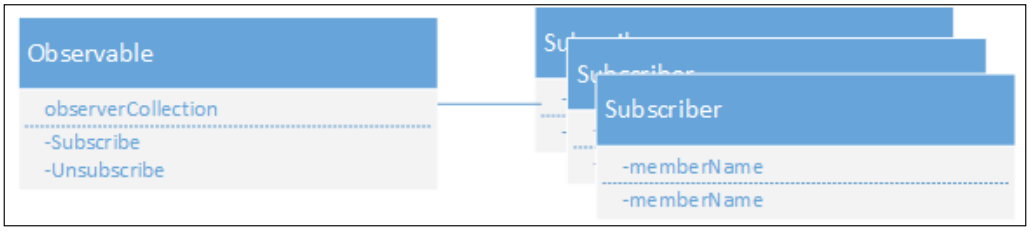
```
class GetterSetter {
  GetProperty() {
    return this._property;
  }
  SetProperty(value) {
    this._property = value;
  }
}
```

The setter function can now be augmented with a call to some other object which is interested in knowing that a value has changed:

```
SetProperty(value) {
  var temp = this._property;
  this._property = value;
  this._listener.Event(value, temp);
}
```

This setter will now notify the listener that a property change has occurred. In this case both the old and new value have been included. This is not necessary as the listener can be tasked with keeping track of the previous value.

The observer pattern generalizes and codifies this idea. Instead of having a single call to the listener, the observer pattern allows interested parties to subscribe to change notifications. Multiple subscribers can be seen in the following diagram:



Implementation

The court of Westeros is a place of great intrigue and trickery. Controlling who is on the throne and what moves they make is a complex game. Many of the players in the game of thrones employ numerous spies to discover what moves others are making. Frequently these spies are employed by more than one player and must report what they have found to all of the players.

The spy is a perfect place to employ the observer pattern. In our particular example, the spy being employed is the official doctor to the king and the players are very interested in how much painkiller is being prescribed to the ailing king. Knowing this can give a player advanced knowledge of when the king might die – a most useful piece of information.

The spy looks like the following:

```
class Spy {
  constructor() {
    this._partiesToNotify = [];
  }
  Subscribe(subscriber) {
    this._partiesToNotify.push(subscriber);
  }
  Unsubscribe(subscriber) {
    this._partiesToNotify.remove(subscriber);
  }
  SetPainKillers(painKillers) {
    this._painKillers = painKillers;
```

```
      for (var i = 0; i < this._partiesToNotify.length; i++) {
        this._partiesToNotify[i](painKillers);
      }
    }
  }
```

In other languages, the subscriber usually has to comply with a certain interface and the observer will call only the interface method. This encumbrance doesn't exist with JavaScript and, in fact, we just give the `Spy` class a function. This means that there is no strict interface required for the subscriber. This is an example:

```
class Player {
  OnKingPainKillerChange(newPainKillerAmount) {
    //perform some action
  }
}
```

This can be used like so:

```
let s = new Spy();
let p = new Player();
s.Subscribe(p.OnKingPainKillerChange); //p is now a subscriber
s.SetPainKillers(12); //s will notify all subscribers
```

This provides a very simple and highly effective way of building observers. Having subscribers decouples the subscriber from the observable object.

The observer pattern can also be applied to methods as well as properties. In so doing you can provide hooks for additional behavior to happen. This is a common method of providing a plugin infrastructure for JavaScript libraries.

In browsers all the event listeners on various items in the DOM are implemented using the observer pattern. For instance, using the popular jQuery library, one can subscribe to all the `click` events on buttons on a page by doing the following:

```
$("body").on("click", "button", function(){/*do something*/})
```

Even in vanilla JavaScript the same pattern applies:

```
let buttons = document.getElementsByTagName("button");
for(let i =0; i< buttons.length; i++)
{
  buttons[i].onclick = function(){/*do something*/}
}
```

Clearly the observer pattern is very useful when dealing with JavaScript. There is no need to change the pattern in any significant fashion.

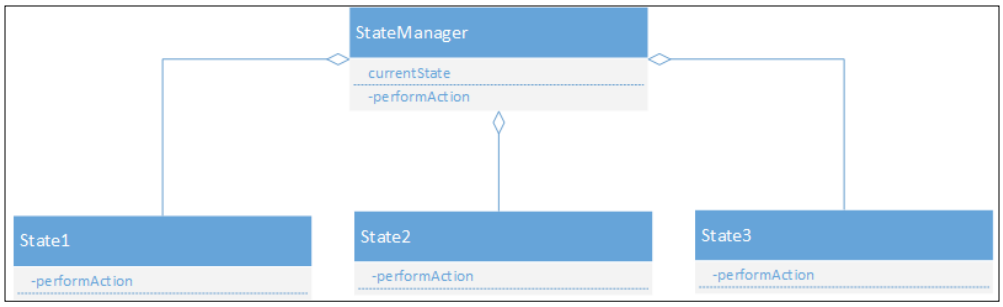
State

State machines are an amazingly useful device in computer programming. Unfortunately they are not used very frequently by most programmers. I'm sure that at least some of the objection to state machines is that many people implement them as a giant `if` statement like so:

```
function (action, amount) {
  if (this.state == "overdrawn" && action == "withdraw") {
    this.state = "on hold";
  }
  if (this.state == "on hold" && action != "deposit") {
    this.state = "on hold";
  }
  if (this.state == "good standing" && action == "withdraw" &&
    amount <= this.balance) {
    this.balance -= amount;
  }
  if (this.state == "good standing" && action == "withdraw" &&
    amount >this.balance) {
    this.balance -= amount;
    this.state = "overdrawn";
  }
};
```

This is just a sample of what could be much longer. The `if` statements of this length are painful to debug and highly error prone. Simply flipping a greater than sign is enough to drastically change how the `if` statement works.

Instead of using a single giant `if` statement block we can make use of the state pattern. The state pattern is characterized by having a state manager which abstracts away the internal state and proxies a message through to the appropriate state which is implemented as a class. All the logic within states and governing state transitions is governed by the individual state classes. The state manager pattern can be seen in the following diagram:



Splitting state into a class per state allows for much smaller blocks of code to debug and makes testing much easier.

The interface for the state manager is fairly simple and usually just provides the methods needed to communicate with the individual states. The manager may also contain some shared state variables.

Implementation

As alluded to in the `if` statement example, Westeros has a banking system. Much of it is centered on the island of Braavos. Banking there runs in much the same way as banking here, with accounts, deposits, and withdrawals. Managing the state of a bank account involves keeping an eye on all of the transactions and changing the state of the bank account in accordance with the transactions.

Let's take a look at some of the code which is needed to manage a bank account at the Iron Bank of Braavos. First is the state manager:

```
class BankAccountManager {
  constructor() {
    this.currentState = new GoodStandingState(this);
  }
  Deposit(amount) {
    this.currentState.Deposit(amount);
  }
  Withdraw(amount) {
    this.currentState.Withdraw(amount);
  }
  addToBalance(amount) {
    this.balance += amount;
  }
  getBalance() {
    return this.balance;
  }
  moveToState(newState) {
    this.currentState = newState;
  }
}
```

The `BankAccountManager` class provides a state for the current balance and also the current state. To protect the balance, it provides an accessory for reading the balance and another for adding to the balance. In a real banking application, I would rather expect the function that sets the balance, have more protection than this. In this version of `BankManager`, the ability to manipulate the current state is accessible to the states. They have the responsibility to change states. This functionality can be centralized in the manager but that increases the complexity of adding new states.

We've identified three simple states for the bank account: `Overdrawn`, `OnHold`, and `GoodStanding`. Each one is responsible for dealing with withdrawals and deposits when in that state. The `GoodStandingstate` class looks like the following:

```
class GoodStandingState {
  constructor(manager) {
    this.manager = manager;
  }
  Deposit(amount) {
    this.manager.addToBalance(amount);
  }
  Withdraw(amount) {
    if (this.manager.getBalance() < amount) {
      this.manager.moveToState(new OverdrawnState(this.manager));
    }
    this.manager.addToBalance(-1 * amount);
  }
}
```

The `OverdrawnState` class looks like the following:

```
class OverdrawnState {
  constructor(manager) {
    this.manager = manager;
  }
  Deposit(amount) {
    this.manager.addToBalance(amount);
    if (this.manager.getBalance() > 0) {
      this.manager.moveToState(new
        GoodStandingState(this.manager));
    }
  }
  Withdraw(amount) {
    this.manager.moveToState(new OnHold(this.manager));
    throw "Cannot withdraw money from an already overdrawn bank
      account";
  }
}
```

Finally, the `OnHold` state looks like the following:

```
class OnHold {
  constructor(manager) {
    this.manager = manager;
  }
  Deposit(amount) {
```

```
        this.manager.addToBalance(amount);
        throw "Your account is on hold and you must attend the bank to
          resolve the issue";
      }
      Withdraw(amount) {
        throw "Your account is on hold and you must attend the bank to
          resolve the issue";
      }
    }
```

You can see that we've managed to reproduce all the logic of the confusing `if` statement in a number of simple classes. The amount of code here looks to be far more than the `if` statement but, in the long run, encapsulating the code into individual classes will pay off.

There is plenty of opportunity to make use of this pattern within JavaScript. Keeping track of state is a typical problem in most applications. When the transitions between the states are complex, then wrapping it up in a state pattern is one method of simplifying things. It is also possible to build up a simple workflow by registering events as sequential. A nice interface for this might be a fluent one so that you could register states like the following:

```
goodStandingState
.on("withdraw")
.when(function(manager){return manager.balance > 0;})
  .transitionTo("goodStanding")
.when(function(manager){return mangaer.balance <=0;})
  .transitionTo("overdrawn");
```

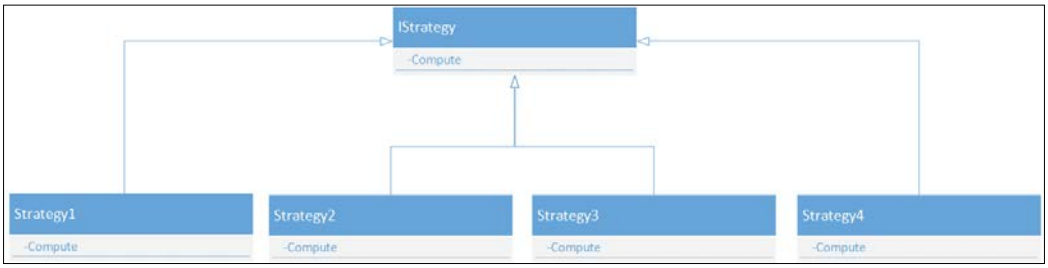
Strategy

It has been said that there is more than one way to skin a cat. I have, wisely, never looked into how many ways there are. The same is frequently true for algorithms in computer programming. Frequently there are numerous versions of an algorithm that trades off memory usage for CPU usage. Sometimes there are different approaches that provide different levels of fidelity. For example, performing a geo-location on a smart phone typically uses one of three different sources of data:

- GPS chip
- Cell phone triangulation
- Nearby WiFi points

Using the GPS chip provides the highest level of fidelity however it is also the slowest and requires the most battery. Looking at the nearby WiFi points requires very little energy and is very quick, however it provides poor fidelity.

The strategy pattern provides a method of swapping these strategies out in a transparent fashion. In a traditional inheritance model each strategy would implement the same interface which would allow for any of the strategies to be swapped in. The following diagram shows multiple strategies that could be swapped in:



Selecting the correct strategy to use can be done in a number of different ways. The simplest method is to select the strategy statically. This can be done through a configuration variable or even hard coded. This approach is best for times when the strategy changes infrequently or is specific to a single customer or user.

Alternately an analysis can be run on the dataset on which the strategy is to be run and then a proper strategy selected. If it is known that strategy A works better than strategy B when the data passed in is clustered around a mean, then a fast algorithm for analyzing spread could be run first and then the appropriate strategy selected.

If a particular algorithm fails on data of a certain type, this too can be taken into consideration when choosing a strategy. In a web application this can be used to call a different API depending on the shape of data. It can also be used to provide a fallback mechanism should one of the API endpoints be down.

Another interesting approach is to use progressive enhancement. The fastest and least accurate algorithm is run first to provide rapid user feedback. At the same time a slower algorithm is also run and, when it is finished, the superior results are used to replace the existing results. This approach is frequently used in the GPS situation outlined above. You may notice when using a map on a mobile device your location is updated a moment after the map loads; this is an example of progressive enhancement.

Finally, the strategy can be chosen completely at random. It sounds like a strange approach but can be useful when comparing the performance of two different strategies. In this case, statistics would be gathered about how well each approach works and an analysis run to select the best strategy. The strategy pattern can be the foundation for A/B testing.

Selecting which strategy to use can be an excellent place to apply the factory pattern.

Implementation

In the land of Westeros there are no planes, trains, or automobiles but there is still a wide variety of different ways to travel. One can walk, ride a horse, sail on a seagoing vessel, or even take a boat down the river. Each one has different advantages and drawbacks but in the end they still take a person from point A to point B. The interface might look something like the following:

```
export interface ITravelMethod{
  Travel(source: string, destination: string) : TravelResult;
}
```

The travel result communicates back to the caller some information about the method of travel. In our case we track how long the trip will take, what the risks are, and how much it will cost:

```
class TravelResult {
  constructor(durationInDays, probabilityOfDeath, cost) {
    this.durationInDays = durationInDays;
    this.probabilityOfDeath = probabilityOfDeath;
    this.cost = cost;
  }
}
```

In this scenario we might like to have an additional method which predicts some of the risks to allow for automating selection of a strategy.

Implementing the strategies is as simple as the following:

```
class SeaGoingVessel {
  Travel(source, destination) {
    return new TravelResult(15, .25, 500);
  }
}

class Horse {
  Travel(source, destination) {
    return new TravelResult(30, .25, 50);
```

```
  }
}

class Walk {
  Travel(source, destination) {
    return new TravelResult(150, .55, 0);
  }
}
```

In a traditional implementation of the strategy pattern the method signature for each strategy should be the same. In JavaScript there is a bit more flexibility as excess parameters to a function are ignored and missing parameters can be given default values.

Obviously, the actual calculations around risk, cost, and duration would not be hard coded in an actual implementation. To make use of these one needs only to do the following:

```
var currentMoney = getCurrentMoney();
var strat;
if (currentMoney> 500)
  strat = new SeaGoingVessel();
else if (currentMoney> 50)
  strat = new Horse();
else
  strat = new Walk();
var travelResult = strat.Travel();
```

To improve the level of abstraction for this strategy we might replace the specific strategies with more generally named ones that describe what it is we're optimizing for:

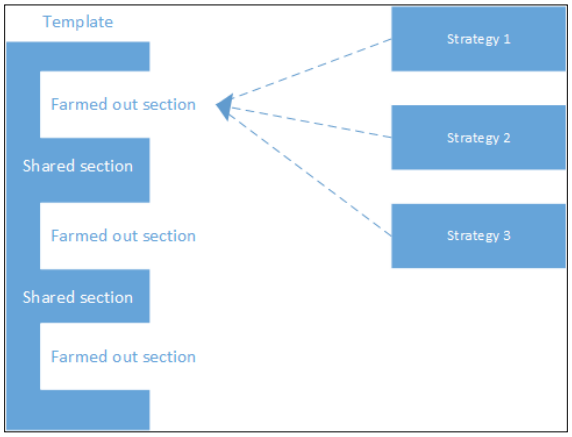
```
var currentMoney = getCurrentMoney();
var strat;
if (currentMoney> 500)
  strat = new FavorFastestAndSafestStrategy();
else
  strat = new FavorCheapest();
var travelResult = strat.Travel();
```

Strategy is a very useful pattern in JavaScript. We're also able to make the approach much simpler than in a language which doesn't use prototype inheritance: there is no need for an interface. We don't need to return the same shaped object from each of the different strategies. So long as the caller is somewhat aware that the returned object may have additional fields, this is a perfectly reasonable, if difficult to maintain, approach.

Template method

The strategy pattern allows for replacing an entire algorithm with a complimentary one. Frequently, replacing the entire algorithm is overkill: the vast majority of the algorithm remains the same in every strategy with only minor variations in specific sections.

The template method pattern is an approach which allows for some sections of an algorithm to be shared and other sections implemented using different approaches. These farmed out sections can be implemented by any one of a family of methods:

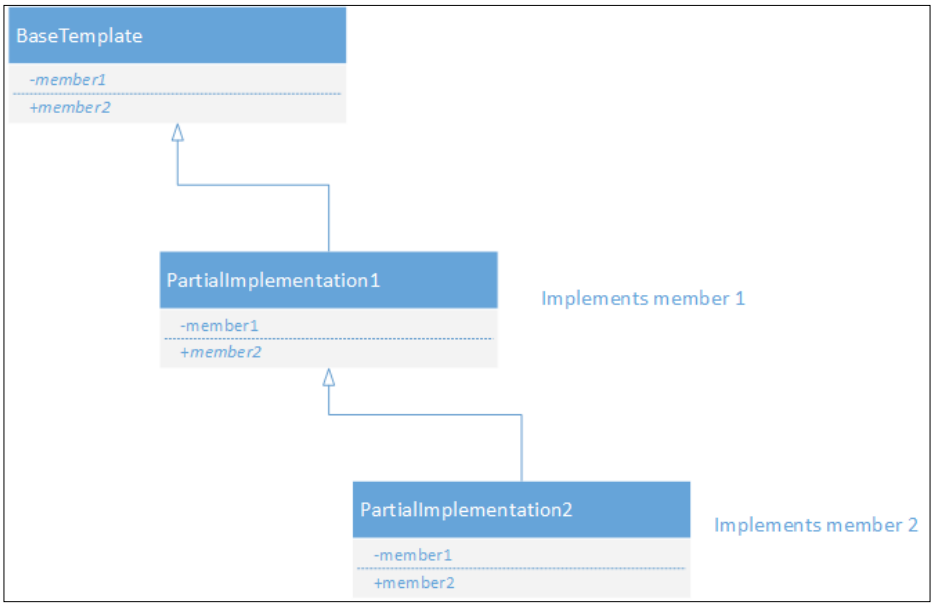


The template class implements parts of the algorithm and leaves other parts as abstract to later be overridden by classes which extend it. The inheritance hierarchy can be several layers deep, with each level implementing more and more of the template class.



An abstract class is one that contains abstract methods. Abstract methods are simply methods that have no body to them. The abstract class cannot be used directly and must, instead, be extended by another class that implements the abstract methods. An abstract class may extend another abstract class so that not all methods need to be implemented by the extending class.

This approach applies the principles of progressive enhancement to an algorithm. We move closer and closer to a fully implemented algorithm and, at the same time, build up an interesting inheritance tree. The template method helps keep identical code to a single location while still allowing for some deviation. A chain of partial implementations can be seen in the following diagram:



Overriding methods left as abstract is a quintessential part of object oriented programming. It is likely that this pattern is one which you've used frequently without even being aware that it had a name.

Implementation

I have been told, by those in the know, that there are many different ways to produce beer. These beers differ in their choice of ingredients and in their method of production. In fact beer does not even need to contain hops – it can be made from any number of grains. However there are similarities between all beers. They are all created through the fermentation process and all proper beers contain some alcohol content.

In Westeros there are a great number of craftsmen who pride themselves on creating top notch beers. We would like to describe their processes as a set of classes, each one describing a different beer making methodology. We start with a simplified implementation of creating a beer:

```
class BasicBeer {
  Create() {
    this.AddIngredients();
    this.Stir();
    this.Ferment();
```

```
      this.Test();
      if (this.TestingPassed()) {
        this.Distribute();
      }
    }
    AddIngredients() {
      throw "Add ingredients needs to be implemented";
    }
    Stir() {
      //stir 15 times with a wooden spoon
    }
    Ferment() {
      //let stand for 30 days
    }
    Test() {
      //draw off a cup of beer and taste it
    }
    TestingPassed() {
      throw "Conditions to pass a test must be implemented";
    }
    Distribute() {
      //place beer in 50L casks
    }
  }
```

As there is no concept of abstract in JavaScript we've added exceptions to the various methods which must be overridden. The remaining methods can be changed but do not require it. An implementation of this for a raspberry beer would look like the following:

```
  class RaspberryBeer extends BasicBeer {
    AddIngredients() {
      //add ingredients, probably including raspberries
    }
    TestingPassed() {
      //beer must be reddish and taste of raspberries
    }
  }
```

Additional sub-classing may be performed at this stage for more specific raspberry beers.

The template method remains a fairly useful pattern in JavaScript. There is some added syntactic sugar around creating classes, but it isn't anything we haven't already seen in a previous chapter. The only warning I would give is that the template method uses inheritance and thus strongly couples the inherited classes with the parent class. This is generally not a desirable state of affairs.

Visitor

The final pattern in this section is the visitor pattern. The visitor provides a method of decoupling an algorithm from the object structure on which it operates. If we wanted to perform some action over a collection of objects which differ in type and we want to perform a different action depending on the object type, we would typically need to make use of a large `if` statement.

Let's get right into an example of this in Westeros. An army is made up of a few different classes of fighting person (it is important that we be politically correct as there are many notable female fighters in Westeros). However, each member of the army implements a hypothetical interface called `IMemberOfArmy`:

```
interface IMemberOfArmy{
  printName();
}
```

A simple implementation of this might be the following:

```
class Knight {
  constructor() {
    this._type = "Westeros.Army.Knight";
  }
  printName() {
    console.log("Knight");
  }
  visit(visitor) {
    visitor.visit(this);
  }
}
```

Now we have a collection of these different types, we can use an `if` statement to only call the `printName` function on the knights:

```
var collection = [];
collection.push(new Knight());
collection.push(new FootSoldier());
collection.push(new Lord());
collection.push(new Archer());
```

```
for (let i = 0; i<collection.length; i++) {
  if (typeof (collection[i]) == 'Knight')
    collection[i].printName();
  else
    console.log("Not a knight");
}
```

Except, if you run this code, you'll actually find that all we get is the following:

```
Not a knight
Not a knight
Not a knight
Not a knight
```

This is because, despite an object being a knight, it is still an object and `typeof` will return object in all cases.

An alternative approach is to use `instanceof` instead of `typeof`:

```
var collection = [];
collection.push(new Knight());
collection.push(new FootSoldier());
collection.push(new Lord());
collection.push(new Archer());

for (var i = 0; i < collection.length; i++) {
  if (collection[i] instanceof Knight)
    collection[i].printName();
  else
    console.log("No match");
}
```

The instance of approach works great until we run into somebody who makes use of the `Object.create` syntax:

```
collection.push(Object.create(Knight));
```

Despite being a knight this will return `false` when asked if it is an instance of `Knight`.

This poses something of a problem for us. The problem is exacerbated by the visitor pattern as it requires that the language supports method overloading. JavaScript does not really support this. There are various hacks which can be used to make JavaScript somewhat aware of overloaded methods but the usual advice is to simply not bother and create methods with different names.

Let's not abandon this pattern just yet, though; it is a useful pattern. What we need is a way to reliably distinguish one type from another. The simplest approach is to just define a variable on the class which denotes its type:

```
var Knight = (function () {
  function Knight() {
    this._type = "Knight";
  }
  Knight.prototype.printName = function () {
    console.log("Knight");
  };
  return Knight;
})();
```

Given the new `_type` variable we can now fake having real method overrides:

```
var collection = [];
collection.push(new Knight());
collection.push(new FootSoldier());
collection.push(new Lord());
collection.push(new Archer());

for (vari = 0; i<collection.length; i++) {
  if (collection[i]._type == 'Knight')
    collection[i].printName();
  else
    console.log("No match");
}
```

Given this approach we can now implement a visitor. The first step is to expand our various members of the army to have a generic method on them which takes a visitor and applies it:

```
var Knight = (function () {
  function Knight() {
    this._type = "Knight";
  }
  Knight.prototype.printName = function () {
    console.log("Knight");
  };
  Knight.prototype.visit = function (visitor) {
    visitor.visit(this);
  };
  return Knight;
})();
```

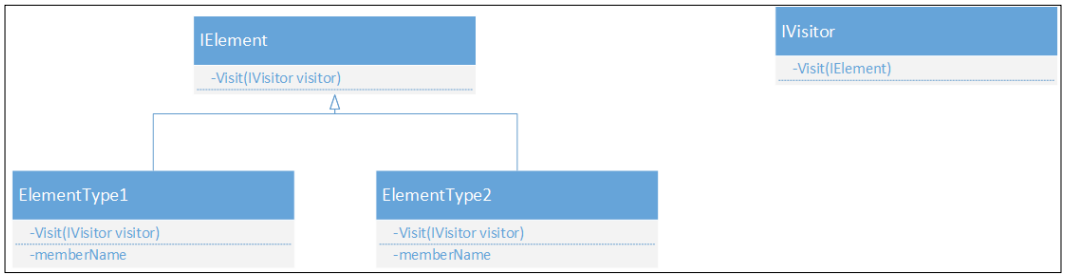
Now we need to build a visitor. This code approximates the `if` statements we had in the preceding code:

```
varSelectiveNamePrinterVisitor = (function () {
  function SelectiveNamePrinterVisitor() {
  }
  SelectiveNamePrinterVisitor.prototype.Visit = function
    (memberOfArmy) {
    if (memberOfArmy._type == "Knight") {
      this.VisitKnight(memberOfArmy);
    } else {
      console.log("Not a knight");
    }
  };

  SelectiveNamePrinterVisitor.prototype.VisitKnight = function
    (memberOfArmy) {
    memberOfArmy.printName();
  };
  return SelectiveNamePrinterVisitor;
})();
```

This visitor would be used as such:

```
var collection = [];
collection.push(new Knight());
collection.push(new FootSoldier());
collection.push(new Lord());
collection.push(new Archer());
var visitor = new SelectiveNamePrinterVisitor();
for (vari = 0; i<collection.length; i++) {
  collection[i].visit(visitor);
}
```

As you can see we've pushed the decisions about the type of the item in the collection down to the visitor. This decouples the items themselves from the visitor as can be seen in the following diagram:



If we allow the visitor to make decisions about what methods are called on the visited objects there is a fair bit of trickery required. If we can provide a constant interface for the visited objects then all the visitor needs do is call the interface method. This does, however, move logic from the visitor into the objects that are visited, which is contrary to the idea that the objects shouldn't know they are part of a visitor.

Whether suffering through the trickery is worthwhile is really an exercise for you. Personally I would tend to avoid using the visitor pattern in JavaScript as the requirements to get it working are complicated and non-obvious.

Hints and tips

Here are a couple of brief tips to keep in mind about some of the patterns we've seen in this chapter:

- When implementing the interpreter pattern you may be tempted to use JavaScript proper as your DSL and then use the `eval` function to execute the code. This is actually a very dangerous idea as `eval` opens up an entire world of security issues. It is generally considered to be very bad form to use `eval` in JavaScript.
- If you find yourself in the position to audit the changes to data in your project, then the memento pattern can easily be modified to suit. Instead of keeping track of just the state changes, you can also track when the change was made and who changed it. Saving these mementos to disk somewhere allows you to go back and rapidly build an audit log pointing to precisely what happened to change the object.
- The observer pattern is notorious for causing memory leaks when listeners aren't properly unregistered. This can happen even in a memory managed environment such as JavaScript. Be wary of failing to unhook observers.

Summary

In this chapter we've looked at a bunch of behavioral patterns. Some of these patterns such as observer and iterator will be ones you'll use almost every day, while others such as interpreter you might use no more than a handful of times in your entire career. Learning about these patterns should help you identify well-defined solutions to common problems.

Most of the patterns are directly applicable to JavaScript and some of them, such as the strategy pattern, become more powerful in a dynamic language. The only pattern we found that has some limitations is the visitor pattern. The lack of static classes and polymorphism makes this pattern difficult to implement without breaking proper separation of concerns.

These aren't, by any means, all of the behavioral patterns in existence. The programming community has spent the past two decades building on the ideas of the GoF book and identifying new patterns. The remainder of this book is dedicated to these newly identified patterns. The solutions may be very old ones but not generally recognized as common solutions until more recently. As far as I'm concerned this is the point where the book starts to get very interesting as we start looking at less well known and more JavaScript-specific patterns.

Part 2

Other Patterns

- Functional Programming
- Reactive Programming
- Application Patterns
- Web Patterns
- Messaging Patterns
- Microservices
- Patterns for Testing
- Advanced Patterns
- ECMAScript-2015/2016 Solutions Today

In Part 1 we focused on patterns originally identified in the GoF book that were the original impetus behind patterns in software design. In this part of the book we'll expand beyond those patterns to look at patterns that are related to functional programming, large-scale patterns for structuring an entire application, patterns which are specific to the Web, and messaging patterns. In addition, we'll look at patterns for testing and some rather interesting advanced patterns. Finally, we'll look at how we can get many of the features of the next version of JavaScript today.

6
Functional Programming

Functional programming is a different approach to development than the heavily object oriented approach that we have focused on so far. Object oriented programming is a fantastic tool for solving a great number of problems but it has some issues. Parallel programming within an object oriented context is difficult as the state can be changed by various different threads with unknown side effects. Functional programming does not permit state or mutable variables. Functions act as primary building blocks in functional programming. Places where you might have used a variable in the past will now use a function.

Even in a single threaded program, functions can have side-effects that change global state. This means that, when calling an unknown function, it can alter the whole flow of the program. This makes debugging a program quite difficult.

JavaScript is not a functional programming language but we can still apply some functional principles to our code. We'll look at a number of patterns in the functional space:

- Function passing
- Filters and pipes
- Accumulators
- Memoization
- Immutability
- Lazy instantiation

Functional functions are side-effect-free

A core tenant of functional programming is that functions should not change state. Values local to the function may be set but nothing outside the function may change. This approach is very useful for making code more maintainable. There need no longer be any concern that passing an array into a function is going to play havoc with its contents. This is especially a concern when using libraries that are not under your control.

There is no mechanism within JavaScript to prevent you from changing global state. Instead you must rely on developers to write side-effect-free functions. This may be difficult or not, depending on the maturity of the team.

It may not be desirable to put all the code from your application into functions, but separating as much as possible is desirable. There is a pattern called command query separation that suggests that methods should fall into two categories. Either they are a function that reads a value or they are a command that sets a value. Never the twain should meet. Keeping methods categorized like this eases in debugging and in code reuse.

One of the consequences of side effect-free functions is that they can be called any number of times with the same inputs and the result will be the same. Furthermore, because there are no changes to state, calling the function many times will not cause any ill side effects, other than making it run slower.

Function passing

In functional programming languages, functions are first class citizens. Functions can be assigned to variables and passed around just like you would with any other variable. This is not entirely a foreign concept. Even languages such as C had function pointers that could be treated just like other variables. C# has delegates and, in more recent versions, lambdas. The latest release of Java has also added support for lambdas as they have proven to be so useful.

JavaScript allows for functions to be treated as variables and even as objects and strings. In this way JavaScript is functional in nature.

Because of JavaScript's single threaded nature, callbacks are a common convention and you can find them pretty much everywhere. Consider calling a function at a later date on a web page. This is done by setting a timeout on the window object like so:

```
setTimeout(function(){alert("Hello from the past")}, 5 * 1000);
```

The arguments for the set timeout function are a function to call and a time to delay in milliseconds.

Irrespective of the JavaScript environment in which you're working, it is almost impossible to avoid functions in the shape of callbacks. Node.js' asynchronous processing model is highly dependent on being able to call a function and pass in something to be completed at a later date. 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 = new XMLHttpRequest()
xmlhttp.onreadystatechange = function()
if (xmlhttp.readyState==4 && xmlhttp.status==200){
  //process returned data
}
};
xmlhttp.open("GET", http://some.external.resource, true);
xmlhttp.send();
```

You may notice that we assign the `onreadystatechange` function before we even send the request. This is because assigning it later may result in a race condition in which the server responds before the function is attached to the ready state change. In this case, we've used an inline function to process the returned data. Because functions are first class citizens we can change this to look like the following:

```
let xmlhttp;
function requestData(){
  xmlhttp = new XMLHttpRequest()
  xmlhttp.onreadystatechange=processData;
  xmlhttp.open("GET", http://some.external.resource, true);
  xmlhttp.send();
}

function 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', function(json){
  //process returned data
});
```

In this case the boiler plate of dealing with ready state changes is handled for you. There is even convenience provided for you if the request for data fails:

```
$.ajax('http://some.external.resource',
  { success: function(json){
      //process returned data
    },
    error: function(){
      //process failure
    },
    dataType: "json"
});
```

In this case, we've passed an object into the `ajax` call which defines a number of properties. Amongst these properties are function callbacks for success and failure. This method of passing numerous functions into another suggests a great way of providing expansion points for classes.

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. We saw some treatment of functions in the previous chapter, *Chapter 5*, *Behavioral Patterns*, when passing function into the observer.

Implementation

In Westeros the tourism industry is almost non-extant. There are great difficulties with bandits killing tourists and tourists becoming entangled in regional conflicts. Nonetheless, some enterprising folks have started to advertise a grant tour of Westeros in which they will take those with the means on a tour of all the major attractions. From King's Landing to Eyrie, to the great mountains of Dorne - the tour will cover it all. In fact, a rather mathematically inclined member of the tourism board has taken to calling it a Hamiltonian tour as it visits everywhere once.

The `HamiltonianTour` class provides an options object which allows the definition of an options object. This object contains the various places to which a callback can be attached. In our case the interface for it would look something like the following:

```
export class HamiltonianTourOptions{
  onTourStart: Function;
  onEntryToAttraction: Function;
  onExitFromAttraction: Function;
  onTourCompletion: Function;
}
```

The full `HamiltonianTour` class looks like the following:

```
class HamiltonianTour {
  constructor(options) {
    this.options = options;
  }
  StartTour() {
    if (this.options.onTourStart && typeof
      (this.options.onTourStart) === "function")
      this.options.onTourStart();
      this.VisitAttraction("King's Landing");
      this.VisitAttraction("Winterfell");
      this.VisitAttraction("Mountains of Dorne");
      this.VisitAttraction("Eyrie");
    if (this.options.onTourCompletion && typeof
      (this.options.onTourCompletion) === "function")
      this.options.onTourCompletion();
  }
  VisitAttraction(AttractionName) {
    if (this.options.onEntryToAttraction && typeof
      (this.options.onEntryToAttraction) === "function")
      this.options.onEntryToAttraction(AttractionName);
      //do whatever one does in a Attraction
    if (this.options.onExitFromAttraction && typeof
      (this.options.onExitFromAttraction) === "function")
      this.options.onExitFromAttraction(AttractionName);
  }
}
```

You can see in the highlighted code how we check the options and then execute the callback as needed. This can be used by simply doing the following:

```
var tour = new HamiltonianTour({
  onEntryToAttraction: function(cityname){console.log("I'm
    delighted to be in " + cityname)}});
      tour.StartTour();
```

The output from running this code would be the following:

```
I'm delighted to be in King's Landing
I'm delighted to be in Winterfell
I'm delighted to be in Mountains of Dorne
I'm delighted to be in Eyrie
```

Passing functions is a great approach to solving a number of problems in JavaScript and tends to be used extensively by libraries such as jQuery and frameworks such as express. It is so commonly adopted that using it provides added barriers to your code's readability.

Filters and pipes

If you're at all familiar with the Unix command line or, to a lesser extent, the Windows command line, then you'll have probably made use of pipes. A pipe, which is represented by the `|` character is shorthand for "take the output of program A and put it into program B". This relatively simple idea makes the Unix command line incredibly powerful. For instance, if you wanted to list all the files in a directory and then sort them and filter for any which start with either the letters `b` or `g` and end with an `f` then the command might look like the following:

```
ls|sort|grep "^[gb].*f$"
```

The `ls` command lists all files and directories, the `sort` command sorts them, and the `grep` command matches file names against a regular expression. Running this command in the `etc` directory on an Ubuntu box in `/etc` would give a result which looks something like the following:

```
stimms@ubuntu1:/etc$ ls|sort|grep "^[gb].*f$"
blkid.conf
bogofilter.cf
brltty.conf
gai.conf
gconf
groff
gssapi_mech.conf
```

Some functional programming languages such as F# offer a special syntax for piping between functions. In F#, filtering a list for even numbers can be done in the following way:

```
[1..10] |>List.filter (fun n -> n% 2 = 0);;
```

This syntax is very nice-looking, especially when used for long chains of functions. As an example, taking a number, casting it to a float, square rooting it, and then rounding it would look like the following:

```
10.5 |> float |>Math.Sqrt |>Math.Round
```

This is a clearer syntax than the C-style syntax that would look more like the following:

```
Math.Round(Math.Sqrt((float)10.5))
```

Unfortunately, there is no ability to write pipes in JavaScript using a nifty F# style syntax, but we can still improve upon the normal method shown in the preceding code by using method chaining.

Everything in JavaScript is an object, which means that we can have some real fun adding functionality to existing objects to improve their look. Operating on collections of objects is a space in which functional programming provides some powerful features. Let's start by adding a simple filtering method to the array object. You can think of these queries as being like SQL database queries written in a functional fashion.

Implementation

We would like to provide a function that performs a match against each member of the array and returns a set of results:

```
Array.prototype.where = function (inclusionTest) {
  let results = [];
  for (let i = 0; i<this.length; i++) {
    if (inclusionTest(this[i]))
      results.push(this[i]);
  }
  return results;
};
```

The rather simple looking function allows us to quickly filter an array:

```
var items = [1,2,3,4,5,6,7,8,9,10];
items.where(function(thing){ return thing % 2 ==0;});
```

What we return is also an object, an array object in this case. We can continue to chain methods onto it like the following:

```
items.where(function(thing){ return thing % 2 ==0;})
  .where(function(thing){ return thing % 3 == 0;});
```

The result of this is an array containing only the number 6, as it is the only number between 1 and 10 which is both even and divisible by three. This method of returning a modified version of the original object without changing the original is known as a fluent interface. By not changing the original item array, we've introduced a small degree of immutability into our variables.

If we add another function to our library of array extensions, we can start to see how useful these pipes can be:

```
Array.prototype.select=function(projection){
  let results = [];
  for(let i = 0; i<this.length;i++){
    results.push(projection(this[i]));
  }
  return results;
};
```

This extension allows for projections of the original items based on an arbitrary projection function. Given a set of objects which contain IDs and names, we can use our fluent extensions to array to perform complex operations:

```
let children = [{ id: 1, Name: "Rob" },
{ id: 2, Name: "Sansa" },
{ id: 3, Name: "Arya" },
{ id: 4, Name: "Brandon" },
{ id: 5, Name: "Rickon" }];
let filteredChildren = children.where(function (x) {
  return x.id % 2 == 0;
}).select(function (x) {
  return x.Name;
});
```

This code will build a new array which contains only children with even IDs and instead of full objects, the array will contain only their names: `Sansa` and `Brandon`. For those familiar with .Net these functions may look very familiar. The **Language Integrated Queries** (**LINQ**) library on .Net provides similarly named functional inspired functions for the manipulation of collections.

Chaining functions in this manner can be both easier to understand and easier to build than alternatives: temporary variables are avoided and the code made terser. Consider the preceding example re-implemented using loops and temporary variables:

```
let children = [{ id: 1, Name: "Rob" },
{ id: 2, Name: "Sansa" },
{ id: 3, Name: "Arya" },
{ id: 4, Name: "Brandon" },
```

```
{ id: 5, Name: "Rickon" }];
let evenIds = [];
for(let i=0; i<children.length;i++)
{
  if(children[i].id%2==0)
    evenIds.push(children[i]);
}
let names = [];
for(let i=0; i< evenIds.length;i++)
{
  names.push(evenIds[i].name);
}
```

A number of JavaScript libraries such as d3 are constructed to encourage this sort of programming. At first it seems like the code created following this convention is bad due to very long line length. I would argue that this is a function of line length not being a very good tool to measure complexity rather than an actual problem with the approach.

Accumulators

We've looked at some simple array functions which add filtering and pipes to arrays. Another useful tool is the accumulator. Accumulators aid in building up a single result by iterating over a collection. Many common operations such as summing up the elements of an array can be implemented using an accumulator instead of a loop.

Recursion is popular within functional programming languages and many of them actually offer an optimization called "tail call optimization". A language that supports this provides optimizations for functions using recursion in which the stack frame is reused. This is very efficient and can easily replace most loops. Details on whether tail call optimization is supported in any JavaScript interpreter are sketchy. For the most part it doesn't seem like it is but we can still make use of recursion.

The problem with `for` loops is that the control flow through the loop is mutable. Consider this rather easy-to-make mistake:

```
let result = "";
let multiArray = [[1,2,3], ["a", "b", "c"]];
for(vari=0; i<multiArray.length; i++)
  for(var j=0; i<multiArray[i].length; j++)
    result += multiArray[i][j];
```

Did you spot the error? It took me several attempts to get a working version of this code I could break. The problem is in the loop counter in the second loop, it should read as follows:

```
let result = "";
let multiArray = [[1,2,3], ["a", "b", "c"]];
for(let i=0; i<multiArray.length; i++)
  for(let j=0; j<multiArray[i].length; j++)
    result +=multiArray[i][j];
```

Obviously this could be somewhat mitigated through better variable naming but we would like to avoid the problem altogether.

Instead we can make use of an accumulator, a tool for combining multiple values from a collection into a single value. We've rather missed Westeros for a couple of patterns so let's get back to our mythical example land. Wars cost a great deal of money but fortunately there are a great number of peasants to pay taxes and finance the lords in their games for the throne.

Implementation

Our peasants are represented by a simple model which looks like the following:

```
let peasants = [
  {name: "Jory Cassel", taxesOwed: 11, bankBalance: 50},
  {name: "VardisEgen", taxesOwed: 15, bankBalance: 20}];
```

Over this set of peasants we have an accumulator which looks like the following:

```
TaxCollector.prototype.collect = function (items, value, projection) {
  if (items.length> 1)
    return projection(items[0]) + this.collect(items.slice(1), value,
      projection);
  return projection(items[0]);
};
```

This code takes a list of items, an accumulator value, and a function that projects the value to be integrated into the accumulation.

The projection function looks something like the following:

```
function (item) {
  return Math.min(item.moneyOwed, item.bankBalance);
}
```

In order to prime this function, we simply need to pass in an initial value for the accumulator along with the array and projection. The priming value will vary but more often than not it will be an identity; an empty string in the case of a string accumulator and a 0 or 1 in the case of mathematical ones.

Each pass through the accumulator shrinks the size of the array over which we are operating. All this is done without a single mutable variable.

The inner accumulation can really be any function you like: string appending, addition, or something more complicated. The accumulator is somewhat like the visitor pattern except that modifying values in the collection inside an accumulator is frowned upon. Remember that functional programming is side-effect-free.

Memoization

Not to be confused with memorization, memoization is a specific term for retaining a number of previously calculated values from a function.

As we saw earlier, side-effect-free functions can be called multiple times without causing problems. The corollary to this is that a function can also be called fewer times than needed. Consider an expensive function which does some complex or, at least, time-consuming math. We know that the result of the function is entirely predicated on the inputs to the function. So the same inputs will always produce the same outputs. Why, then, would we need to call the function multiple times? If we saved the output of the function, we could retrieve that instead of redoing the time-consuming math.

Trading off space for time is a classic computing science problem. By caching the result, we make the application faster but we will consume more memory. Deciding when to perform caching and when to simply recalculate the result is a difficult problem.

Implementation

In the land of Westeros, learned men, known as Maesters, have long had a fascination with a sequence of numbers which seems to reappear a great deal in the natural world. In a strange coincidence they call this sequence the Fibonacci sequence. It is defined by adding the two previous terms in the sequence to get the next one. The sequence is bootstrapped by defining the first few terms as 0, 1, 1. So to get the next term we would simply add 1 and 1 to get 2. The next term would add 2 and 1 to get 3 and so forth. Finding an arbitrary member of the sequence requires finding the two previous members, so it can end up being a bit of calculation.

In our world we have discovered a closed form that avoids much of this calculation but in Westeros no such discovery has been made.

A naïve approach is to simply calculate every term like so:

```
let Fibonacci = (function () {
  function Fibonacci() {
  }
  Fibonacci.prototype.NaieveFib = function (n) {
    if (n == 0)
      return 0;
    if (n <= 2)
      return 1;
    return this.NaieveFib(n - 1) + this.NaieveFib(n - 2);
  };
  return Fibonacci;
})();
```

This solution works very quickly for small numbers such as 10. However, for larger numbers, say greater than 40, there is a substantial slow-down. This is because the base case is called 102,334,155 times.

Let's see if we can improve things by memoizing some values:

```
let Fibonacci = (function () {
  function Fibonacci() {
    this.memoizedValues = [];
  }

  Fibonacci.prototype.MemetoFib = function (n) {
    if (n == 0)
      return 0;
    if (n <= 2)
      return 1;
    if (!this. memoizedValues[n])
      this. memoizedValues[n] = this.MemetoFib(n - 1) +
        this.MemetoFib(n - 2);
    return this. memoizedValues[n];
  };
  return Fibonacci;
})();
```

We have just memoized every item we encounter. As it turns out for this algorithm we store *n+1* items, which is a pretty good trade-off. Without memoization, calculating the 40th fibonacci number took 963ms while the memoization version took only 11ms. The difference is far more pronounced when the functions become more complex to calculate. Fibonacci of 140 took 12 ms for the memoization version while the naïve version took… well, it is has been a day and it is still running.

The best part of this memoization is that subsequent calls to the function with the same parameter will be lightning-fast as the result is already computed.

In our example only a very small cache was needed. In more complex examples it is difficult to know how large a cache should be or how frequently a value will need to be recomputed. Ideally your cache will be large enough that there will always be room to put more results in. However, this may not be realistic and tough decisions will need to be made about which members of the cache should be removed to save space. There is a plethora of methods for performing cache invalidation. It has been said that cache invalidation is one of the toughest problems in computing science, the reason being that we're effectively trying to predict the future. If anybody has perfected a method of telling the future, it is likely they are applying their skills in a more important domain than cache invalidation. Two options are to prey on the least recently used member of the cache or the least frequently used member. It is possible that the shape of the problem may dictate a better strategy.

Memoization is a fantastic tool for speeding up calculations which need to be performed multiple times or even calculations which have common sub-calculations. One can consider memoization as just a special case of caching, which is a commonly used technique when building web servers or browsers. It is certainly worthwhile exploring in more complex JavaScript applications.

Immutability

One of the cornerstones of functional programming is that so called variables can be assigned only once. This is known as immutability. ECMAScript 2015 supports a new keyword, `const`. The `const` keyword can be used in the same way as `var` except that variables assigned with `const` will be immutable. For instance, the following code shows a variable and a constant that are both manipulated in the same way:

```
let numberOfQueens = 1;
const numberOfKings = 1;
numberOfQueens++;
numberOfKings++;
console.log(numberOfQueens);
console.log(numberOfKings);
```

The output of running this is the following:

```
2
1
```

As you can see, the results for the constant and variable are different.

If you're using an older browser without support, then `const` won't be available to you. A possible workaround is to make use of the `Object.freeze` functionality which is more widely adopted:

```
let consts = Object.freeze({ pi : 3.141});
consts.pi = 7;
console.log(consts.pi);//outputs 3.141
```

As you can see, the syntax here is not very user-friendly. Also an issue is that attempting to assign to an already assigned `const` simply fails silently instead of throwing an error. Failing silently in this fashion is not at all a desirable behavior; a full exception should be thrown. If you enable strict mode, a more rigorous parsing mode is added in ECMAScript 5, and an exception is actually thrown:

```
"use strict";
var consts = Object.freeze({ pi : 3.141});
consts.pi = 7;
```

The preceding code will throw the following error:

```
consts.pi = 7;
          ^
TypeError: Cannot assign to read only property 'pi' of #<Object>
```

An alternative is the `object.Create` syntax we spoke about earlier. When creating properties on the object, one can specify `writable: false` to make the property immutable:

```
var t = Object.create(Object.prototype,
{ value: { writable: false,
  value: 10}
});
t.value = 7;
console.log(t.value);//prints 10
```

However, even in strict mode no exception is thrown when attempting to write to a non-writable property. Thus I would claim that the `const` keyword is not perfect for implementing immutable objects. You're better off using freeze.

Lazy instantiation

If you go into a higher-end coffee shop and place an order for some overly complex beverage (Grande Chai Tea Latte, 3 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 which orders were going to come in that day, they would still not make all the beverages up front. First, because it would result in a large number of ruined, cold beverages, and second, 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 crafting 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 which is expensive to create; that is to say that 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 actually comes to pick up the bread:

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

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

```
var Bakery = (function () {
  function Bakery() {
    this.requiredBreads = [];
  }
  Bakery.prototype.orderBreadType = function (breadType) {
    this.requiredBreads.push(breadType);
  };
```

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

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

```
pickUpBread(breadType) {
  console.log("Picup of bread " + breadType + " requested");
  if (!this.breads) {
    this.createBreads();
  }
  for (var i = 0; i < this.breads.length; i++) {
    if (this.breads[i].breadType == breadType)
      return this.breads[i];
  }
}
createBreads() {
  this.breads = [];
  for (var i = 0; i < this.requiredBreads.length; i++) {
    this.breads.push(new Bread(this.requiredBreads[i]));
  }
}
```

Here we call a series of operations:

```
let 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:

```
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 which 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 is filled in. You can think of the result as being lazily instantiated. We'll look at promises and promise libraries in more detail in *Chapter 9*, *Web Patterns*.

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

Hints and tips

Although callbacks are the standard way of dealing with asynchronous methods in JavaScript they can get out of hand easily. There are a number of approaches to solving this spaghetti code: promise libraries provide a more fluent way of handling callbacks and future versions of JavaScript may adopt an approach similar to the C# `async/await` syntax.

I really like accumulators but they can be inefficient in terms of memory use. The lack of tail recursion means that each pass through adds another stack frame, so this approach may result in memory pressure. All things are a trade-off in this case between memory and code maintainability.

Summary

JavaScript is not a functional programming language. That is not to say that it isn't possible to apply some of the 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.

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 them as a style pattern.

In the next chapter we'll look at a number of patterns for dividing the logic and presentation in applications. These patterns have become more important as JavaScript applications have grown.

7
Reactive Programming

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 – just like the stream that Newton supposedly observed. Given a starting position and a stream of events it should be possible to figure out the state of the system. Indeed, this is the idea behind using an event store. Instead of keeping the final state of an aggregate in a database we instead keep track of all the events which have been applied to that aggregate. 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. For example, a disconnected system, like a cell phone application when the phone isn't connected to the network, which uses an event store can be merged with other events much more easily than simply keeping the end state. 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.

In this chapter we'll cover the following topics:

- Application state changes
- Streams
- Filtering streams
- Merging streams
- Streams for multiplexing

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

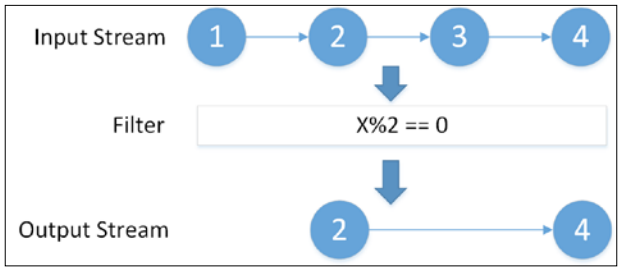
The easiest way to think of an event stream is not to think of the streams you've probably used before in programming, input reader streams, but to think of arrays. Let's say that you have an array with a series of numbers in it:

```
[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. If you're looking to see how to implement a basic stream, then you can base it on the observer pattern outlined in *Chapter 5*, *Behavioral Patterns*.

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:

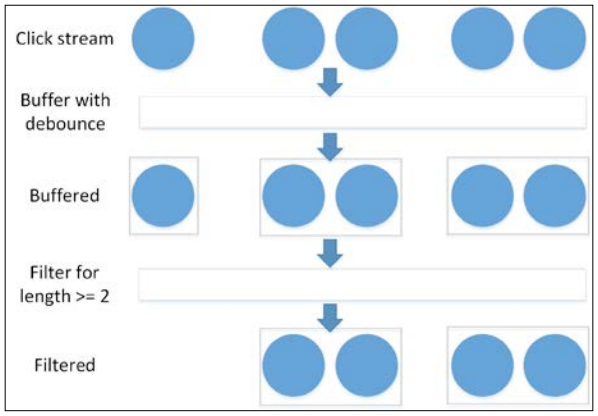
```
var incrementSubscription = source.subscribe(() => counter++);
var subscription = source.filter(x=>counter%2==0).subscribe(function
(e) {
  output.innerHTML = "Clicked " + counter + " time" +
    (counter > 1 ? "s" : "");
});
```

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:

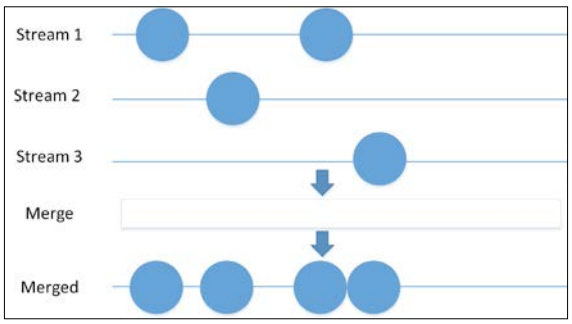
```
Var counter = 0;
generateDeathsStream(deaths).subscribe((item) => { counter++ });
```

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.

Hints and tips

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.

8
Application Patterns

Thus far we have spent a great deal of time examining patterns that are used to solve local problems, that is; problems that span only a handful of classes and not the whole application. These patterns have been narrow in scope. They frequently only relate to two or three classes and might be used but a single time in any given application. As you can imagine there are also larger scale patterns that are applicable to the application as a whole. You might think of "toolbar" as a general pattern that is used in many places in an application. What's more, it is a pattern that is used in a great number of applications to give them a similar look and feel. Patterns can help guide how the whole application is assembled.

In this chapter we're going to look at a family of patterns which I've taken to calling the MV* family. This family includes MVC, MVVM, MVP, and even PAC. Just like their names, the patterns themselves are pretty similar. The chapter will cover each of these patterns and show how, or if, we can apply them to JavaScript. We'll also pay special attention to how the patterns differ from one another. By the end of the chapter you should be able to thrill guests at a cocktail party with an explanation of the nuances of MVP versus MVC.

The topics covered will be as follows:

- History of Model View patterns
- Model View Controller
- Model View Presenter
- Model View ViewModel

First, some history

Separating concerns inside an application is a very important idea. We live in a complex and ever-changing world. This means that not only is it nearly impossible to formulate a computer program which works in exactly the way users want, but that what users want is an ever-shifting maze. Couple this with the fact that an ideal program for user A is totally different from an ideal program for user B and we're guaranteed to end up in a mess. Our applications need to change as frequently as we change our socks: at least once a year.

Layering an application and maintaining modularity reduces the impact of a change. The less each layer knows about the other layers the better. Maintaining simple interfaces between the layers reduces the chances that a change to one layer will percolate to another layer.

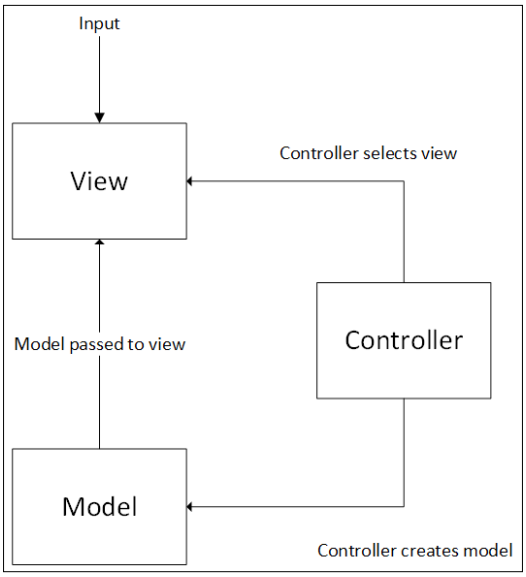
If you've ever taken a close look at a high quality piece of nylon (from a hot air balloon, parachute, or expensive jacket) you may have noticed that that the fabric seems to form tiny squares. This is because, every few millimeters, a thick reinforcing thread is added to the weave to form a crosshatch pattern. If the fabric is ripped, then the rip will be stopped or at least slowed by the reinforcement. This limits the damage to a small area and prevents it from spreading.

Layers and modules in an application are exactly the same: they limit the spread of damage from a change.

In the early chapters of this book, we talked a bit about the seminal language, Smalltalk. It was the language which made classes famous. Like many of these patterns, the original MV* pattern, **Model View Controller** (**MVC**), was used long before it was ever identified. Although difficult to prove it seems that MVC was originally suggested in the late 1970s by Trygve Reenskaug, a Norwegian computer scientist, during a visit to the legendary Xerox PARC. Through the 1980s the pattern became heavily used in Smalltalk applications. However, it was not until 1988 that the pattern was more formally documented in an article entitled, *A Cookbook for Using the Model-View-Controller User Interface Paradigm* by Glenn E. Krasner and Stephen T. Pope.

Model View Controller

MVC is a pattern that is useful for creating rich, interactive user interfaces: just the sort of interfaces which are becoming more and more popular on the web. The astute amongst you will have already figured out that the pattern is made up of three major components: model, view, and controller. You can see how information flows between the components in this illustration:



The preceding diagram shows the relationship between the three components in MVC.

The model contains the state of the program. In many applications this model is contained in some form, in a database. The model may be rehydrated from a persistent store such as the database or it can be transient. Ideally the model is the only mutable part of the pattern. Neither the view nor the controller has any state associated with them.

For a simple login screen the model might look like the following:

```
class LoginModel{
  UserName: string;
  Password: string;
  RememberMe: bool;
  LoginSuccessful: bool;
  LoginErrorMessage: string;
}
```

You'll notice that not only do we have fields for the inputs shown to the user but also for the state of the login. This would not be apparent to the user but it is still part of the application state.

The model is usually modeled as a simple container for information. Typically, there are no real functions in the model. It simply contains data fields and may also contain validation. In some implementations of the MVC pattern the model also contains meta-data about the fields such as validation rules.



The Naked Object pattern is a deviation from the typical MVC pattern. It augments the model with extensive business information as well as hits about the display and editing of data. It even contains methods for persisting the model to storage.

The views in the Naked Object pattern are generated from these models automatically. The controller is also automatically generated by examining the model. This centralizes the logic for displaying and manipulating application states and saves the developer from having to write their own views and controllers. So while the view and controller still exist, they are not actual objects but are dynamically created from the model.

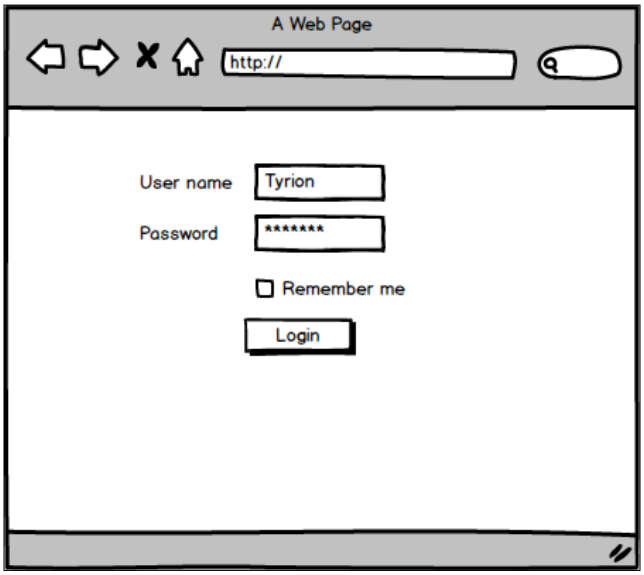
Several systems have been successfully deployed using this pattern. Some criticism has emerged around the ability to generate an attractive user interface from just the models as well as how to properly coordinate multiple views.

In a foreword to the PhD thesis, *presenting Naked Objects* by Reenskaug, he suggests that the naked objects pattern is actually closer to his original vision for MVC than most of the derivations of MVC in the wild.

Updates to the model are communicated to the view whenever the state changes. This is typically done through the use of an observer pattern. The model does not typically know about either the controller or the view. The first is simply the thing telling it to change and the second is only updated through the observer pattern so the model doesn't have direct knowledge of it.

The view does pretty much what you would expect: communicate the model state to a target. I hesitate to suggest that the view must be a visual or graphical representation of the model as frequently the view is being communicated to another computer and may be in the form of XML, JSON, or some other data format. In most cases, especially those related to JavaScript, the view will be a graphical object. In a web context this will typically be HTML which is rendered by the browser. JavaScript is also gaining popularity on phones and on the desktop, so the view could also be a screen on a phone or on the desktop.

The view for the model presented in the preceding paragraph might look like the following figure:



In cases, where the observer pattern is not used, then the view may poll the model at some interval looking for changes. In this case the view may have to keep a representation of the state itself or at least a version number. It is important that the view not unilaterally update this state without passing the updates to the controller, otherwise the model and the copy in the view will get out of sync.

Finally, the state of the model is updated by the controller. The controller usually contains all the logic and business rules for updating fields on the model. A simple controller for our login page might look like the following code:

```
class LoginController {
  constructor(model) {
    this.model = model;
  }
  Login(userName, password, rememberMe) {
    this.model.UserName = userName;
    this.model.Password = password;
    this.model.RememberMe = rememberMe;
    if (this.checkPassword(userName, password))
      this.model.LoginSuccessful;
    else {
```

```
            this.model.LoginSuccessful = false;
            this.model.LoginErrorMessage = "Incorrect username or password";
        }
    }
};
```

The controller knows about the existence of the model and is typically aware of the view's existence as well. It coordinates the two of them. A controller may be responsible for initializing more than one view. For instance, a single controller may provide a list view of all the instances of a model as well as a view that simply provides details. In many systems a controller will have create, read, update, and delete (CRUD) operations on it that work over a model. The controller is responsible for choosing the correct view and for wiring up the communication between the model and the view.

When there is a need for a change to the application then the location of the code should be immediately apparent. For example:

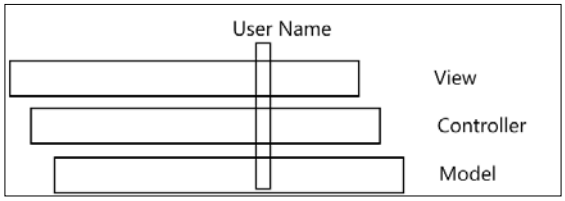
Change	**Location**
Elements don't appear well spaced on the screen, change spacing.	View
No users are able to log in due to a logical error in password validation.	Controller
New field to be added.	All layers



Presentation-Abstraction-Control (**PAC**) is another pattern that makes use of a triad of components. In this case its goal is to describe a hierarchy of encapsulated triples that more closely match how we think of the world. The control, similar to an MVC controller, passes interactions up in the hierarchy of encapsulated components allowing for information to flow between components. The abstraction is similar to a model but may represent only a few fields that are important for that specific PAC instead of the entire model. Finally, the presentation is effectively the same as a view.

The hierarchical nature of PAC allows for parallel processing of the components, meaning that it can be a powerful tool in today's multiprocessor systems.

You might notice that the last one there requires a change in all layers of the application. These multiple locations for responsibility are something that the Naked Objects pattern attempts to address by dynamically creating views and controllers. The MVC pattern splits code into locations by dividing the code by its role in user interaction. This means that a single data field lives in all the layers as is shown in this picture:



Some might call this a cross-cutting concern but really it doesn't span a sufficient amount of the application to be called such. Data access and logging are cross-cutting concerns as they are pervasive and difficult to centralize. This pervasion of a field through the different layers is really not a major problem. However, if it is bugging you then you might be an ideal candidate for using the Naked Objects pattern.

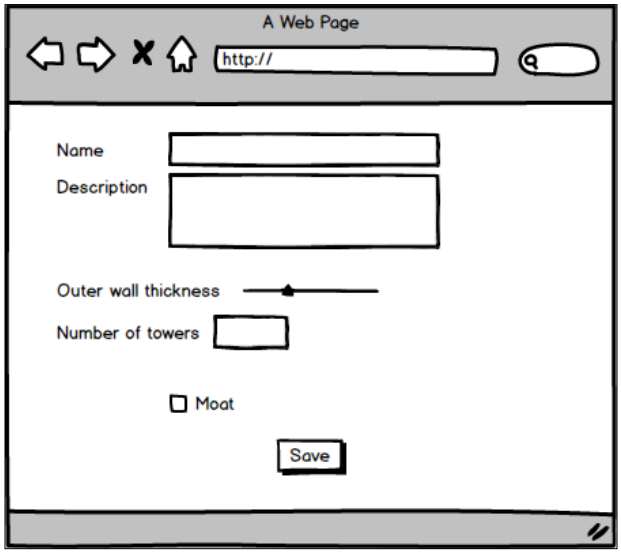
Let's step into building some code to represent a MVC in JavaScript.

MVC code

Let's start with a simple scenario for which we can apply MVC. Unfortunately, Westeros has very few computers, likely due to the lack of electricity. Thus applying application structuring patterns using Westeros as an example is difficult. Sadly we'll have to take a step back and talk about an application which controls Westeros. Let's assume it to be a web application and implement the entirety of MVC on the client side.

It is possible to implement MVC by splitting the model, view and controller between client and server. Typically, the controller would sit on the server and provide an API which is known by the view. The model serves as a communication method both to the view which resides on the web browser and to the data store, likely a database of some form. It is also common that the controller be split between the server and the client in cases where some additional control is required on the client.

In our example we would like to create a screen that controls the properties of a castle. Fortunately, you're lucky that this is not a book on designing user interfaces with HTML as I would certainly fail. We'll stick to a picture in place of the HTML:



For the most part, the view simply provides a set of controls and data for the end user. In this example the view will need to know how to call the save function on the controller. Let's set that up:

```
class CreateCastleView {
  constructor(document, controller, model, validationResult) {
    this.document = document;
    this.controller = controller;
    this.model = model;
    this.validationResult = validationResult;
    this.document.getElementById("saveButton").
      addEventListener("click", () => this.saveCastle());
    this.document.getElementById("castleName").value = model.name;
    this.document.getElementById("description").value = model.
     description;
    this.document.getElementById("outerWallThickness").value = model.
     outerWallThickness;
    this.document.getElementById("numberOfTowers").value = model.
     numberOfTowers;
    this.document.getElementById("moat").value = model.moat;
  }
  saveCastle() {
```

```
      var data = {
        name: this.document.getElementById("castleName").value,
        description: this.document.getElementById("description").value,
        outerWallThickness: this.document.getElementById("outerWallThic
          kness").value,
        numberOfTowers: this.document.getElementById("numberOfTowers").
         value,
        moat: this.document.getElementById("moat").value
      };
      this.controller.saveCastle(data);
    }
  }
```

You'll notice that the constructor for this view contains both a reference to the document and to the controller. The document contains both HTML and styling, provided by CSS. We can get away with not passing in a reference to the document but injecting the document in this fashion allows for easier testability. We'll look at testability more in a later chapter. It also permits reusing the view multiple times on a single page without worrying about the two instances conflicting.

The constructor also contains a reference to the model which is used to add data to fields on the page as needed. Finally, the constructor also references a collection of errors. This allows for validation errors from the controller to be passed back to the view to be handled. We have set the validation result to be a wrapped collection that looks something like the following:

```
class ValidationResult{
  public IsValid: boolean;
  public Errors: Array<String>;
  public constructor(){
    this.Errors = new Array<String>();
  }
}
```

The only functionality here is that the button's `onclick` method is bound to calling save on the controller. Instead of passing in a large number of parameters to the `saveCastle` function on the controller, we build a lightweight object and pass that in. This makes the code more readable, especially in cases where some of the parameters are optional. No real work is done in the view and all the input goes directly to the controller.

The controller contains the real functionality of the application:

```
class Controller {
  constructor(document) {
    this.document = document;
  }
  createCastle() {
    this.setView(new CreateCastleView(this.document, this));
  }
  saveCastle(data) {
    var validationResult = this.validate(data);
    if (validationResult.IsValid) {
      //save castle to storage
      this.saveCastleSuccess(data);
    }
    else {
      this.setView(new CreateCastleView(this.document, this, data,
        validationResult));
    }
  }
  saveCastleSuccess(data) {
    this.setView(new CreateCastleSuccess(this.document, this, data));
  }
  setView(view) {
    //send the view to the browser
  }
  validate(model) {
    var validationResult = new validationResult();
    if (!model.name || model.name === "") {
      validationResult.IsValid = false;
      validationResult.Errors.push("Name is required");
    }
    return;
  }
}
```

The controller here does a number of things. The first thing is that it has a `setView` function which instructs the browser to set the given view as the current one. This is likely done through the use of a template. The mechanics of how that works are not important to the pattern so I'll leave that up to your imagination.

Next, the controller implements a `validate` method. This method checks to make sure that the model is valid. Some validation may be performed on the client, such as testing the format of a postal code, but other validation requires a server trip. If a username must be unique then there is no reasonable way to test that on the client without communicating with the server. In some cases, the validation functionality may exist on the model rather than in the controller.

Methods for setting up various different views are also found in the controller. In this case we have a bit of a workflow with a view for creating a castle then views for both success and failure. The failure case just returns the same view with a collection of validation errors attached to it. The success case returns a whole new view.

The logic to save the model to some sort of persistent storage is also located in the controller. Again the implementation of this is less important than to see that the logic for communicating with the storage system is located in the controller.

The final letter in MVC is the model. In this case, it is a very light weight one:

```
class Model {
  constructor(name, description, outerWallThickness, numberOfTowers,
    moat) {
    this.name = name;
    this.description = description;
    this.outerWallThickness = outerWallThickness;
    this.numberOfTowers = numberOfTowers;
    this.moat = moat;
  }
}
```

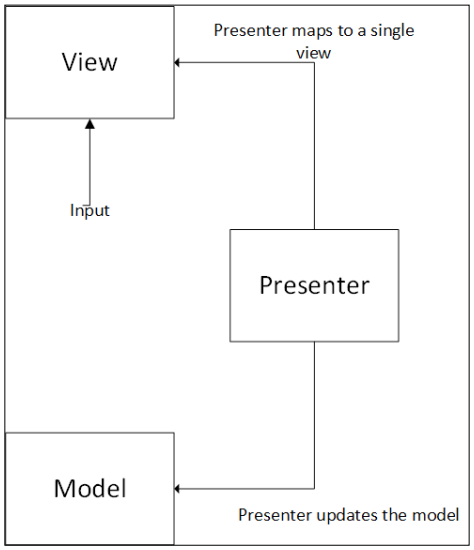
As you can see, all it does is keep track of the variables that make up the state of the application.

Concerns are well separated in this pattern allowing for changes to be made with relative ease.

Model View Presenter

The **Model View Presenter** (**MVP**) pattern is very similar to MVC. It is a fairly well known pattern in the Microsoft world and is generally used to structure WPF and Silverlight applications. It can be used in pure JavaScript as well. The key difference comes down to how the different parts of the system interact and where their responsibility ends.

The first difference is that, with the presenter, there is a one to one mapping between presenter and view. This means that the logic that existed in the controller in the MVC pattern, which selected the correct view to render, doesn't exist. Or rather it exists at a higher level outside the concern of the pattern. The selection of the correct presenter may be handled by a routing tool. Such a router will examine the parameters and provide the best choice for the presenter. The flow of information in the MVP pattern can be seen here:



The presenter is aware of both the view and the model but the view is unaware of the model and the model unaware of the view. All communication is passed through the presenter.

The presenter pattern is often characterized by a great deal of two-way dispatch. A click will fire in the presenter and then the presenter will update the model with the change and then the view. The preceding diagram suggests that the input first passes through the view. In a passive version of the MVP pattern, the view has little to no interaction with the messages as they are passed onto the presenter. However, there is also a variation called active MVP that allows the view to contain some additional logic.

This active version of MVP can be more useful for web situations. It permits adding validation and other simple logic to the view. This reduces the number of requests that need to pass from the client back to the web server.

Let's update our existing code sample to use MVP instead of MVC.

MVP code

Let's start again with the view:

```
class CreateCastleView {
  constructor(document, presenter) {
    this.document = document;
    this.presenter = presenter;
    this.document.getElementById("saveButton").
      addEventListener("click", this.saveCastle);
  }
  setCastleName(name) {
    this.document.getElementById("castleName").value = name;
  }
  getCastleName() {
    return this.document.getElementById("castleName").value;
  }
  setDescription(description) {
    this.document.getElementById("description").value = description;
  }
  getDescription() {
    return this.document.getElementById("description").value;
  }
  setOuterWallThickness(outerWallThickness) {
    this.document.getElementById("outerWallThickness").value =
     outerWallThickness;
  }
  getOuterWallThickness() {
    return this.document.getElementById("outerWallThickness").value;
  }
  setNumberOfTowers(numberOfTowers) {
    this.document.getElementById("numberOfTowers").value =
     numberOfTowers;
  }
  getNumberOfTowers() {
    return parseInt(this.document.getElementById("numberOfTowers").
     value);
  }
  setMoat(moat) {
    this.document.getElementById("moat").value = moat;
  }
  getMoat() {
    return this.document.getElementById("moat").value;
  }
  setValid(validationResult) {
  }
  saveCastle() {
```

```
      this.presenter.saveCastle();
    }
  }
```

As you can see the constructor for the view no longer takes a reference to the model. This is because the view in MVP doesn't have any idea about what model is being used. That information is abstracted away by the presenter. The reference to presenter remains in the constructor to allow sending messages back to the presenter.

Without the model there is an increase in the number of public setter and getter methods. These setters allow the presenter to make updates to the state of the view. The getters provide an abstraction over how the view stores the state and gives the presenter a way to get the information. The `saveCastle` function no longer passes any values to the presenter.

The presenter's code looks like the following:

```
class CreateCastlePresenter {
  constructor(document) {
    this.document = document;
    this.model = new CreateCastleModel();
    this.view = new CreateCastleView(document, this);
  }
  saveCastle() {
    var data = {
      name: this.view.getCastleName(),
      description: this.view.getDescription(),
      outerWallThickness: this.view.getOuterWallThickness(),
      numberOfTowers: this.view.getNumberOfTowers(),
      moat: this.view.getMoat()
    };
    var validationResult = this.validate(data);
    if (validationResult.IsValid) {
      //write to the model
      this.saveCastleSuccess(data);
    }
    else {
      this.view.setValid(validationResult);
    }
  }
  saveCastleSuccess(data) {
    //redirect to different presenter
  }
  validate(model) {
    var validationResult = new validationResult();
    if (!model.name || model.name === "") {
      validationResult.IsValid = false;
```

```
            validationResult.Errors.push("Name is required");
        }
        return;
    }
}
```

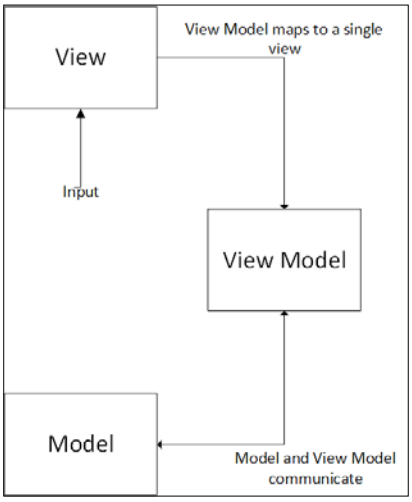
You can see that the view is now referenced in a persistent fashion in the presenter. The `saveCastle` method calls into the view to get its values. However, the presenter does make sure to use the public methods of the view instead of referencing the document directly. The `saveCastle` method updates the model. If there are validation errors, then it will call back into the view to update the `IsValid` flag. This is an example of the double dispatch I mentioned earlier.

Finally, the model remains unchanged from before. We've kept the validation logic in the presenter. At which level the validation is done, model or presenter, matters less than being consistent in where the validation is done through your application.

The MVP pattern is again a fairly useful pattern for building user interfaces. The larger separation between the view and the model creates a stricter API allowing for better adaptation to change. However, this comes at the expense of more code. With more code comes more opportunity for bugs.

Model View ViewModel

The final pattern we'll look at in this chapter is the Model View ViewModel pattern, more commonly known as MVVM. By now this sort of pattern should be getting quite familiar. Again you can see the flow of information between components in this illustration:



You can see here that many of the same constructs have returned but that the communication between them is somewhat different.

In this variation, what has previously been the controller and presenter is now the view model. Just like with MVC and MVP, the majority of the logic is held within the central component, in this case the view model. The model itself is actually very simple in MVVM. Typically, it acts as an envelope that just holds data. Validation is done within the view model.

Just like with MVP, the view is totally unaware of the existence of the model. The difference is that, with MVP, the view was aware that it was talking to some intermediate class. It called methods rather than simply setting values. In MVVM the view believes that the view model is its view. Instead of calling operations like `saveCastle` and passing in data or waiting for data to be requested, the view updates fields on the view model as they change. In effect, the fields on the view are bound to the view model. The view model may proxy these values through to the model or wait until a commit-like operation like save is called to pass the data along.

Equally, changes to the view model should be reflected at once in the view. A single view may have a number of view models. Each of these view models may push updates to the view or have changes pushed to it via the view.

Let's take a look at a really rudimentary implementation of this and then we'll discuss how to make it better.

MVVM code

The naïve view implementation is, frankly, a huge mess:

```
var CreateCastleView = (function () {
  function CreateCastleView(document, viewModel) {
    this.document = document;
    this.viewModel = viewModel;
    var _this = this;
    this.document.getElementById("saveButton").
      addEventListener("click", function () {
    return _this.saveCastle();
  });
  this.document.getElementById("name").addEventListener("change",
    this.nameChangedInView);
  this.document.getElementById("description").
   addEventListener("change", this.descriptionChangedInView);
  this.document.getElementById("outerWallThickness").
   addEventListener("change", this.outerWallThicknessChangedInView);
```

```
    this.document.getElementById("numberOfTowers").
      addEventListener("change", this.numberOfTowersChangedInView);
    this.document.getElementById("moat").addEventListener("change",
      this.moatChangedInView);
  }
  CreateCastleView.prototype.nameChangedInView = function (name) {
    this.viewModel.nameChangedInView(name);
  };

  CreateCastleView.prototype.nameChangedInViewModel = function (name) {
    this.document.getElementById("name").value = name;
  };
  //snipped more of the same
  CreateCastleView.prototype.isValidChangedInViewModel = function
    (validationResult) {
    this.document.getElementById("validationWarning").innerHtml =
      validationResult.Errors;
    this.document.getElementById("validationWarning").className =
      "visible";
  };
  CreateCastleView.prototype.saveCastle = function () {
    this.viewModel.saveCastle();
  };
  return CreateCastleView;
  })();
  CastleDesign.CreateCastleView = CreateCastleView;
```

It is highly repetitive and each property must be proxied back to `ViewModel`. I've truncated most of this code but it adds up to a good 70 lines. The code inside the view model is equally terrible:

```
  var CreateCastleViewModel = (function () {
    function CreateCastleViewModel(document) {
      this.document = document;
      this.model = new CreateCastleModel();
      this.view = new CreateCastleView(document, this);
    }
    CreateCastleViewModel.prototype.nameChangedInView = function (name)
    {
      this.name = name;
    };

    CreateCastleViewModel.prototype.nameChangedInViewModel = function
      (name) {
```

```
    this.view.nameChangedInViewModel(name);
  };
  //snip
  CreateCastleViewModel.prototype.saveCastle = function () {
    var validationResult = this.validate();
    if (validationResult.IsValid) {
      //write to the model
      this.saveCastleSuccess();
    } else {
      this.view.isValidChangedInViewModel(validationResult);
    }
  };

  CreateCastleViewModel.prototype.saveCastleSuccess = function () {
    //do whatever is needed when save is successful.
    //Possibly update the view model
  };

  CreateCastleViewModel.prototype.validate = function () {
    var validationResult = new validationResult();
    if (!this.name || this.name === "") {
      validationResult.IsValid = false;
        validationResult.Errors.push("Name is required");
    }
    return;
  };
  return CreateCastleViewModel;
})();
```

One look at this code should send you running for the hills. It is set up in a way that will encourage copy and paste programming: a fantastic way to introduce errors into a code base. I sure hope there is a better way to transfer changes between the model and the view.

A better way to transfer changes between the model and the view

It may not have escaped your notice that there are a number of MVVM-style frameworks for JavaScript in the wild. Obviously they would not have been readily adopted if they followed the approach that we described in the preceding section. Instead they follow one of two different approaches.

The first approach is known as dirty checking. In this approach, after every interaction with the view model we loop over all of its properties looking for changes. When changes are found, the related value in the view is updated with the new value. For changes to values in the view change, actions are attached to all the controls. These then trigger updates to the view model.

This approach can be slow for large models as it is expensive to iterate over all the properties of a large model. The number of things which can cause a model to change is high and there is no real way to tell if a distant field in a model has been changed by changing another without going and validating it. On the upside, dirty checking allows you to use plain old JavaScript objects. There is no need to write your code any differently than before. The same is not true of the other approach: container objects.

With a container object a special interface is provided to wrap existing objects so that changes to the object may be directly observed. Basically this is an application of the observer pattern but applied dynamically so the underlying object has no idea it is being observed. The spy pattern, perhaps?

An example might be helpful here. Let us say that we have the model object we've been using up until now:

```
var CreateCastleModel = (function () {
  function CreateCastleModel(name, description, outerWallThickness,
    numberOfTowers, moat) {
    this.name = name;
    this.description = description;
    this.outerWallThickness = outerWallThickness;
    this.numberOfTowers = numberOfTowers;
    this.moat = moat;
  }
  return CreateCastleModel;
})();
```

Then, instead of `model.name` being a simple string, we would wrap some function around it. In the case of the Knockout library this would look like the following:

```
var CreateCastleModel = (function () {
  function CreateCastleModel(name, description, outerWallThickness,
    numberOfTowers, moat) {
    this.name = ko.observable(name);
    this.description = ko.observable(description);
    this.outerWallThickness = ko.observable(outerWallThickness);
    this.numberOfTowers = ko.observable(numberOfTowers);
```

```
      this.moat = ko.observable(moat);
    }
    return CreateCastleModel;
})();
```

In the highlighted code, the various properties of the model are being wrapped with an observable. This means that they must now be accessed differently:

```
var model = new CreateCastleModel();
model.name("Winterfell"); //set
model.name(); //get
```

This approach obviously adds some friction to your code and makes changing frameworks quite involved.

Current MVVM frameworks are split on their approach to container objects versus dirty checking. AngularJS uses dirty checking while Backbone, Ember, and Knockout all make use of container objects. There is currently no clear winner.

Observing view changes

Fortunately, the pervasiveness of MV* patterns on the web and the difficulties with observing model changes has not gone unnoticed. You might be expecting me to say that this will be solved in ECMAScript-2015 as is my normal approach. Weirdly, the solution to all of this, `Object.observe`, is a feature under discussion for ECMAScript-2016. However, at the time of writing, at least one major browser already supports it.

It can be used like the following:

```
var model = { };
Object.observe(model, function(changes){
  changes.forEach(function(change) {
    console.log("A " + change.type + " occured on " +  change.name +
      ".");
    if(change.type=="update")
      console.log("\tOld value was " + change.oldValue );
  });
});
model.item = 7;
model.item = 8;
delete model.item;
```

Having this simple interface to monitor changes to objects removes much of the logic provided by large MV* frameworks. It will be easier to roll your own functionality for MV* and there may, in fact, be no need to use external frameworks.

Tips and tricks

The different layers of the various MV* patterns need not all be on the browser, nor do they all need to be written in JavaScript. Many popular frameworks allow for maintaining a model on the server and communicating with it using JSON.

`Object.observe` may not be available on all browsers yet, but there are polyfills that can be used to create a similar interface. The performance is not as good as the native implementation, but it is still usable.

Summary

Separating concerns to a number of layers ensures that changes to the application are isolated like a ripstop. The various MV* patterns allow for separating the concerns in a graphical application. The differences between the various patterns come down to how the responsibilities are separated and how information is communicated.

In the next chapter we'll look at a number of patterns and techniques to improve the experience of developing and deploying JavaScript to the Web.

9
Web Patterns

The rise of Node.js has proven that JavaScript has a place on web servers, even very high throughput servers. There is no denying that JavaScript's pedigree remains in the browser for client side programming.

In this chapter we're going to look at a number of patterns to improve the performance and usefulness of JavaScript on the client. I'm not sure that all of these can be thought of as patterns in the strictest sense. They are, however, important and worth mentioning.

The concepts we'll examine in this chapter are as follows:

- Sending JavaScript
- Plugins
- Multithreading
- Circuit breaker pattern
- Back-off
- Promises

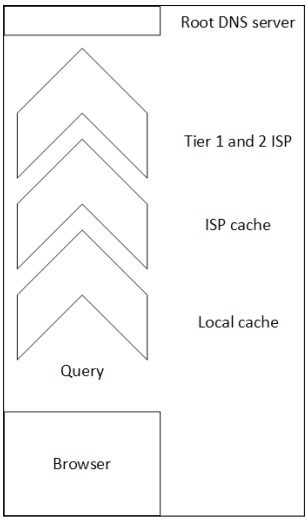
Sending JavaScript

Communicating JavaScript to the client seems to be a simple proposition: so long as you can get the code to the client it doesn't matter how that happens, right? Well not exactly. There are actually a number of things that need to be considered when sending JavaScript to the browser.

Combining files

Way back in *Chapter 2*, *Organizing Code*, we looked at how to build objects using JavaScript, although opinions on this vary. I consider it to be good form to have a one-class-to-one-file organization of my JavaScript or really any of my object oriented code. By doing this, it makes finding code easy. Nobody needs to hunt through a 9000 line long JavaScript file to locate that one method. It also allows for a hierarchy to be established again allowing for good code organization. However, good organization for a developer is not necessarily good organization for a computer. In our case having a lot of small files is actually highly detrimental. To understand why, you need to know a little bit about how browsers ask for and receive content.

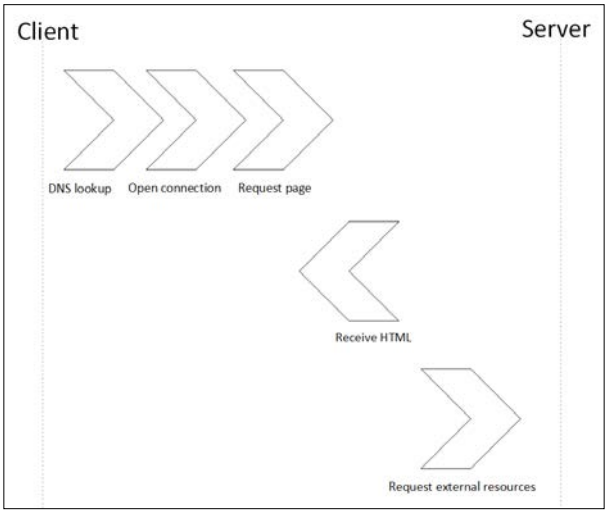
When you type a URL into the address bar of a browser and hit *Enter*, a cascading series of events happens. The first thing is that the browser will ask the operating system to resolve the website name to an IP address. On both Windows and Linux (and OSX) the standard C library function `gethostbyname` is used. This function will check the local DNS cache to see if the mapping from name to address is already known. If it is, then that information is returned. If not, then the computer makes a request to the DNS server one step up from it. Typically, this is the DNS server provided by the ISP but on a larger network it could also be a local DNS server. The path of a query between DNS servers can be seen here:



If a record doesn't exist on that server then the request is propagated up a chain of DNS servers in an attempt to find one that knows about the domain. Eventually the propagation stops at the root servers. These root servers are the stopping point for queries – if they don't know who is responsible for DNS information for a domain then the lookup is deemed to have failed.

Once the browser has an address for the site it opens up a connection and sends a request for the document. If no document is provided, then a / is sent. Should the connection be a secure one, then negotiation of SSL/TSL is performed at this time. There is some computational expense to setting up an encrypted connection but this is slowly being fixed.

The server will respond with a blob of HTML. As the browser receives this HTML it starts to process it; the browser does not wait for the entire HTML document to be downloaded before it goes to work. If the browser encounters a resource that is external to the HTML it will kick off a new request to open another connection to the web server and download that resource. The maximum number of connections to a single domain is limited so that the web server isn't flooded. It should also be mentioned that setting up a new connection to the web server carries overhead. The flow of data between a web client and server can be seen in this illustration:



Connections to the web server should be limited to avoid paying the connection setup costs repeatedly. This brings us to our first concept: combining files.

If you've followed the advice to leverage namespaces and classes in your JavaScript, then putting all of your JavaScript together in a single file is a trivial step. One need only concatenate the files together and everything should continue to work as normal. Some minor care and attention may need to be paid to the order of inclusion, but not typically.

The previous code we've written has been pretty much one file per pattern. If there is a need for multiple patterns to be used, then we could simply concatenate the files together. For instance, the combined builder and factory method patterns might look like the following:

```
var Westeros;
(function (Westeros) {
  (function (Religion) {
      …
  })(Westeros.Religion || (Westeros.Religion = {}));
  var Religion = Westeros.Religion;
})(Westeros || (Westeros = {}));
(function (Westeros) {
  var Tournament = (function () {
    function Tournament() {
  }
  return Tournament;
})();
Westeros.Tournament = Tournament;
…
})();
Westeros.Attendee = Attendee;
})(Westeros || (Westeros = {}));
```

The question may arise as to how much of your JavaScript should be combined and loaded at once. It is a surprisingly difficult question to answer. On one hand it is desirable to front load all the JavaScript for the entire site when users first arrive at the site. This means that users will pay a price initially but will not have to download any additional JavaScript as they travel about the site. This is because the browser will cache the script and reuse it instead of downloading it from the server again. However, if users only visit a small subset of the pages on the site then they will have loaded a great deal of JavaScript that was not needed.

On the other hand, splitting up the JavaScript means that additional page visits incur a penalty for retrieving additional JavaScript files. There is a sweet spot somewhere in the middle of these two approaches. Script can be organized into blocks that map to different sections of the website. This can be a place where using proper name spacing will come in handy once again. Each namespace can be combined into a single file and then loaded as users visit that part of the site.

In the end, the only approach that makes sense is to maintain statistics about how users move about the site. Based on this information an optimal strategy for finding the sweet spot can be established.

Minification

Combining JavaScript into a single file solves the problem of limiting the number of requests. However, each request may still be large. Again we come to a schism between what makes code fast and readable by humans and what makes it fast and readable by computers.

We humans like descriptive variable names, bountiful whitespace, and proper indentation. Computers don't care about descriptive names, whitespace, or proper indentation. In fact, these things increase the size of the file and thus decrease the speed at which the code can be read.

Minification is a compile step that transforms the human readable code into smaller, but equivalent, code. External variables' names remain the same, as the minifier has no way to know what other code may be relying on the variable names remaining unchanged.

As an example, if we start with the composite code from *Chapter 4*, *Structural Patterns*, the minified code looks like the following:

```
var Westros;(function(Westros){(function(Food){var
SimpleIngredient=(function(){function SimpleIngredient(name
,calories,ironContent,vitaminCContent){this.name=name;this.
calories=calories;this.ironContent=ironContent;this.
vitaminCContent=vitaminCContent}SimpleIngredient.prototype.
GetName=function(){return this.name};SimpleIngredient.prototype.
GetCalories=function(){return this.calories};SimpleIngredient.
prototype.GetIronContent=function(){return this.
ironContent};SimpleIngredient.prototype.GetVitaminCContent=function()
{return this.vitaminCContent};return SimpleIngredient})();Food.Sim
pleIngredient=SimpleIngredient;var CompoundIngredient=(function()
{function CompoundIngredient(name){this.name=name;this.ingredients=new
Array()}CompoundIngredient.prototype.AddIngredient=function(ing
redient){this.ingredients.push(ingredient)};CompoundIngredient.
prototype.GetName=function(){return this.name};CompoundIngredient.
prototype.GetCalories=function(){var total=0;for(var i=0;i<this.
ingredients.length;i++){total+=this.ingredients[i].GetCalories()}
return total};CompoundIngredient.prototype.GetIronContent=function()
{var total=0;for(var i=0;i<this.ingredients.length;i++){total+=this.
ingredients[i].GetIronContent()}return total};CompoundIngredient.
prototype.GetVitaminCContent=function(){var total=0;for(var
i=0;i<this.ingredients.length;i++){total+=this.ingredients[i].
GetVitaminCContent()}return total};return CompoundIngredient})();Food.
CompoundIngredient=CompoundIngredient})(Westros.Food||(Westros.
Food={}));var Food=Westros.Food})(Westros||(Westros={}));
```

You'll notice that all the spacing has been removed and that any internal variables have been replaced with smaller versions. At the same time, you can spot some well-known variable names have remained unchanged.

Minification saved this particular piece of code 40%. Compressing the content stream from the server using gzip, a popular approach, is lossless compression. That means that there is a perfect bijection between compressed and uncompressed. Minification, on the other hand, is a lossy compression. There is no way to get back to the unminified code from just the minified code once it has been minified.



You can read more about gzip compression at `http://betterexplained.com/articles/how-to-optimize-your-site-with-gzip-compression/`.

If there is a need to return to the original code, then source maps can be used. A source map is a file that provides a translation from one format of code to another. It can be loaded by the debugging tools in modern browsers to allow you to debug the original code instead of unintelligible minified code. Multiple source maps can be combine to allow for translation from, say, minified code to unminified JavaScript to TypeScript.

There are numerous tools which can be used to construct minified and combined JavaScript. Gulp and Grunt are JavaScript-based tools for building a pipeline which manages JavaScript assets. Both these tools call out to external tools such as Uglify to do the actual work. Gulp and Grunt are the equivalent to GNU Make or Ant.

Content Delivery Networks

The final delivery trick is to make use of **Content Delivery Networks** (**CDNs**). CDNs are distributed networks of hosts whose only purpose is to serve out static content. In much the same way that the browser will cache JavaScript between pages on the site, it will also cache JavaScript that is shared between multiple web servers. Thus, if your site makes use of jQuery, pulling jQuery from a well-known CDN such as `https://code.jquery.com/` or Microsoft's ASP.net CDN may be faster as it is already cached. Pulling from a CDN also means that the content is coming from a different domain and doesn't count against the limited connections to your server. Referencing a CDN is as simple as setting the source of the script tag to point at the CDN.

Once again, some metrics will need to be gathered to see whether it is better to use a CDN or simply roll libraries into the JavaScript bundle. Examples of such metrics may include the added time to perform additional DNS lookup and the difference in the download sizes. The best approach is to use the timing APIs in the browser.

The long and short of distributing JavaScript to the browser is that experimentation is required. Testing a number of approaches and measuring the results will give the best result for end users.

Plugins

There are a great number of really impressive JavaScript libraries in the wild. For me the library that changed how I look at JavaScript was jQuery. For others it may have been one of the other popular libraries such as MooTool, Dojo, Prototype, or YUI. However, jQuery has exploded in popularity and has, at the time of writing, won the JavaScript library wars. 78.5% of the top ten thousand websites, by traffic, on the internet make use of some version of jQuery. None of the rest of the libraries even breaks 1%.

Many developers have seen fit to implement their own libraries on top of these foundational libraries in the form of plugins. A plugin typically modifies the prototype exposed by the library and adds additional functionality. The syntax is such that, to the end developer, it appears to be part of the core library.

How plugins are built varies depending on the library you're trying to extend. Nonetheless, let's take a look at how we can build a plugin for jQuery and then for one of my favourite libraries, d3. We'll see if we can extract some commonalities.

jQuery

At jQuery's core is the CSS selector library called `sizzle.js`. It is sizzle that is responsible for all the really nifty ways jQuery can select items on a page using CSS3 selectors. Use jQuery to select elements on a page like so:

```
$(":input").css("background-color", "blue");
```

Here, a jQuery object is returned. The jQuery object acts a lot like, although not completely like, an array. This is achieved by creating a series of keys on the jQuery object numbered 0 through to n-1 where n is the number of elements matched by the selector. This is actually pretty smart as it enables array like accessors:

```
$($(":input")[2]).css("background-color", "blue");
```

While providing a bunch of additional functions, the items at the indices are plain HTML Elements and not wrapped with jQuery, hence the use of the second `$()`.

For jQuery plugins, we typically want to make our plugins extend this jQuery object. Because it is dynamically created every time the selector is fired we actually extend an object called `$.fn`. This object is used as the basis for creating all jQuery objects. Thus creating a plugin that transforms all the text in inputs on the page into uppercase is nominally as simple as the following:

```
$.fn.yeller = function(){
  this.each(function(_, item){
    $(item).val($(item).val().toUpperCase());
    return this;
  });
};
```

This plugin is particularly useful for posting to bulletin boards and for whenever my boss fills in a form. The plugin iterates over all the objects selected by the selector and converts their content to uppercase. It also returns this. By doing so we allow for chaining of additional functions. You can use the function like so:

```
$(function(){$("input").yeller();});
```

It does rather depend on the `$` variable being assigned to jQuery. This isn't always the case as `$` is a popular variable in JavaScript libraries, likely because it is the only character that isn't a letter or a number and doesn't already have special meaning.

To combat this, we can use an immediately evaluated function in much the same way we did way back in *Chapter 2*, *Organizing Code*:

```
(function($){
  $.fn.yeller2 = function(){
    this.each(function(_, item){
      $(item).val($(item).val().toUpperCase());
      return this;
    });
  };
})(jQuery);
```

The added advantage here is that, should our code require helper functions or private variables, they can be set inside the same function. You can also pass in any options required. jQuery provides a very helpful `$.extend` function that copies properties between objects, making it ideal for extending a set of default options with those passed in. We looked at this in some detail in a previous chapter.

The jQuery plugin documentation recommends that the jQuery object be polluted as little as possible with plugins. This is to avoid conflicts between multiple plugins that want to use the same names. Their solution is to have a single function that has different behaviours depending on the parameters passed in. For instance, the jQuery UI plugin uses this approach for dialog:

```
$(«.dialog»).dialog(«open»);
$(«.dialog»).dialog(«close»);
```

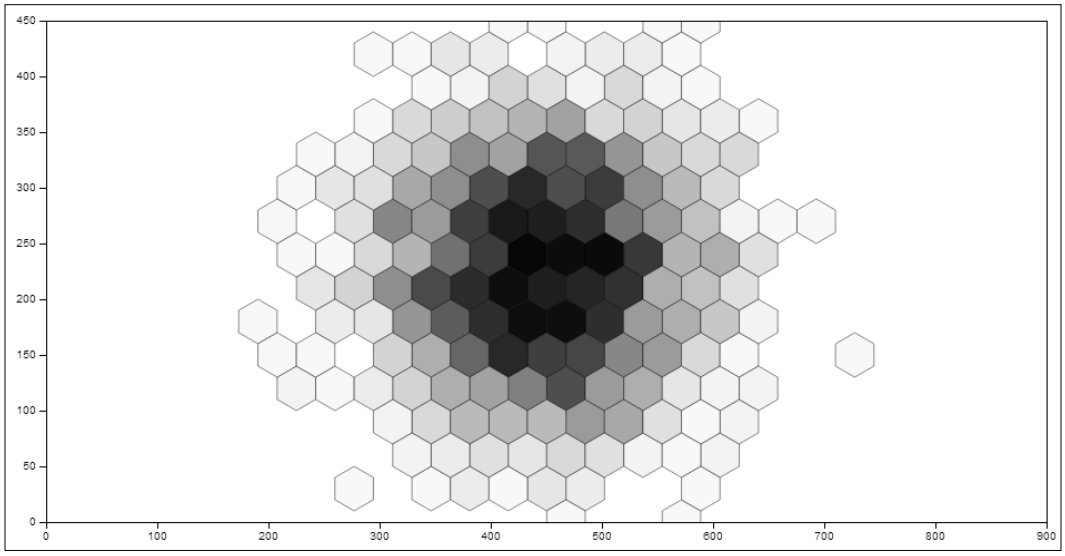
I would much rather call these like the following:

```
$(«.dialog»).dialog().open();
$(«.dialog»).dialog().close();
```

With dynamic languages there really isn't a great deal of difference but I would much rather have well named functions that can be discovered by tooling than magic strings.

d3

d3 is a great JavaScript library that is used for creating and manipulating visualizations. For the most part, people use d3 in conjunction with scalable vector graphics to produce graphics such as this hexbin graph by Mike Bostock:



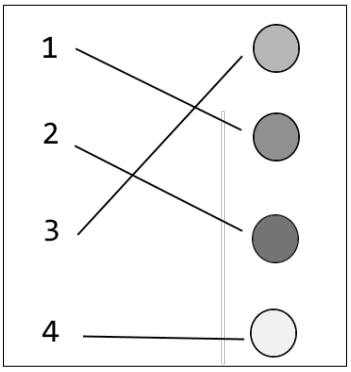
d3 attempts to be non-opinionated about the sorts of visualizations it creates. Thus there is no built-in support for creating such things as bar charts. There is, however, a collection of plugins that can be added to d3 to enable a wide variety of graphs including the hexbin one shown in the preceding figure.

Even more, the jQuery d3 places emphasis on creating chainable functions. For example, this code is a snippet that creates a column chart. You can see that all the attributes are being set through chaining:

```
var svg = d3.select(containerId).append("svg")
var bar = svg.selectAll("g").data(data).enter().append("g");
bar.append("rect")
.attr("height", yScale.rangeBand()).attr("fill", function (d, _) {
  return colorScale.getColor(d);
})
.attr("stroke", function (d, _) {
  return colorScale.getColor(d);
})
.attr("y", function (d, i) {
  return yScale(d.Id) + margins.height;
})
```

The core of `d3` is the `d3` object. Off that object hang a number of namespaces for layouts, scales, geometry, and numerous others. As well as whole namespaces, there are functions for doing array manipulation and loading data from external sources.

Creating a plugin for `d3` starts with deciding where we're going to plug into the code. Let's build a plugin that creates a new color scale. A color scale is used to map a domain of values to a range of colors. For instance, we might wish to map the domain of the following four values onto a range of four colors:



Let's plug in a function to provide a new color scale, in this case one that supports grouping elements. A scale is a function that maps a domain to a range. For a color scale, the range is a set of colors. An example might be a function that maps all even numbers to red and all odd to white. Using this scale on a table would result in zebra striping:

```
d3.scale.groupedColorScale = function () {
  var domain, range;

  function scale(x) {
    var rangeIndex = 0;
    domain.forEach(function (item, index) {
      if (item.indexOf(x) > 0)
        rangeIndex = index;
    });
    return range[rangeIndex];
  }

  scale.domain = function (x) {
    if (!arguments.length)
      return domain;
    domain = x;
    return scale;
  };

  scale.range = function (x) {
    if (!arguments.length)
      return range;
    range = x;
    return scale;
  };
  return scale;
};
```

We simply attach this plugin to the existing `d3.scale` object. This can be used by simply giving an array of arrays as a domain and an array as a range:

```
var s = d3.scale.groupedColorScale().domain([[1, 2, 3], [4, 5]]).
range(["#111111", "#222222"]);
s(3); //#111111
s(4); //#222222
```

This simple plugin extends the functionality of d3's scale. We could have replaced existing functionality or even wrapped it such that calls into existing functionality would be proxied through our plugin.

Plugins are generally not that difficult to build but they do vary from library to library. It is important to keep an eye on the existing variable names in libraries so we don't end up clobbering them or even clobbering functionality provided by other plugins. Some suggest prefixing functions with a string to avoid clobbering.

If the library has been designed with it in mind there may be additional places into which we can hook. A popular approach is to provide an options object that contains optional fields for hooking in our own functions as event handlers. If nothing is provided the function continues as normal.

Doing two things at once – multithreading

Doing two things at once is hard. For many years the solution in the computer world was to use either multiple processes or multiple threads. The difference between the two is fuzzy due to implementation differences on different operating systems but threads are typically lighter-weight versions of processes. JavaScript on the browser supports neither of these approaches.

Historically, there has been no real need for multithreading on a browser. JavaScript was used to manipulate the user interface. When manipulating a UI, even in other languages and windowing environments, only one thread is permitted to act at a time. This avoids race conditions that would be very obvious to users.

However, as JavaScript grows in popularity, more and more complicated software is being written to run inside the browser. Sometimes that software could really benefit from performing complex calculations in the background.

Web workers provide a mechanism for doing two things at once in a browser. Although a fairly recent innovation, web workers now have good support in mainstream browsers. In effect a worker is a background thread that can communicate with the main thread using messages. Web workers must be self-contained in a single JavaScript file.

To make use of web workers is fairly easy. We'll revisit our example from a few chapters ago when we looked at the fibonacci sequence. The worker process listens for messages like so:

```
self.addEventListener('message', function(e) {
  var data = e.data;
  if(data.cmd == 'startCalculation'){
    self.postMessage({event: 'calculationStarted'});
    var result = fib(data.parameters.number);
    self.postMessage({event: 'calculationComplete', result: result});
  };
}, false);
```

Here we start a new instance of `fib` any time we get a `startCalculation` message. `fib` is simply the naive implementation from earlier.

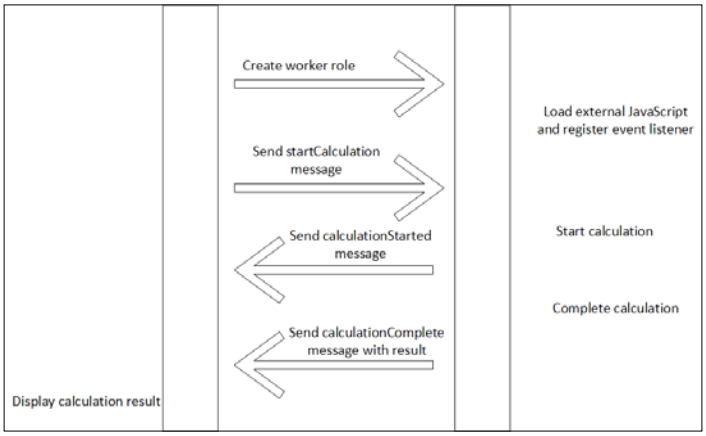
The main thread loads the worker process from its external file and attaches a number of listeners:

```
function startThread(){
  worker =  new Worker("worker.js");
  worker.addEventListener('message', function(message) {
    logEvent(message.data.event);
    if(message.data.event == "calculationComplete"){
      writeResult(message.data.result);
    }
    if(message.data.event == "calculationStarted"){
      document.getElementById("result").innerHTML = "working";
    }
  });
};
```

In order to start the calculation, all that is needed is to send a command:

```
worker.postMessage({cmd: 'startCalculation',
  parameters: { number: 40}});
```

Here we pass the number of the term in the sequence we want to calculate. While the calculation is running in the background the main thread is free to do whatever it likes. When the message is received back from the worker it is placed in the normal event loop to be processed:



Web workers may be useful to you if you have to do any time consuming calculations in JavaScript.

If you're making use of server-side JavaScript through the use of Node.js then there is a different approach to doing more than one thing at a time. Node.js offers the ability to fork child processes and provides an interface not dissimilar to the web worker one to communicate between the child and parent processes. This method forks an entire process though, which is much more resource intensive than using lightweight threads.

Some other tools exist that create lighter-weight background workers in Node.js. These are probably a closer parallel to what exists on the web side than forking a child process.

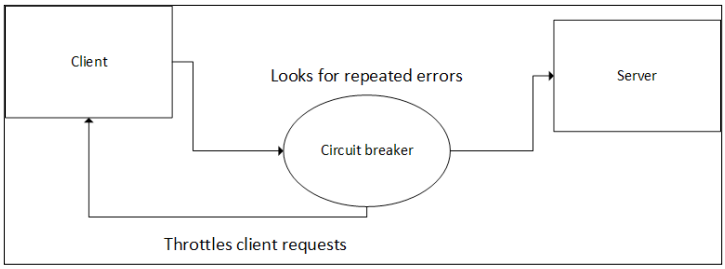
Circuit breaker pattern

Systems, even the best designed systems, fail. The larger and more distributed a system, the higher the probability of failure. Many large systems such as Netflix or Google have extensive built-in redundancies. The redundancies don't decrease the chance of a failure of a component but they do provide a backup. Switching to the backup is frequently transparent to the end user.

The circuit breaker pattern is a common component of a system that provides this sort of redundancy. Let's say that your application queries an external data source every five seconds, perhaps you're polling for some data that you're expecting to change. What happens when this polling fails? In many cases the failure is simply ignored and the polling continues. This is actually a pretty good behaviour on the client side as data updates are not always crucial. In some cases, a failure will cause the application to retry the request immediately. Retrying server requests in a tight loop can be problematic for both the client and the server. The client may become unresponsive as it spends more time in a loop requesting data.

On the server side, a system that is attempting to recover from a failure is being slammed every five seconds by what could be thousands of clients. If the failure is due to the system being overloaded, then continuing to query it will only make matters worse.

The circuit breaker pattern stops attempting to communicate with a system that is failing once a certain number of failures have been reached. Basically, repeated failures result in the circuit being broken and the application ceasing to query. You can see the general pattern of a circuit breaker in this illustration:



For the server, having the number of clients drop off as failures pile up allows for some breathing room to recover. The chances of a storm of requests coming in and keeping the system down is minimized.

Of course we would like the circuit breaker to reset at some point so that service can be restored. The two approaches for this are that, either the client polls periodically (less frequently than before) and resets the breaker, or that the external system communicates back to its clients that service has been restored.

Back-off

A variation on the circuit breaker pattern is to use some form of back-off instead of cutting out communication to the server completely. This is an approach that is suggested by many database vendors and cloud providers. If our original polling was at five second intervals, then when a failure is detected change the interval to every 10 seconds. Repeat this process using longer and longer intervals.

When requests start to work again then the pattern of changing the time interval is reversed. Requests are sent closer and closer together until the original polling interval is resumed.

Monitoring the status of the external resource availability is a perfect place to use background worker roles. The work is not complex but it is totally detached from the main event loop.

Again this reduces the load on the external resource giving it more breathing room. It also keeps the clients unburdened by too much polling.

An example using jQuery's `ajax` function looks like the following:

```
$.ajax({
  url : 'someurl',
  type : 'POST',
  data :  ....,
  tryCount : 0,
```

```
    retryLimit : 3,
    success : function(json) {
      //do something
    },
    error : function(xhr, textStatus, errorThrown ) {
      if (textStatus == 'timeout') {
        this.tryCount++;
        if (this.tryCount <= this.retryLimit) {
           //try again
           $.ajax(this);
           return;
        }
        return;
      }
      if (xhr.status == 500) {
        //handle error
      } else {
        //handle error
      }
    }
  });
```

You can see that the highlighted section retries the query.

This style of back-off is actually used in Ethernet to avoid repeated packet collisions.

Degraded application behavior

There is likely a very good reason that your application is calling out to external resources. Backing off and not querying the data source is perfectly reasonable but it is still desirable that users have some ability to interact with the site. One solution to this problem is to degrade the behavior of the application.

For instance, if your application shows real-time stock quote information, but the system for delivering stock information is broken, then a less than real time service could be swapped in. Modern browsers have a whole raft of different technologies that allow for storing small quantities of data on the client computer. This storage space is ideal for caching old versions of some data should the latest versions be unavailable.

Even in cases where the application is sending data to the server, it is possible to degrade behaviour. Saving the data updates locally and then sending them altogether when the service is restored is generally acceptable. Of course, once a user leaves a page, then any background works will terminate. If the user never again returns to the site, then whatever updates they had queued to send to the server will be lost.



A word of warning: if this is an approach you take it might be best to warn users that their data is old, especially if your application is a stock trading application.

Promise pattern

I said earlier that JavaScript is single threaded. This is not entirely accurate. There is a single event loop in JavaScript. Blocking this event loop with a long running process is considered to be bad form. Nothing else can happen while your greedy algorithm is taking up all the CPU cycles.

When you launch an asynchronous function in JavaScript, such as fetching data from a remote server, then much of this activity happens in a different thread. The success or failure handler functions are executed in the main event thread. This is part of the reason that success handlers are written as functions: it allows them to be easily passed back and forth between contexts.

Thus there are activities which truly do happen in an asynchronous, parallel fashion. When the `async` method has completed then the result is passed into the handler we provided and the handler is put into the event queue to be picked up next time the event loop repeats. Typically, the event loop runs many hundreds or thousands of times a second, depending on how much work there is to do on each iteration.

Syntactically, we write the message handlers as functions and hook them up:

```
var xmlhttp = new XMLHttpRequest();
xmlhttp.onreadystatechange = function() {
  if (xmlhttp.readyState === 4){
    alert(xmlhttp.readyState);
  }
;};
```

This is reasonable if the situation is simple. However, if you would like to perform some additional asynchronous actions with the results of the callback then you end up with nested callbacks. If you need to add error handling that too is done using callbacks. The complexity of waiting for multiple callbacks to return and orchestrating your response rises quickly.

The promise pattern provides some syntactic help to clean up the asynchronous difficulties. If we take a common asynchronous operation such as retrieving data over XMLHttp Request using jQuery, then the code takes both an error and a success function. It might look something like the following:

```
$.ajax("/some/url",
{ success: function(data, status){},
  error: function(jqXHR, status){}
});
```

Using a promise instead would transform the code to look more like the following:

```
$.ajax("/some/url").then(successFunction, errorFunction);
```

In this case the `$.ajax` method returns a promise object that contains a value and a state. The value is populated when the async call completes. The status provides some information about the current state of the request: has it completed, was it successful?

The promise also has a number of functions called on it. The `then()` function takes a success and an error function and it returns an additional promise. Should the success function run synchronously, then the promise returns as already fulfilled. Otherwise it remains in a working state, known as pending, until the asynchronous success has fired.

In my mind, the method in which jQuery implements promises is not ideal. Their error handing doesn't properly distinguish between a promise that has failed to be fulfilled and a promise that has failed but has been handled. This renders jQuery promises incompatible with the general idea of promises. For instance, it is not possible to do the following:

```
$.ajax("/some/url").then(
  function(data, status){},
  function(jqXHR, status){
    //handle the error here and return a new promise
  }
).then(/*continue*/);
```

Even though the error has been handed and a new promise returned, processing will discontinue. It would be much better if the function could be written as the following:

```
$.ajax("/some/url").then(function(data, status){})
.catch(function(jqXHR, status){
  //handle the error here and return a new promise
})
.then(/*continue*/);
```

There has been much discussion about the implementation of promises in jQuery and other libraries. As a result of this discussion the current proposed promise specification is different from jQuery's promises and is incompatible. Promises/A+ are the certification that is met by numerous promise libraries such as `when.js` and Q. It also forms the foundation of the promises specification that came with ECMAScript-2015.

Promises provide a bridge between synchronous and asynchronous functions, in effect turning the asynchronous functions into something that can be manipulated as if it were synchronous.

If promise sounds a lot like the lazy evaluation pattern we saw some chapters ago then you're exactly correct. Promises are constructed using lazy evaluation, the actions called on them are queued inside the object rather than being evaluated at once. This is a wonderful application of a functional pattern and even enables scenarios like the following:

```
when(function(){return 2+2;})
.delay(1000)
.then(function(promise){ console.log(promise());})
```

Promises greatly simplify asynchronous programming in JavaScript and should certainly be considered for any project that is heavily asynchronous in nature.

Hints and tips

ECMAScript 2015 promises are well supported on most browsers. Should you need to support an older browser then there are some great shims out there that can add the functionality with a minimum of overhead.

When examining the performance of retrieving JavaScript from a remote server, there are tools provided in most modern browsers for viewing a timeline of resource loading. This timeline will show when the browser is waiting for scripts to be downloaded and when it is parsing the scripts. Using this timeline allows for experimenting to find the best way to load a script or series of scripts.

Summary

In this chapter we've looked at a number of patterns or approaches that improve the experience of developing JavaScript. We looked at a number of concerns around delivery to the browser. We also looked at how to implement plugins against a couple of libraries and extrapolated general practices. Next we took a look at how to work with background processes in JavaScript. Circuit breakers were suggested as a method of keeping remote resource-fetching sane. Finally, we examined how promises can improve the writing of asynchronous code.

In the next chapter we'll spend quite a bit more time looking at messaging patterns. We saw a little about messing with web workers but we'll expand quite heavily on them in the next section.

10
Messaging Patterns

When Smalltalk, the first real object oriented programming language, was first developed, the communication between classes was envisioned as being messages. Somehow we've moved away from this pure idea of messages. We spoke a bit about how functional programming avoids side effects, well, much the same is true of messaging-based systems.

Messaging also allows for impressive scalability as messages can be fanned out to dozens, or even hundreds, of computers. Within a single application, messaging promotes low-coupling and eases testing.

In this chapter we're going to look at a number of patterns related to messaging. By the end of the chapter you should be aware of how messages work. When I first learned about messaging I wanted to rewrite everything using it.

We will be covering the following topics:

- What's a message anyway?
    - Commands
    - Events
- Request-reply
- Publish-subscribe
    - Fan out
- Dead letter queues
- Message replay
- Pipes and filters

What's a message anyway?

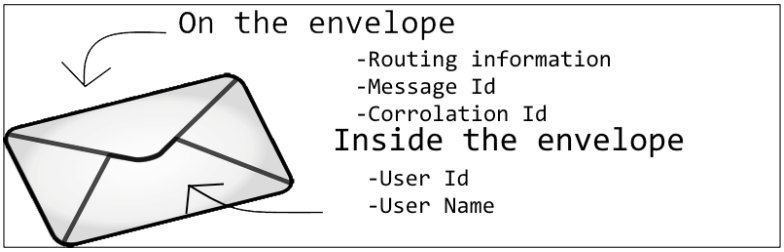
In the simplest definition a message is a collection of related bits of data that have some meaning together. The message is named in a way that provides some additional meaning to it. For instance, both an `AddUser` and a `RenameUser` message might have the following fields:

- User ID
- Username

But the fact that the fields exist inside a named container gives them different meaning.

Messages are usually related to some action in the application or some action in the business. A message contains all the information needed for a receiver to act upon the action. In the case of the `RenameUser` message, the message contains enough information for any component that keeps track of a relationship between a user ID and a username to update its value for username.

Many messaging systems, especially those that communicate between application boundaries, also define an **envelope**. The envelope has metadata on it that could help with message auditing, routing, and security. The information on the envelope is not part of the business process but is part of the infrastructure. So having a security annotation on the envelope is fine, as security exists outside of the normal business workflow and is owned by a different part of the application. The contents on the envelope look like the one shown in the following diagram:



Messages should be sealed so that no changes can be made to them once they have been created. This makes certain operations like auditing and replaying much easier.

Messaging can be used to communicate inside a single process or it can be used between applications. For the most part there is no difference to sending a message within an application and between applications. One difference is the treatment of synchronicity. Within a single process, messages can be handled in a synchronous fashion. This means that the main processing effectively waits for the handling of the message to complete before continuing.

In an asynchronous scenario, the handling of the message may occur at a later date. Sometimes the later date is far in the future. When calling out to an external server, asynchronous will certainly be the correct approach – this is due to the inherit latency associated with network I/O. Even within a single process, the single threaded nature of JavaScript encourages using asynchronous messaging. While using asynchronous messaging, some additional care and attention needs to be taken as some of the assumptions made for synchronous messaging cease to be safe. For instance, assuming the messages will be replied to in the same order in which they were sent is no longer safe.

There are two different flavors of messages: commands and events. Commands instruct things to happen while events notify about something which has happened.

Commands

A command is simply an instruction from one part of a system to another. It is a message so it is really just a simple data transfer object. If you think back to the command pattern introduced in *Chapter 5*, *Behavioral Patterns*, this is exactly what it uses.

As a matter of convention, commands are named using the imperative. The format is usually `<verb><object>`. Thus a command might be called `InvadeCity`. Typically, when naming a command, you want to avoid generic names and focus on exactly what is causing the command.

As an example, consider a command that changes the address of a user. You might be tempted to simply call the command `ChangeAddress` but doing so does not add any additional information. It would be better to dig deeper and see why the address is being changed. Did the person move or was the original address entered incorrectly? Intent is as important as the actual data changes. For instance, altering an address due to a mistake might trigger a different behavior from a person who has moved. Users that have moved could be sent a moving present, while those correcting their address would not.

Messages should have a component of business meaning to increase their utility. Defining messages and how they can be constructed within a complex business is a whole field of study on its own. Those interested might do well to investigate **domain driven design** (**DDD**).

Commands are an instruction targeted at one specific component giving it instructions to perform a task.

Within the context of a browser you might consider that a command would be the click that is fired on a button. The command is transformed into an event and that event is what is passed to your event listeners.

Only one end point should receive a specific command. This means that only one component is responsible for an action taking place. As soon as a command is acted upon by more than one end point any number of race conditions are introduced. What if one of the end points accepts the command and another rejects it as invalid? Even in cases where several near identical commands are issued they should not be aggregated. For instance, sending a command from a king to all his generals should send one command to each general.

Because there is only one end point for a command it is possible for that end point to validate and even cancel the command. The cancellation of the command should have no impact on the rest of the application.

When a command is acted upon, then one or more events may be published.

Events

An event is a special message that notifies that something has happened. There is no use in trying to change or cancel an event because it is simply a notification that something has happened. You cannot change the past unless you own a Delorian.

The naming convention for events is that they are written in the past tense. You might see a reversal of the ordering of the words in the command, so we could end up with `CityInvaded` once the `InvadeCity` command has succeeded.

Unlike commands, events may be received by any number of components. There are not real race conditions presented by this approach. As no message handler can change the message nor interfere with the delivery of other copies of the message, each handler is siloed away from all others.

You may be familiar with events from having done user interface work. When a user clicks a button then an event is "raised". In effect the event is broadcast to a series of listeners. You subscribe to a message by hooking into that event:

```
document.getElementById("button1").addEventListener("click",
  doSomething);
```

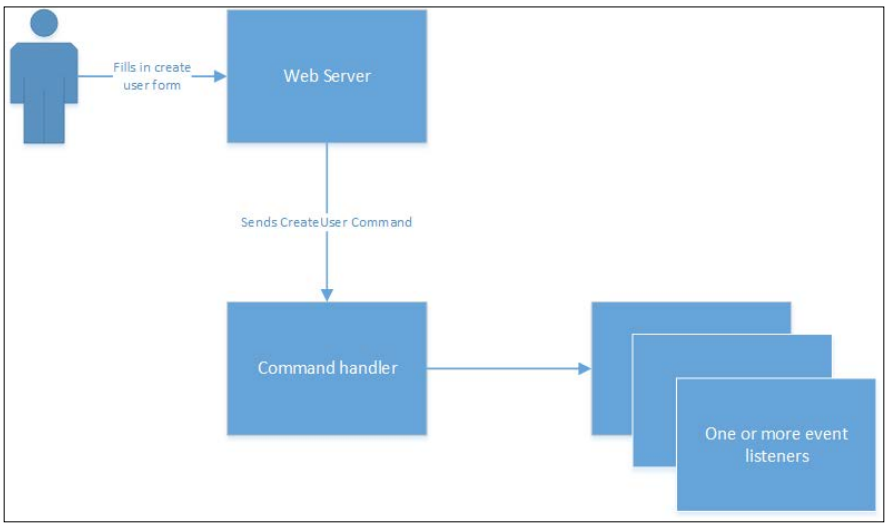
The events in browsers don't quite meet the definition of an event I gave in the preceding paragraph. This is because event handlers in the browser can cancel events and stop them from propagating to the next handler. That is to say, when there are a series of event handlers for the same message one of them can completely consume the message and not pass it on to subsequent handlers. There is certainly utility to an approach like this but it does introduce some confusion. Fortunately for UI messages, the number of handlers is typically quite small.

In some systems, events can be polymorphic in nature. That is to say that if I had an event called `IsHiredSalary` that is fired when somebody is hired in a salaried role, I could make it a descendant of the message `IsHired`. Doing so would allow for both handlers subscribed to `IsHiredSalary` and `IsHired` to be fired upon receipt of an `IsHiredSalary` event. JavaScript doesn't have polymorphism in the true sense, so such things aren't particularly useful. You can add a message field that takes the place of polymorphism but it looks somewhat messy:

```
var IsHiredSalary = { __name: "isHiredSalary",
  __alsoCall: ["isHired"],
  employeeId: 77,
  …
}
```

In this case I've used __ to denote fields that are part of the envelope. You could also construct the message with separate fields for message and envelope, it really doesn't matter all that much.

Let's take a look at a simple operation like creating a user so we can see how commands and events interact:



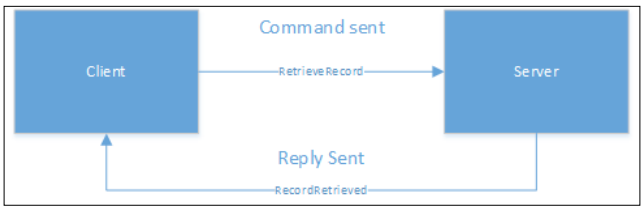
Here a user enters data into a form and submits it. The web server takes in the input, validates it and, if it is correct, creates a command. The command is now sent to the command handler. The command handler performs some action, perhaps writes to a database, it then publishes an event that is consumed by a number of event listeners. These event listeners might send confirmation e-mails, notify system administrators, or any number of things.

All of this looks familiar because systems already contain commands and events. The difference is that we are now modeling the commands and events explicitly.

Request-reply

The simplest pattern you'll see with messaging is the request-reply pattern. Also known as request-response, this is a method of retrieving data that is owned by another part of the application.

In many cases the sending of a command is an asynchronous operation. A command is fired and the application flow continues on. Because of this, there is no easy way to do things like lookup a record by ID. Instead one needs to send a command to retrieve a record and then wait for the associated event to be returned. A normal workflow looks like the following diagram:



Most events can be subscribed to by any number listeners. While it is possible to have multiple event listeners for a request-response pattern, it is unlikely and is probably not advisable.

We can implement a very simple request-response pattern here. In Westeros there are some issues with sending messages in a timely fashion. Without electricity, sending messages over long distances rapidly can really only be accomplished by attaching tiny messages to the legs of crows. Thus we have a Crow Messaging System.

We'll start with building out what we'll call the **bus**. A bus is simply a distribution mechanism for messages. It can be implemented in process, as we've done here, or out of process. If implementing it out of process, there are many options from 0mq, a lightweight message queue, to RabbitMQ, a more fully featured messaging system, to a wide variety of systems built on top of databases and in the cloud. Each of these systems exhibit some different behaviors when it comes to message reliability and durability. It is important to do some research into the way that the message distribution systems work as they may dictate how the application is constructed. They also implement different approaches to dealing with the underlying unreliability of applications:

```
class CrowMailBus {
  constructor(requestor) {
    this.requestor = requestor;
    this.responder = new CrowMailResponder(this);
  }
  Send(message) {
    if (message.__from == "requestor") {
      this.responder.processMessage(message);
    }
    else {
      this.requestor.processMessage(message);
    }
  }
}
```

One thing which is a potential trip-up is that the order in which messages are received back on the client is not necessarily the order in which they were sent. To deal with this it is typical to include some sort of a correlation ID. When the event is raised it includes a known ID from the sender so that the correct event handler is used.

This bus is a highly naïve one as it has its routing hard coded. A real bus would probably allow the sender to specify the address of the end point for delivery. Alternately, the receivers could register themselves as interested in a specific sort of message. The bus would then be responsible for doing some limited routing to direct the message. Our bus is even named after the messages it deals with – certainly not a scalable approach.

Next we'll implement the requestor. The requestor contains only two methods: one to send a request and the other to receive a response from the bus:

```
class CrowMailRequestor {
  Request() {
    var message = { __messageDate: new Date(),
```

```
      __from: "requestor",
      __corrolationId: Math.random(),
      body: "Hello there. What is the square root of 9?" };
      var bus = new CrowMailBus(this);
      bus.Send(message);
      console.log("message sent!");
    }
    processMessage(message) {
      console.dir(message);
    }
}
```

The process message function currently just logs the response but it would likely do more in a real world scenario such as updating the UI or dispatching another message. The correlation ID is invaluable for understanding which sent message the reply is related to.

Finally, the responder simply takes in the message and replies to it with another message:

```
class CrowMailResponder {
  constructor(bus) {
    this.bus = bus;
  }
  processMessage(message) {
    var response = { __messageDate: new Date(),
    __from: "responder",
    __corrolationId: message.__corrolationId,
    body: "Okay invaded." };
    this.bus.Send(response);
    console.log("Reply sent");
  }
}
```

Everything in our example is synchronous but all it would take to make it asynchronous is to swap out the bus. If we're working in node then we can do this using `process.nextTick` which simply defers a function to the next time through the event loop. If we're in a web context, then web workers may be used to do the processing in another thread. In fact, when starting a web worker, the communication back and forth to it takes the form of a message:

```
class CrowMailBus {
  constructor(requestor) {
    this.requestor = requestor;
    this.responder = new CrowMailResponder(this);
```

```
    }
    Send(message) {
      if (message.__from == "requestor") {
        process.nextTick(() => this.responder.processMessage(message));
      }
      else {
        process.nextTick(() => this.requestor.processMessage(message));
      }
    }
}
```

This approach now allows other code to run before the message is processed. If we weave in some print statements after each bus send, then we get output like the following:

```
Request sent!
Reply sent
{ __messageDate: Mon Aug 11 2014 22:43:07 GMT-0600 (MDT),
  __from: 'responder',
  __corrolationId: 0.5604551520664245,
  body: 'Okay, invaded.' }
```

You can see that the print statements are executed before the message processing as that processing happens on the next iteration.

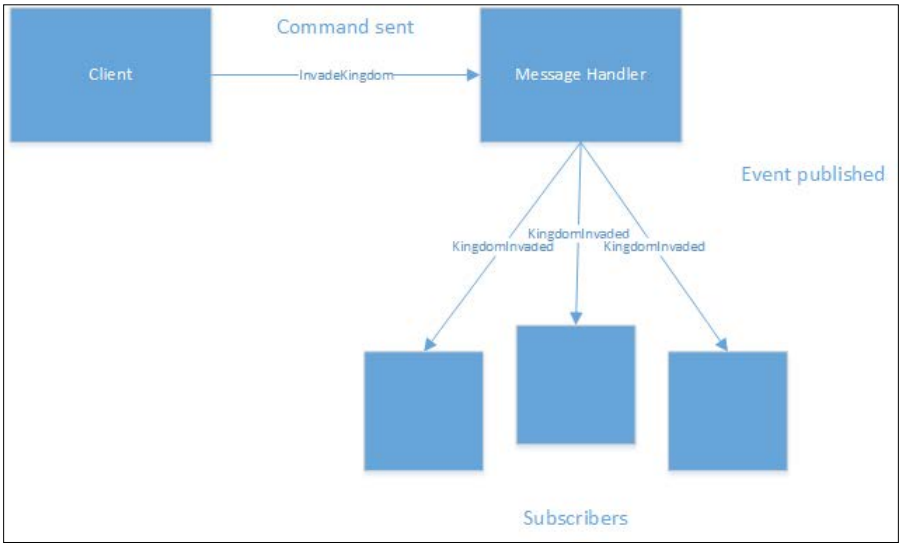
Publish-subscribe

I've alluded to the publish-subscribe model elsewhere in this chapter. Publish-subscribe is a powerful tool for decoupling events from processing code.

At the crux of the pattern is the idea that, as a message publisher, my responsibility for the message should end as soon as I send it. I should not know who is listening to messages or what they will do with the messages. So long as I am fulfilling a contract to produce correctly formatted messages, the rest shouldn't matter.

It is the responsibility of the listener to register its interest in the message type. You'll, of course, wish to register some sort of security to disallow registration of rogue services.

We can update our service bus to do more, to do a complete job of routing and sending multiple messages. Let's call our new method `Publish` instead of `Send`. We'll keep `Send` around to do the sending functionality:



The crow mail analogy we used in the previous section starts to fall apart here as there is no way to broadcast a message using crows. Crows are too small to carry large banners and it is very difficult to train them to do sky writing. I'm unwilling to totally abandon the idea of crows so let's assume that there exists a sort of crow broadcast centre. Sending a message here allows for it to be fanned out to numerous interested parties who have signed up for updates. This centre will be more or less synonymous with a bus.

We'll write our router so that it works as a function of the name of the message. One could route a message using any of its attributes. For instance, a listener could subscribe to all the messages called `invoicePaid` where the `amount` field is greater than $10000. Adding this sort of logic to the bus will slow it down and make it far harder to debug. Really this is more the domain of business process orchestration engines than a bus. We'll continue on without that complexity.

The first thing to set up is the ability to subscribe to published messages:

```
CrowMailBus.prototype.Subscribe = function (messageName, subscriber) {
  this.responders.push({ messageName: messageName, subscriber:
    subscriber });
};
```

The `Subscribe` function just adds a message handler and the name of a message to consume. The responders array is simply an array of handlers.

When a message is published we loop over the array and fire each of the handlers that have registered for messages with that name:

```
Publish(message) {
  for (let i = 0; i < this.responders.length; i++) {
    if (this.responders[i].messageName == message.__messageName) {
      (function (b) {
        process.nextTick(() => b.subscriber.processMessage(message));
      })(this.responders[i]);
    }
  }
}
```

The execution here is deferred to the next tick. This is done using a closure to ensure that the correctly scoped variables are passed through. We can now change our `CrowMailResponder` to use the new `Publish` method instead of `Send`:

```
processMessage(message) {
  var response = { __messageDate: new Date(),
  __from: "responder",
  __corrolationId: message.__corrolationId,
  __messageName: "SquareRootFound",
  body: "Pretty sure it is 3." };
  this.bus.Publish(response);
  console.log("Reply published");
}
```

Instead of allowing the `CrowMailRequestor` object to create its own bus as earlier, we need to modify it to accept an instance of `bus` from outside. We simply assign it to a local variable in `CrowMailRequestor`. Similarly, `CrowMailResponder` should also take in an instance of `bus`.

In order to make use of this we simply need to create a new bus instance and pass it into the requestor:

```
var bus = new CrowMailBus();
bus.Subscribe("KingdomInvaded", new TestResponder1());
bus.Subscribe("KingdomInvaded", new TestResponder2());
var requestor = new CrowMailRequestor(bus);
requestor.Request();
```

Here we've also passed in two other responders that are interested in knowing about `KingdomInvaded` messages. They look like the following:

```
var TestResponder1 = (function () {
  function TestResponder1() {}
  TestResponder1.prototype.processMessage = function (message) {
    console.log("Test responder 1: got a message");
  };
  return TestResponder1;
})();
```

Running this code will now get the following:

```
Message sent!
Reply published
Test responder 1: got a message
Test responder 2: got a message
Crow mail responder: got a message
```

You can see that the messages are sent using `Send`. The responder or handler does its work and publishes a message that is passed onto each of the subscribers.

There are some great JavaScript libraries which make publish and subscribe even easier. One of my favorites is Radio.js. It has no external dependencies and its name is an excellent metaphor for publish subscribe. We could rewrite our preceding subscribe example like so:

```
radio("KingdomInvalid").subscribe(new TestResponder1().
  processMessage);
radio("KingdomInvalid").subscribe(new TestResponder2().
  processMessage);
```

Then publish a message using the following:

```
radio("KingdomInvalid").broadcast(message);
```

Fan out and in

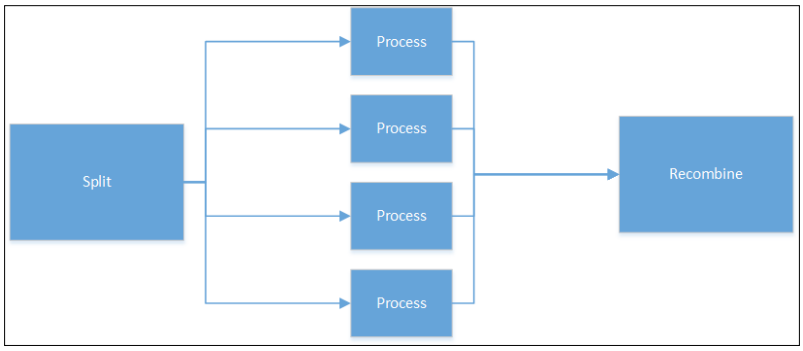
A fantastic use of the publish subscribe pattern is allowing you to fan out a problem to a number of different nodes. Moore's law has always been about the doubling of the number of transistors per square unit of measure. If you've been paying attention to processor clock speeds you may have noticed that there hasn't really been any significant change in clock speeds for a decade. In fact, clock speeds are now lower than they were in 2005.

This is not to say that processors are "slower" than they once were. The work that is performed in each clock tick has increased. The number of cores has also jumped up. It is now unusual to see a single core processor; even in cellular phones dual core processors are becoming common. It is the rule, rather than the exception, to have computers that are capable of doing more than one thing at a time.

At the same time, cloud computing is taking off. The computers you purchase outright are faster than the ones available to rent from the cloud. The advantage of cloud computing is that you can scale it out easily. It is nothing to provision a hundred or even a thousand computers to form a cloud provider.

Writing software that can take advantage of multiple cores is the great computing problem of our time. Dealing directly with threads is a recipe for disaster. Locking and contention is far too difficult a problem for most developers: me included! For a certain class of problems, they can easily be divided up into sub problems and distributed. Some call this class of problems "embarrassingly parallelizable".

Messaging provides a mechanism for communicating the inputs and outputs from a problem. If we had one of these easily parallelized problems, such as searching, then we would bundle up the inputs into one message. In this case it would contain our search terms. The message might also contain the set of documents to search. If we had 10,000 documents then we could divide the search space up into, say, four collections of 2500 documents. We would publish five messages with the search terms and the range of documents to search as can be seen here:



Different search nodes will pick up the messages and perform the search. The results will then be sent back to a node that will collect the messages and combine them into one. This is what will be returned to the client.

Of course this is a bit of an over simplification. It is likely that the receiving nodes themselves would maintain a list of documents over which they had responsibility. This would prevent the original publishing node from having to know anything about the documents over which it was searching. The search results could even be returned directly to the client that would do the assembling.

Even in a browser, the fan out and in approach can be used to distribute a calculation over a number of cores through the use of web workers. A simple example might take the form of creating a potion. A potion might contain a number of ingredients that can be combined to create a final product. It is quite computationally complicated combining ingredients so we would like to farm the process out to a number of workers.

We start with a combiner that contains a `combine()` method as well as a `complete()` function that is called once all the distributed ingredients are combined:

```
class Combiner {
  constructor() {
    this.waitingForChunks = 0;
  }
  combine(ingredients) {
    console.log("Starting combination");
    if (ingredients.length > 10) {
      for (let i = 0; i < Math.ceil(ingredients.length / 2); i++) {
        this.waitingForChunks++;
        console.log("Dispatched chunks count at: " + this.
          waitingForChunks);
        var worker = new Worker("FanOutInWebWorker.js");
        worker.addEventListener('message', (message) => this.
          complete(message));
        worker.postMessage({ ingredients: ingredients.slice(i, i * 2)
          });
      }
    }
  }
  complete(message) {
    this.waitingForChunks--;
    console.log("Outstanding chunks count at: " + this.
      waitingForChunks);
    if (this.waitingForChunks == 0)
      console.log("All chunks received");
  }
};
```

In order to keep track of the number of workers outstanding, we use a simple counter. Because the main section of code is single threaded we have no risk of race conditions. Once the counter shows no remaining workers we can take whatever steps are necessary. The web worker looks like the following:

```
self.addEventListener('message', function (e) {
  var data = e.data;
  var ingredients = data.ingredients;
  combinedIngredient = new Westeros.Potion.CombinedIngredient();
  for (let i = 0; i < ingredients.length; i++) {
    combinedIngredient.Add(ingredients[i]);
  }
  console.log("calculating combination");
  setTimeout(combinationComplete, 2000);
}, false);

function combinationComplete() {
  console.log("combination complete");
  (self).postMessage({ event: 'combinationComplete', result:
    combinedIngredient });
}
```

In this case we simply put in a timeout to simulate the complex calculation needed to combine ingredients.

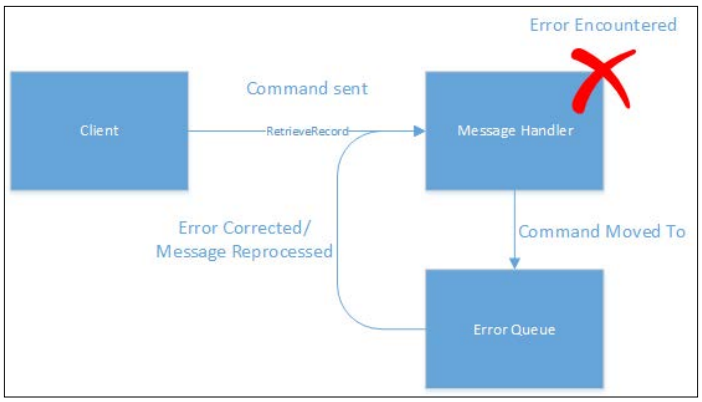
The sub problems that are farmed out to a number of nodes don't have to be identical problems. However, they should be sufficiently complicated that the cost savings of farming them out are not consumed by the overhead of sending out a message.

Dead letter queues

No matter how hard I try I have yet to write any significant block of code that does not contain any errors. Nor have I been very good at predicting the wide range of crazy things users do with my applications. Why would anybody click that link 73 times in a row? I'll never know.

Dealing with failures in a messaging scenario is very easy. The core of the failure strategy is to embrace errors. We have exceptions for a reason and to spend all of our time trying to predict and catch exceptions is counter-productive. You'll invariably spend time building in catches for errors that never happen and miss errors that happen frequently.

In an asynchronous system, errors need not be handled as soon as they occur. Instead, the message that caused an error can be put aside to be examined by an actual human later. The message is stored in a dead letter, or error, queue. From there the message can easily be reprocessed after it has been corrected or the handler has been corrected. Ideally the message handler is changed to deal with messages exhibiting whatever property caused the errors. This prevents future errors and is preferable to fixing whatever generates the message as there is no guarantee that other messages with the same problem aren't lurking somewhere else in the system. The workflow of a message through the queue and error queue can be seen here:



As more and more errors are caught and fixed, the quality of the message handlers increases. Having an error queue of messages ensures that nothing important, such as a `BuySimonsBook` message is missed. This means that getting to a correct system becomes a marathon instead of a sprint. There is no need to rush a fix into production before it is properly tested. Progress towards a correct system is constant and reliable.

Using a dead letter queue also improves the catching of intermittent errors. These are errors that result from an external resource being unavailable or incorrect. Imagine a handler that calls out to an external web service. In a traditional system, a failure in the web service guarantees failure in the message handler. However, with a message based system, the command can be moved back to the end of the input queue and tried again whenever it reaches the front of the queue. On the envelope we write down the number of times the message has been dequeued (processed). Once this dequeue count reaches a limit, like five, only then is the message moved into the true error queue.

This approach improves the overall quality of the system by smoothing over the small failures and stopping them from becoming large failures. In effect, the queues provide failure bulkheads to prevent small errors from overflowing and becoming large errors that might have an impact on the system as a whole.

Message replay

When developers are working with a set of messages that produce an error, the ability to reprocess messages is also useful. Developers can take a snapshot of the dead letter queue and reprocess it in debug mode again and again until they have correctly processed the messages. A snapshot of a message can also make up a part of the testing for a message handler.

Even without there being an error, the messages sent to a service on a daily basis are representative of the normal workflows of users. These messages can be mirrored to an audit queue as they enter into the system. The data from the audit queue can be used for testing. If a new feature is introduced, then a normal day's workload can be played back to ensure that there has been no degradation in either correct behavior or performance.

Of course if the audit queue contains a list of every message, then it becomes trivial to understand how the application arrived at its current state. Frequently people implement history by plugging in a lot of custom code or by using triggers and audit tables. Neither of these approaches do as good of a job as messaging at understanding not only which data has changed, but why it has changed. Consider again the address change scenario, without messaging we will likely never know why an address for a user is different from the previous day.

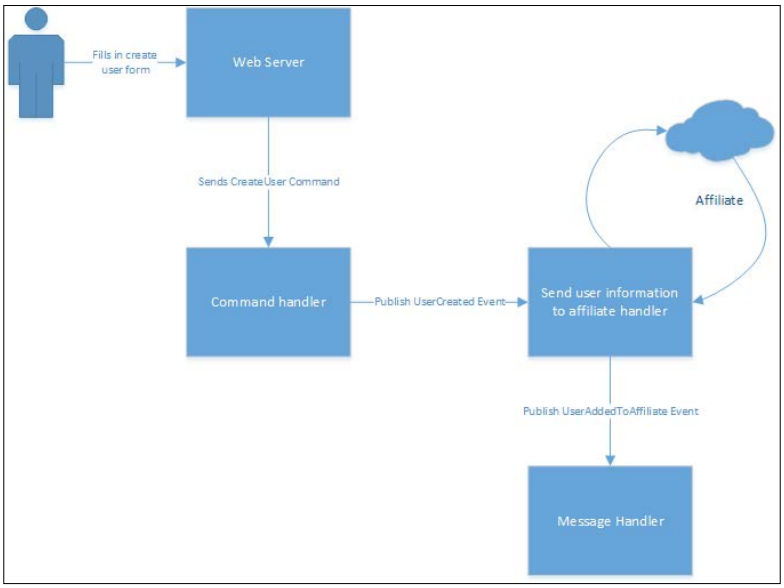
Maintaining a good history of changes to system data is storage intensive but that cost is easily paid by allowing auditors to see how and why each change was made. Well-constructed messages also allow for the history to contain the intent of the user making the change.

While it is possible to implement this sort of messaging system, in a single process it is difficult. Ensuring that messages are properly saved in the event of errors is difficult, as the entire process that deals with messages may crash, taking the internal message bus with it. Realistically if the replaying of messages sounds like something worth investigating then external message busses are the solution.

Pipes and filters

I mentioned earlier that messages should be considered immutable. This is not to say that messages cannot be rebroadcast with some properties changed or even broadcast as a new type of message. In fact, many message handlers may consume an event and then publish a new event after having performed some task.

As an example, you might consider the workflow for adding a new user to a system:



In this case, the `CreateUser` command triggers a `UserCreated` event. That event is consumed by a number of different services. One of these services passes on user information to a select number of affiliates. As this service runs, it publishes its own set of events, one for each affiliate that receives the new user's details. These events may, in turn, be consumed by other services which could trigger their own events. In this way changes can ripple through the entire application. However, no service knows more than what starts it and what events it publishes. This system has very low coupling. Plugging in new functionality is trivial and even removing functionality is easy: certainly easier than a monolithic system.

Systems constructed using messaging and autonomous components are frequently referred to as using **Service Oriented Architecture** (**SOA**) or Microservices. There remains a great deal of debate as to the differences, if indeed there are any, between SOA and Microservices.

The altering and rebroadcasting of messages can be thought of as being a pipe or a filter. A service can proxy messages through to other consumers just as a pipe would do or can selectively republish messages as would be done by a filter.

Versioning messages

As systems evolve, the information contained in a message may also change. In our user creation example, we might have originally been asking for a name and e-mail address. However, the marketing department would like to be able to send e-mails addressed to Mr. Jones or Mrs. Jones so we need to also collect the user's title. This is where message versioning comes in handy.

We can now create a new message that extends the previous message. The message can contain additional fields and might be named using the version number or a date. Thus a message like `CreateUser` might become `CreateUserV1` or `CreateUser20140101`. Earlier I mentioned polymorphic messages. This is one approach to versioning messages. The new message extends the old so all the old message handlers still fire. However, we also talked about how there are no real polymorphic capabilities in JavaScript.

Another option is to use upgrading message handlers. These handlers will take in a version of the new message and modify it to be the old version. Obviously the newer messages need to have at least as much data in them as the old version or have data that permits converting one message type to another.

Consider a v1 message that looked like the following:

```
class CreateUserv1Message implements IMessage{
  __messageName: string
  UserName: string;
  FirstName: string;
  LastName: string;
  EMail: string;
}
```

Consider a v2 message that extended it adding a user title:

```
class CreateUserv2Message extends CreateUserv1Message implements
  IMessage{
  UserTitle: string;
}
```

Then we would be able to write a very simple upgrader or downgrader that looks like the following:

```
var CreateUserv2tov1Downgrader = (function () {
  function CreateUserv2tov1Downgrader (bus) {
    this.bus = bus;
  }
  CreateUserv2tov1Downgrader.prototype.processMessage = function
    (message) {
```

```
        message.__messageName = "CreateUserv1Message";
        delete message.UserTitle;
        this.bus.publish(message);
      };
      return CreateUserv2tov1Downgrader;
    })();
```

You can see that we simply modify the message and rebroadcast it.

Hints and tips

Messages create a well-defined interface between two different systems. Defining messages should be done by members of both teams. Establishing a common language can be tricky especially as terms are overloaded between different business units. What a sales department considers a customer may be totally different from what a shipping department considers a customer. Domain driven design provides some hints as to how boundaries can be established to avoid mixing terms.

There is a huge preponderance of queue technologies available. Each of them have a bunch of different properties around reliability, durability, and speed. Some of the queues support reading and writing JSON over HTTP: ideal for those interested in building JavaScript applications. Which queue is appropriate for your application is a topic for some research.

Summary

Messaging and the associated patterns are large topics. Delving too deeply into messages will bring you in contact with **domain driven design** (**DDD**) and **command query responsibility segregation** (**CQRS**) as well as touching on high performance computing solutions.

There is substantial research and discussion ongoing as to the best way to build large systems. Messaging is one possible solution that avoids creating a big ball of mud that is difficult to maintain and fragile to change. Messaging provides natural boundaries between components in a system and the messages themselves provide for a consistent API.

Not every application benefits from messaging. There is additional overhead to building a loosely coupled application such as this. Applications that are collaborative, ones where losing data is especially undesirable, and those that benefit from a strong history story are good candidates for messaging. In most cases a standard CRUD application will be sufficient. It is still worthwhile to know about messaging patterns, as they will offer alternative thinking.

In this chapter we've taken a look at a number of different messaging patterns and how they can be applied to common scenarios. The differences between commands and events were also explored.

In the next chapter we'll look at some patterns for making testing code a little bit easier. Testing is jolly important so read on!

11
Microservices

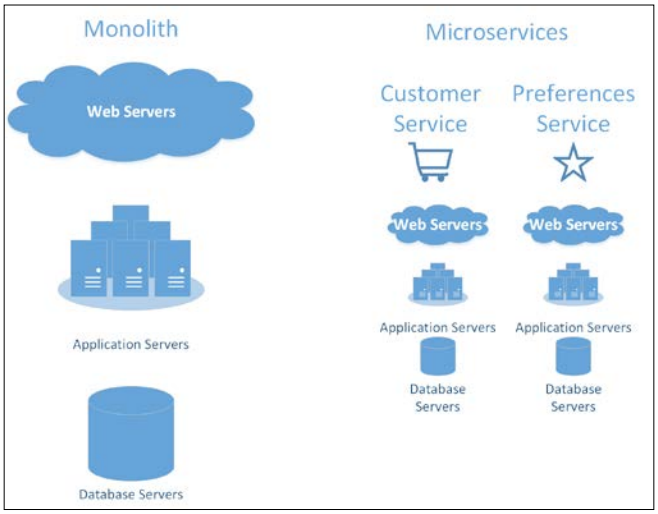
It seems like no book on programming these days is complete without at least some mention of microservices. For fear that this book could be singled out for ridicule as a non-conformant publication, a chapter has been included on microservices.

Microservices are billed as the solution to the problems of monolithic applications. Likely every application you've dealt with has been a monolith: that is, the application has a single logical executable and is perhaps split into layers such as a user interface, a service or application layer, and a data store. In many applications these layers might be a web page, a server side application, and a database. Monoliths have their issues as I'm sure you've encountered.

Maintaining a monolithic application quickly becomes an exercise in limiting the impact of change. Frequently in such applications a change to one, seemingly isolated, corner of the application has an unintended effect on some other part of the application. Although there are many patterns and approaches to describe well isolated components, these often fall by the wayside inside a monolith. Often we take shortcuts which may save time now but will return to make our lives terrible down the road.

Monolithic applications are also difficult to scale. Because we tend to have only three layers, we are limited to scaling each one of those layers. We can add more application servers if the middle tier is becoming slow or more web servers if the web tier is laggy. If the database is slow then we can increase the power of the database server. These scaling approaches are very large operations. If the only part of the application which is slow is signing up new users, then we really have no way to simply scale that one component. This means that components which are not frequently used (one might call these cold or cool components) must be able to scale as the whole application scales. This sort of scaling doesn't come for free.

Consider that scaling from a single web server to multiple web servers introduces the problem of sharing sessions between many web servers. If we were, instead, to divide our application into a number of services, of which each acts as the canonical source of truth for a piece of data, then we could scale these sections independently. A service for logging users in, another service for saving and retrieving their preferences, yet another for sending out reminder e-mails about abandoned shopping carts, each one responsible for its own functions and own data. Each service stands alone as a separate application and may run on a separate machine. In effect we have taken our monolithic application and sharded it into many applications. Not only does each service have an isolated function but it also has its own datastore and could be implemented using its own technology. The difference between a monolith and microservices can be seen here:



Applications are written more by composing services than by writing singular monolithic applications. The UI of an application can even be created by asking a number of services to provide visual components to be slotted into a composite UI by some form of composing service.

Node.js' lightweight approach to building applications with just the required components makes it an ideal platform to build lightweight microservices. Many microservice deployments make heavy use of HTTP to communicate between services while others rely more heavily on messaging systems such as **RabbitMQ** or **ZeroMQ**. These two communication methods may be mixed in deployments. One might split the technology used along the lines of using HTTP against services which are query-only, and messaging against services which perform some action. This is because messaging is more reliable (depending on your messaging system and configuration) than sending HTTP requests.

While it may seem that we've introduced a great deal of complexity into the system it is a complexity that is easier to manage with modern tooling. Very good tooling exists for managing distributed log files and for monitoring the performance of applications for performance issues. Isolating and running many applications with virtualization is more approachable than ever with containerization technologies.

Microservices may not be the solution to all our maintenance and scalability issues but they are certainly an approach that is viable for consideration. In this chapter we'll explore some of the patterns that may assist in using microservices:

- Façade
- Aggregate services
- Pipeline
- Message upgrader
- Service selector
- Failure patterns

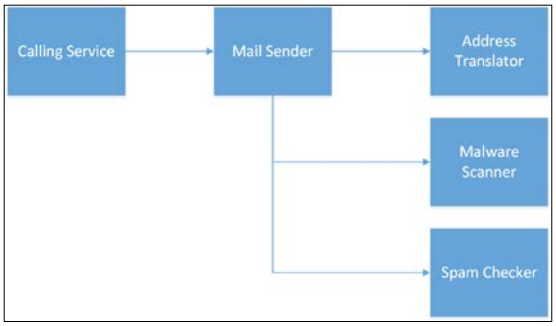
Because microservices are a relatively new development, there are likely to be many more patterns which emerge as more and more applications are created with the microservice approach. There is some similarity between the Microservices approach and **Service Oriented Architecture** (**SOA**). This means that there are likely some patterns from the SOA world which will be applicable in the microservices world.

Façade

If you feel that you recognize the name of this pattern, then you're correct. We discussed this pattern way back in *Chapter 4*, *Structural Patterns*. In that application of the pattern we created a class which could direct the actions of a number of other classes providing a simpler API. Our example was that of an admiral who directed a fleet of ships. In the microservices world we can simply replace the concept of classes with that of services. After all, the functionality of a service is not that different from a microservice – they both perform a single action.

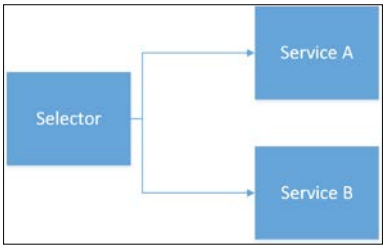
We might make use of a façade to coordinate the use of a number of other services. This pattern is a base pattern for many of the other patterns listed in this chapter. Coordinating services can be difficult, but by putting them behind a façade we can make the entire application much simpler. Let us consider a service which sends e-mails. Sending e-mails is quite a complex process which may involve a number of other services: a username to e-mail address translator, an anti-malware scanner, a spam checker, a formatter to message the e-mail body for various e-mail clients, and so forth.

Most clients who want to send e-mail don't want to concern themselves with all of these other services so a façade e-mail-sending service can be put in place which holds the responsibility of coordinating other services. The coordination pattern can be seen here:



Service selector

Along the same lines as a façade we have the service selector pattern. In this pattern we have a service which fronts a number of other services. Depending on the message which arrives, a different service could be selected to respond to the initial request. This pattern is useful in upgrade scenarios and for experimentation. If you're rolling out a new service and want to ensure that it will function correctly under load then you could make use of the service selector pattern to direct a small portion of your production traffic to the new service while monitoring it closely. Another application might be for directing specific customers or groups of customers to a different service. The distinguishing factor could be anything from directing people who have paid for your service toward faster end points, to directing traffic from certain countries to country-specific services. The service selector pattern can be seen in this illustration:



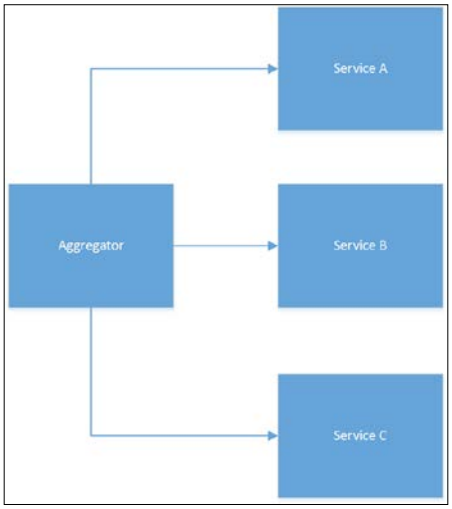
Aggregate services

Data in a microservice architecture is owned by a single service, but there are many times when we might need to retrieve data from a number of different sources at once. Consider, again, a member of the Small Council in the land of Westeros. They may have a number of informants from whom they gather information about the workings of the kingdom. You can consider each informant to be its own microservice.



Informants are a fantastic metaphor for microservices as each one is independent and holds its own data. Services may also fail from time to time just as informants may be captured and terminated. Messages are passed between informants just as they are among a collection of microservices. Each informant should know very little about how the rest of the informants work, and even, who they are – an abstraction which works for microservices too.

With the aggregate service pattern, we ask each one of a collection of nodes to perform some action or return some piece of data. This is a fairly common pattern even outside the microservice world and is a special case of the façade or even adapter pattern. The aggregator requests information from a number of other services and then waits for them to return. Once all the data has been returned, then the aggregator may perform some additional tasks such as summarizing the data or counting records. The information is then passed back to the caller. The aggregator can be seen in this illustration:



This pattern may also have some provision for dealing with slow-to-return services or failures of services. The aggregator service may return partial results or return data from a cache in the event that one of the child services reaches a timeout. In certain architectures, the aggregator could return a partial result and then return additional data to the caller when it becomes available.

Pipeline

A pipeline is another example of a microservice connecting pattern. If you have made use of the shell on a *NIX system, then you have certainly piped the output of one command to another command. The programs on a *NIX system such as ls, sort, uniq, and grep are designed to perform just one task; their power comes from the ability to chain the tools together to build quite complex workflows. For instance, this command:

```
ls -1| cut -d \. -f 2 -s | sort |uniq
```

This command will list all the unique file extensions in the current directory. It does this by taking the list of files, then cutting them and taking the extension; this is then sorted and finally passed to `uniq` which removes duplicates. While I wouldn't suggest having a microservice for such trivial actions as sorting or deduplicating, you might have a series of services which build up more and more information.

Let's imagine a query service that returns a collection of company records:

```
| Company Id| Name | Address | City | Postal Code | Phone Number |
```

This record is returned by our company lookup service. Now we can pass this record onto our sales accounting service which will add a sales total to the record:

```
| Company Id| Name | Address | City | Postal Code | Phone Number | 2016
orders Total |
```

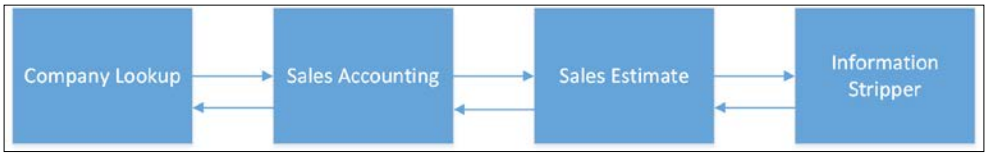
Now that record can be passed onto a sales estimate service, which further enhances the record with an estimate of 2017 sales:

```
| Company Id| Name | Address | City | Postal Code | Phone Number | 2016
orders Total | 2017 Sales Estimate |
```

This sort of progressive enhancement could be reversed too by a service that stripped out information which shouldn't be presented to the users. The record might now become the following:

```
| Name | Address | City | Postal Code | Phone Number | 2016 orders Total
| 2017 Sales Estimate |
```

Here we have dropped the company identifier because it is an internal identifier. A microservice pipeline should be bidirectional so that a quantum of information is passed into each step in the pipeline and then passed back out again through each step. This affords services the opportunity to act upon the data twice, manipulating it as they see fit. This is the same approach used in many web servers where modules such as PHP are permitted to act upon the request and the response. A pipeline can be seen illustrated here:



Message upgrader

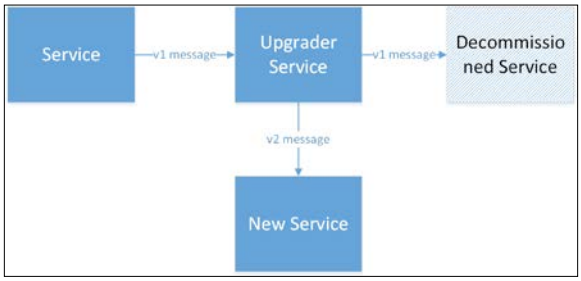
One of the highest-risk activities with some monolithic applications is upgrading. To do so you basically need to upgrade the entire application at once. With even a medium-sized application there are far too many aspects to reasonably test. Thus at some point you simply need to cut over from the old system to the new system. With a microservice approach, the cutover can be done for each individual service. Smaller services mean that the risk can be spread out over a long time and, should something go wrong, the source of the error can be more quickly pinpointed: the singular new component.

At issue are the services which are still talking to the old version of the upgraded service. How can we continue to serve these services without having to update all those services too? If the interface to the service remains unchanged, say our service calculates the distance between two points on the earth and we change it from using a simple Pythagorean approach to using haversine (a formula to find the distance between two spots on a sphere), then there may be no need to make changes to the input and output formats. Frequently, however, this approach isn't available to us as the message format must change. Even in the previous example there is a possibility of changing the output message. Haversine is more accurate than a Pythagorean approach so we could have more significant digits requiring a larger data type. There are two good approaches to deal with this:

1. Continue to use the old version of our service and the new version. We can then slowly move the client services over to the new service as time permits. There are problems with this approach: we now need to maintain more code. Also, if the reason we change the service out was one which would not permit us to continue to run it (a security problem, termination of a dependent service, and so on) then we are at something of an impasse.

2. Upgrade messages and pass them on. In this approach we take the old message format and upgrade it to the new format. This is done by, you guessed it, another service. This service's responsibility is to take in the old message format and emit the new message format. At the other end you might need an equivalent service to downgrade messages back to the expected output format for older services.

Upgrader services should have a limited lifespan. Ideally we would want to make updates to the services which depend on deprecated services as quickly as possible. The small code footprint of microservices, coupled with the ability to rapidly deploy services, should make these sorts of upgrade much easier than those used to a monolithic approach might expect. An example message upgrader service can be seen here:



Failure patterns

We have already touched upon some of the ways of dealing with failures in microservices in this chapter. There are, however, a couple of more interesting approaches we should consider. The first of these is service degradation.

Service degradation

This pattern could also be called graceful degradation and is related to progressive enhancement. Let us hark back to the example of replacing the Pythagorean distance function with the haversine equivalent. If the haversine service is down for some reason, the less demanding function could be used in its place without a huge impact on users. In fact, they may not notice it at all. It isn't ideal that users have a worse version of the service but it is certainly more desirable than simply showing the user an error message. When the haversine service returns to life then we can stop using the less desirable service. We could have multiple levels of fallback allowing several different services to fail while we continue to present a fully functional application to the end user.

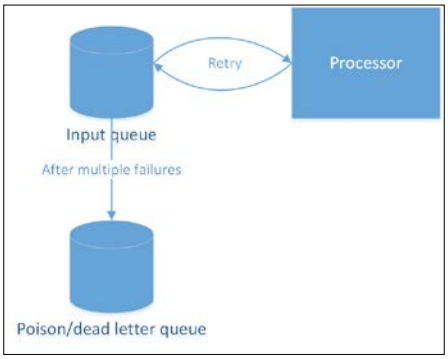
Another good application of this form of degradation is to fall back to more expensive services. I once had an application that sent SMS messages. It was quite important that these messages actually be sent. We used our preferred SMS gateway provider the majority of the time but, if our preferred service was unavailable, something we monitored closely, then we would fail over to using a different provider.

Message storage

We've already drawn a bit of a distinction between services which are query-only and those which actually perform some lasting data change. When one of these updating services fails there is still a need to run the data change code at some point in the future. Storing these requests in a message queue allows them to be run later without risk of losing any of the ever-so important messages. Typically, when a message causes an exception it is returned to the processing queue where it can be retried.

There is an old saying that insanity is doing the same thing over again and expecting a different outcome. However, there are many transient errors which can be solved by simply performing the same action over again. Database deadlocks are a prime example of this. Your transaction may be killed to resolve a deadlock, in which case performing it again is, in fact, the recommended approach. However, one cannot retry messages ad infinitum so it is best to choose some relatively small number of retry attempts, three or five. Once this number has been reached then the message can be sent to a dead letter or poison message queue.

Poison messages, or dead letters as some call them, are messages which have actual legitimate reasons for failing. It is important to keep these messages around not only for debugging purposes but because the messages may represent a customer order or a change to a medical record: not data you can afford to lose. Once the message handler has been corrected these messages can be replayed as if the error never happened. A storage queue and message reprocessor can be seen illustrated here:



Message replay

Although not a real production pattern, a side-effect of having a message-based architecture around all the services which change data is that you can acquire the messages for later replay outside of production. Being able to replay messages is very handy for debugging complex interactions between numerous services as the messages contain almost all the information to set up a tracing environment identical to production. Replay capabilities are also very useful for environments where one must be able to audit any data changes to the system. There are other methods to address such audit requirements but a very solid message log is simply a delight to work with.

Indempotence of message handling

The final failure pattern we'll discuss is idempotence of message handling. As systems grow larger it is almost certain that a microservices architecture will span many computers. This is even more certain due to the growing importance of containers, which can, ostensibly, be thought of as computers. Communicating between computers in a distributed system is unreliable; thus, a message may end up being delivered more than once. To handle such an eventuality one might wish to make messaging handling idempotent.



For more about the unreliability of distributed computing, I cannot recommend any paper more worth reading than *Falacies of Distributed Computing Explained* by Arnon Rotem-Gal-Oz at `http://rgoarchitects.com/Files/fallacies.pdf`.

Idempotence means that a message can be processed many times without changing the outcome. This can be harder to achieve than one might realize, especially with services which are inherently non-transactional such as sending e-mails. In these cases, one may need to write a record that an e-mail has been sent to a database. There are some scenarios in which the e-mail will be sent more than once, but a service crashing in the critical section between the e-mail being sent and the record of it being written is unlikely. The decision will have to be made: is it better to send an e-mail more than once or not send it at all?

Hints and tips

If you think of a microservice as a class and your microservice web as an application, then it rapidly becomes apparent that many of the same patterns we've seen elsewhere in the book are applicable to microservices. Service discovery could be synonymous with dependency injection. Singleton, decorator, proxy; all of them could be applicable to the microservice world just as they are within the boundaries of a process.

One thing to keep in mind is that many of these patterns are somewhat chatty, sending significant data back and forth. Within a process there is no overhead to passing around pointers to data. The same is not true of microservices. Communicating over the network is likely to incur a performance penalty.

Summary

Microservices are a fascinating idea and one which is more likely to be realized in the next few years. It is too early to tell if this is simply another false turn on the way to properly solving software engineering or a major step in the right direction. In this chapter we've explored a few patterns which may be of use should you embark upon a journey into the microservices world. Because we're only on the cusp of microservices becoming mainstream, it is likely that, more than any other chapter of this book, the patterns here will quickly become dated and found to be suboptimal. Remaining vigilant with regard to developments and being aware of the bigger picture when you're developing is highly advisable.

12
Patterns for Testing

Throughout this book we've been pushing the idea that JavaScript is no longer a toy language with which we can't do useful things. Real world software is being written in JavaScript right now and the percentage of applications using JavaScript is only likely to grow over the next decade.

With real software comes concerns about correctness. Manually testing software is painful and, weirdly, error-prone. It is far cheaper and easier to produce unit and integration tests that run automatically and test various aspects of the application.

There are countless tools available for testing JavaScript, from test runners to testing frameworks; the ecosystem is a rich one. We'll try to maintain a more or less tool-agnostic approach to testing in this chapter. This book does not concern itself with which framework is the best or friendliest. There are overarching patterns that are common to testing as a whole. It is those that we'll examine. We will touch on some specific tools but only as a shortcut to having to write all our own testing tools.

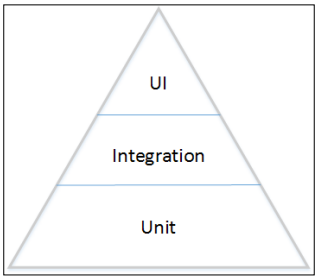
In this chapter we'll look at the following topics:

- Fake objects
- Monkey patching
- Interacting with the user interface

The testing pyramid

We computer programmers are, as a rule, highly analytical people. This means that we're always striving to categorize and understand concepts. This has led to our developing some very interesting global techniques that can be applied outside computer programming. For instance, agile development has applications in general society but can trace its roots back to computing. One might even argue that the idea of patterns owes much of its popularity to it being used by computer programmers in other walks of life.

This desire to categorize has led to the concept of testing code being divided up into a number of different types of tests. I've seen as many as eight different categories of tests from unit tests, right the way up to workflow tests and GUI tests. This is, perhaps, an overkill. It is much more common to think about having three different categories of test: unit, integration, and user interface:



Unit tests form the foundation of the pyramid. They are the most numerous, the easiest to write, and the most granular in the errors they give. An error in a unit test will allow you to find the individual method that has an error in it. As we move up the pyramid, the number of tests falls along with the granularity while the complexity of each test increases. At a higher level, when a test fails we might only be able to say: "There is an issue with adding an order to the system".

Testing in the small with unit tests

To many, unit testing is a foreign concept. This is understandable as it is a topic which is poorly taught in many schools. I know that I've done six years of higher education in computing science without it being mentioned. It is unfortunate because delivering a quality product is a pretty important part of any project.

For those who know about unit testing, there is a big barrier to adoption. Managers, and even developers, frequently see unit testing, and automated testing as a whole, as a waste of time. After all you cannot ship a unit test to your customer nor do most customers care whether their product has been properly unit tested.

Unit testing is notoriously difficult to define. It is close enough to integration testing that people slip back and forth between the two easily. In the seminal book; *The Art of Unit Testing*, *Roy Osherove*, the author defines a unit test as:

> *A unit test is an automated piece of code that invokes a unit of work in the system and then checks a single assumption about the behavior of that unit of work.*

The exact size of a unit of work is up for some debate. Some people restrict it to a single function or a single class, while others allow a unit of work to span multiple classes. I tend to think that a unit of work that spans multiple classes can actually be broken into smaller, testable units.

The key to unit testing is that it tests a small piece of functionality and it quickly tests the functionality in a repeatable, automated fashion. Unit tests written by one person should be easily runnable by any other member of the team.

For unit testing we want to test small pieces of functionality because we believe that if all the components of a system work correctly then the system as a whole will work. This is not the whole truth. The communication between modules is just as likely to fail as a function within the unit. This is why we want to write tests on several levels. Unit tests check that the code we're writing right now is correct. Integration testing tests entire workflows through the application and will uncover problems in the interaction of units.

The test-driven development approach suggests writing tests at the same time as we write code. While this gives us great confidence that the code we're writing is correct, the real advantage is that it helps drive good architecture. When code has too many interdependencies it is far harder to test than well-separated modular code. A lot of the code that developers write goes unread by anybody ever again. Unit tests provide a useful way of keeping developers on the right path even in cases where they know that nobody will ever see their code. There is no better way to produce a quality product than to tell people they are going to be checked on it, even if the checker happens to be an automated test.

Tests can be run both while developing new code and in an automatic fashion on the build machines. If every time a developer checks in a change, the entire project is built and tested, then some reassurance can be provided that the newly checked-in code is correct. From time to time the build will break and that will be a flag that something that was just added was in error. Often the code that is broken may not even be proximal to the code changed. An altered return value may percolate through the system and manifest itself somewhere wholly unexpected. Nobody can keep anything more than the most trivial system in their mind at any one time. Testing acts as a sort of second memory, checking and rechecking assumptions made previously.

Failing the build as soon as an error occurs shortens the time it takes between an error being made in the code and it being found and fixed. Ideally the problem will still be fresh in the developer's mind so the fix can easily be found. If the errors were not discovered until months down the road, the developer will certainly have forgotten what s/he was working on at the time. The developer may not even be around to help solve the problem, throwing somebody who has never seen the code in to fix it.

Arrange-Act-Assert

When building tests for any piece of code, a very common approach to follow is that of Arrange-Act-Assert. This describes the different steps that take place inside a single unit test.

The first thing we do is set up a test scenario (arrange). This step can consist of a number of actions and may involve putting in place fake objects to simulate real objects as well as creating new instances of the subject under test. If you find that your test setup code is long or involved, it is likely a smell and you should consider refactoring your code. As mentioned in the previous section, testing is helpful for driving not just correctness but also architecture. Difficult-to-write tests are indicative that the architecture is not sufficiently modular.

Once the test is set up then the next step is to actually execute the function we would like to test (act). The act step is usually very short, in many cases no more than a single line of code.

The final part is to check to make sure that the result of the function or the state of the world is as you would expect (assert).

A very simple example of this might be a castle builder:

```
class CastleBuilder {
  buildCastle(size) {
    var castle = new Castle();
    castle.size = size;
    return castle;
  }
}
```

This class simply builds a new castle of a specific size. We want to make sure that no shenanigans are going on and that when we build a castle of size `10` we get a castle of size `10`:

```
function When_building_a_castle_size_should_be_correctly_set() {
  var castleBuilder = new CastleBuilder();
  var expectedSize = 10;
  var builtCastle = castleBuilder.buildCastle(10);
  assertEqual(expectedSize, builtCastle.size);
}
```

Assert

You may have noticed that in the last example we made use of a function called `assertEquals`. An assert is a test that, when it fails, throws an exception. There is currently no built-in assert functionality in JavaScript, although there is a proposal in the works to add it.

Fortunately, building an assert is pretty simple:

```
function assertEqual(expected, actual){
  if(expected !== actual)
  throw "Got " + actual + " but expected " + expected;
}
```

It is helpful to mention, in the error, the actual value as well as the expected value.

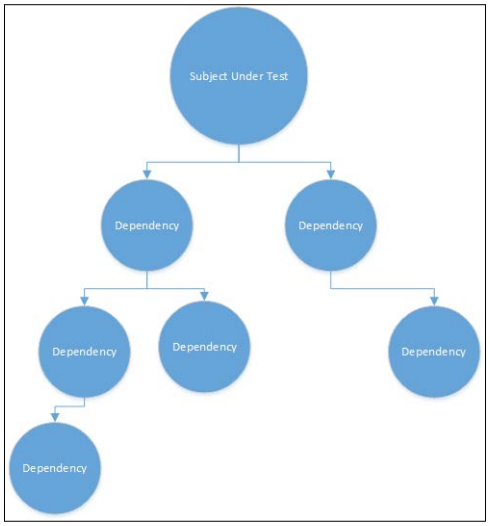
There is a great number of assertion libraries in existence. Node.js ships with one, creatively called `assert.js`. If you end up using a testing framework for JavaScript it is likely that it will also contain an assertion library.

Fake objects

If we think of the interdependencies between objects in an application as a graph it becomes quickly apparent that there are a number of nodes that have dependencies on, not just one, but many other objects. Attempting to place an object with a lot of dependencies under test is challenging. Each of the dependent objects must be constructed and included in the test. When these dependencies interact with external resources such as the network or file system, the problem becomes intractable. Pretty soon we're testing the entire system at a time. This is a legitimate testing strategy, known as **integration testing**, but we're really just interested in ensuring that the functionality of a single class is correct.

Integration testing tends to be slower to execute than unit tests.

The subject of a test can have a large dependency graph that makes testing it difficult. You can see an example here:



We need to find a way to isolate the class under test so that we don't have to recreate all the dependencies, including the network. We can think of this approach as adding bulkheads to our code. We will insert bulkheads to stop tests from flowing over from one class to many. These bulkheads are similar to how oil tankers maintain separation to limit the impact of spills and maintain weight distribution as can be seen here:

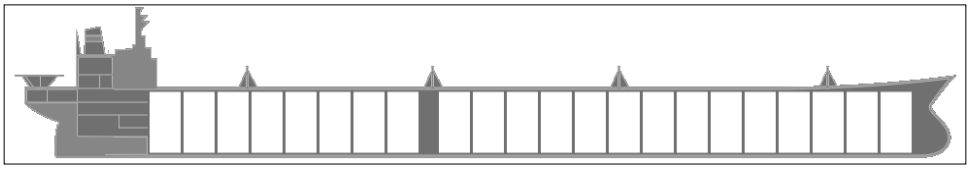


*Image courtesy of `http://www.reactivemanifesto.org/`.

To this end we can use fake objects that have a limited set of functionalities in place of the real objects. We'll look at three different methods of creating fake objects.

The first is the, rather niftily named, test spy.

Test spies

A spy is an approach that wraps all the methods of an object and records the inputs and outputs from that method as well as the number of calls. By wrapping the calls, it is possible to examine exactly what was passed in and what came out of the function. Test spies can be used when the exact inputs into a function are not known beforehand.

In other languages, building test spies requires reflection and can be quite complicated. We can actually get away with making a basic test spy in no more than a couple of lines of code. Let's experiment.

To start we'll need a class to intercept:

```
var SpyUpon = (function () {
  function SpyUpon() {
  }
  SpyUpon.prototype.write = function (toWrite) {
    console.log(toWrite);
  };
  return SpyUpon;
})();
```

Now we would like to spy on this function. Because functions are first class objects in JavaScript we can simply rejigger the `SpyUpon` object:

```
var spyUpon = new SpyUpon();
spyUpon._write = spyUpon.write;
spyUpon.write = function (arg1) {
  console.log("intercepted");
  this.called = true;
  this.args = arguments;
  this.result = this._write(arg1, arg2, arg3, arg4, arg5);
  return this.result;
};
```

Here we take the existing function and give it a new name. Then we create a new function that calls the renamed function and also records some things. After the function has been called we can examine the various properties:

```
console.log(spyUpon.write("hello world"));
console.log(spyUpon.called);
console.log(spyUpon.args);
console.log(spyUpon.result);
```

Running this code in node gets us the following:

```
hello world
7
true
{ '0': 'hello world' }
7
```

Using this technique, it is possible to get all sorts of insight into how a function is used. There are a number of libraries that support creating test spies in a more robust way than our simple version here. Some provide tools for recording exceptions, the number of times called, and the arguments for each call.

Stubs

A **stub** is another example of a fake object. We can use stubs when we have some dependencies in the subject under test that need to be satisfied with an object that returns a value. They can also be used to provide a bulkhead to stop computationally expensive or I/O reliant functions from being run.

Stubs can be implemented in much the same way that we implemented spies. We just need to intercept the call to the method and replace it with a version that we wrote. However, with stubs we actually don't call the replaced function. It can be useful to keep the replaced function around just in case we need to restore the functionality of the stubbed out class.

Let's start with an object that depends on another object for part of its functionality:

```
class Knight {
  constructor(credentialFactory) {
    this.credentialFactory = credentialFactory;
  }
  presentCredentials(toRoyalty) {
    console.log("Presenting credentials to " + toRoyalty);
    toRoyalty.send(this.credentialFactory.Create());
    return {};
  }
}
```

This knight object takes a `credentialFactory` argument as part of its constructor. By passing in the object we exteriorize the dependency and remove the responsibility for creating `credentialFactory` from the knight. We've seen this sort of inversion of control previously and we'll look at it in more detail in the next chapter. This makes our code more modular and testing far easier.

Now when we want to test the knight without worrying about how a credential factory works, we can use a fake object, in this case a stub:

```
class StubCredentialFactory {
  constructor() {
    this.callCounter = 0;
  }
  Create() {
    //manually create a credential
  };
}
```

This stub is a very simple one that simply returns a standard new credential. Stubs can be made quite complicated if there need to be multiple calls to it. For instance, we could rewrite our simple stub as the following:

```
class StubCredentialFactory {
  constructor() {
    this.callCounter = 0;
  }
  Create() {
    if (this.callCounter == 0)
      return new SimpleCredential();
    if (this.callCounter == 1)
      return new CredentialWithSeal();
    if (this.callCounter == 2)
      return null;
    this.callCounter++;
  }
}
```

This version of the stub returns a different sort of credential every time it is called. On the third call it returns null. As we set up the class using an inversion of control, writing a test is as simple as the following:

```
var knight = new Knight(new StubCredentialFactory());
knight.presentCredentials("Queen Cersei");
```

We can now execute the test:

```
var knight = new Knight(new StubCredentialFactory());
var credentials = knight.presentCredentials("Lord Snow");
assert(credentials.type === "SimpleCredentials");
credentials = knight.presentCredentials("Queen Cersei");
assert(credentials.type === "CredentialWithSeal");
credentials = knight.presentCredentials("Lord Stark");
assert(credentials == null);
```

Because there is no hard typing system in JavaScript, we can build stubs without worrying about implementing interfaces. There is also no need to stub an entire object but only the function in which we're interested.

Mock

The final type of fake object is a **mock**. The difference between a mock and a stub is where the verification is done. With a stub, our test must check if the state is correct after the act. With a mock object, the responsibility for testing the asserts falls to the mock itself. Mocks are another place where it is useful to leverage a mocking library. We can, however, build the same sort of thing, simply, ourselves:

```
class MockCredentialFactory {
  constructor() {
    this.timesCalled = 0;
  }
  Create() {
    this.timesCalled++;
  }
  Verify() {
    assert(this.timesCalled == 3);
  }
}
```

This `mockCredentialsFactory` class takes on the responsibility of verifying the correct functions were called. This is a very simple sort of approach to mocking and can be used as such:

```
var credentialFactory = new MockCredentialFactory();
var knight = new Knight(credentialFactory);
var credentials = knight.presentCredentials("Lord Snow");
credentials = knight.presentCredentials("Queen Cersei");
credentials = knight.presentCredentials("Lord Stark");
credentialFactory.Verify();
```

This is a static mock that keeps the same behavior every time it is used. It is possible to build mocks that act as recording devices. You can instruct the mock object to expect certain behaviors and then have it automatically play them back.

The syntax for this is taken from the documentation for the mocking library; Sinon. It looks like the following:

```
var mock = sinon.mock(myAPI);
mock.expects("method").once().throws();
```

Monkey patching

We've seen a number of methods for creating fake objects in JavaScript. When creating the spy, we made use of a method called **monkey patching**. Monkey patching allows you to dynamically change the behavior of an object by replacing its functions. We can use this sort of approach without having to revert to full fake objects. Any existing object can have its behavior changed in isolation using this approach. This includes built-in objects such as strings and arrays.

Interacting with the user interface

A great deal of the JavaScript in use today is used on the client and is used to interact with elements that are visible on the screen. Interacting with the page flows through a model of the page known as **Document Object Model** (**DOM**).

Every element on the page is represented in the DOM. Whenever a change is made to the page, the DOM is updated. If we add a paragraph to the page, then a paragraph is added to the DOM. Thus if our JavaScript code adds a paragraph, checking that it does so is simply a function of checking the DOM.

Unfortunately, this requires that a DOM actually exists and that it is formed in the same way that it is on the actual page. There are a number of approaches to testing against a page.

Browser testing

The most naïve approach is to simply automate the browser. There are a few projects out there that can help with this task. One can either automate a fully-fledged browser such as Firefox, Internet Explorer, or Chrome, or one can pick a browser that is headless. The fully-fledged browser approach requires that a browser be installed on the test machine and that the machine be running in a mode that has a desktop available.

Many Unix-based build servers will not have been set up to show a desktop as it isn't needed for most build tasks. Even if your build machine is a Windows one, the build account frequently runs in a mode that has no ability to open a window. Tests using full browsers also have a tendency to break, to my mind. Subtle timing issues crop up and tests are easily interrupted by unexpected changes to the browser. It is a frequent occurrence that manual intervention will be required to unstick a browser that has ended up in an incorrect state.

Fortunately, efforts have been made to decouple the graphical portions of a web browser from the DOM and JavaScript. For Chrome this initiative has resulted in PhantomJS and for Firefox SlimerJS.

Typically, the sorts of test that require a full browser require some navigation of the browser across several pages. This is provided for in the headless browsers through an API. I tend to think of tests at this scale as integration tests rather than unit tests.

A typical test using PhantomJS and the CasperJS library that sits on top of the browser might look like the following:

```
var casper = require('casper').create();
casper.start('http://google.com', function() {
  assert.false($("#gbqfq").attr("aria-haspopup"));
  $("#gbqfq").val("redis");
  assert.true($("#gbqfq").attr("aria-haspopup"));
});
```

This would test that entering a value into the search box on Google changes the `aria-haspopup` property from `false` to `true`.

Testing things this way puts a great deal of reliance on the DOM not changing too radically. Depending on the selectors used to find elements on the page, a simple change to the style of the page could break every test. I like to keep tests of this sort away from the look of that page by never using CSS properties to select elements. Instead make use of IDs or, better yet, data-* attributes. We don't necessarily have the luxury of that when it comes to testing existing pages but certainly for new pages it is a good plan.

Faking the DOM

Much of the time, we don't need a full page DOM to perform our tests. The page elements we need to test are part of a section on the page instead of the entire page. A number of initiatives exist that allow for the creation of a chunk of the document in pure JavaScript. `jsdom` for instance is a method for injecting a string of HTML and receiving back a fake window.

In this example, modified slightly from their README, they create some HTML elements, load JavaScript, and test that it returns correctly:

```
var jsdom = require("jsdom");
jsdom.env( '<p><a class="the-link" ref="https://github.com/tmpvar/
  jsdom">jsdom!</a></p>',
  ["http://code.jquery.com/jquery.js"],
```

```
    function (errors, window) {
      assert.equal(window.$("a.the-link").text(), "jsdom!");
    }
  );
```

If your JavaScript is focused on a small section of the page, perhaps you're building custom controls or web components, then this is an ideal approach.

Wrapping the manipulation

The final approach to dealing with graphical JavaScript is to stop interacting directly with elements on the page. This is the approach that many of the more popular JavaScript frameworks of today use. One simply updates a JavaScript model and this model then updates the page through the use of some sort of MV* pattern. We looked at this approach in some detail some chapters ago.

Testing in this case becomes quite easy. Our complicated JavaScript can simply be tested by building a model state prior to running the code and then testing to see if the model state after running the code is as we expect.

As an example we could have a model that looks like the following:

```
class PageModel{
  titleVisible: boolean;
  users: Array<User>;
}
```

The test code for it might look as simple as the following:

```
var model = new PageModel();
model.titleVisible = false;
var controller = new UserListPageController(model);
controller.AddUser(new User());
assert.true(model.titleVisible);
```

As everything on the page is manipulated, through the bindings to the model, we can be confident that changes in the model are correctly updating the page.

Some would argue that we've simply shifted the problem. Now the only place for errors is if the binding between the HTML and the model is incorrect. So we also need to test if we have bindings correctly applied to the HTML. This falls to higher-level testing that can be done more simply. We can cover far more with a higher-level test than with a lower-level one, although at the cost of knowing exactly where the error occurred.

You're never going to be able to test everything about an application but the smaller you can make the untested surface, the better.

Tips and tricks

I have seen tests where people split up the Arrange-Act-Assert by putting in place comments:

```
function testMapping(){
  //Arrange
  …
  //Act
  …
  //Assert
  …
}
```

You're going to wear your fingers to the bone typing those comments for every single test. Instead I just split them up with a blank line. The separation is clear and anybody who knows Arrange-Act-Assert will instantly recognize what it is that you're doing. You'll have seen the example code in this chapter split up in this fashion.

There are countless JavaScript testing libraries available to make your life easier. Choosing one may depend on your preferred style. If you like a gherkin-style syntax then cuumber.js might be for you. Otherwise try mocha, either on its own, or with the chai BDD style assertion library , which is is fairly nice. There are also testing frameworks such as Protractor which are specific to Angular apps (although you can use it to test other frameworks with a bit of work). I'd suggest taking a day and playing with a few to find your sweet spot.

When writing tests, I tend to name them in a way that makes it obvious that they are tests and not production code. For most JavaScript I follow camel case naming conventions such as `testMapping`. However, for test methods I follow an underscored naming pattern `When_building_a_castle_size_should_be_correctly_set`. In this way the test reads more like a specification. Others have different approaches to naming and there is no "right" answer, so feel free to experiment.

Summary

Producing a quality product is always going to require extensive and repeated testing; this is exactly the sort of thing computers are really good at. Automate as much as possible.

Testing JavaScript code is an up-and-coming thing. The tooling around, mocking out objects, and even the tools for running tests are undergoing constant changes. Being able to use tools such as Node.js to run tests quickly and without having to boot up an entire browser is stunningly helpful. This is an area that is only going to improve over the next few years. I am enthused to see what changes come from it.

In the next chapter we'll take a look at some advanced patterns in JavaScript that you might not want to use every day but are very handy.

13
Advanced Patterns

I hesitated when naming this chapter, *Advanced Patterns*. This isn't really about patterns that are more complicated or sophisticated than other patterns. It is about patterns that you wouldn't use very frequently. Frankly, coming from a static programming language background, some of them seem crazy. Nonetheless they are completely valid patterns and are in use within big name projects everywhere.

In this chapter we'll be looking at the following topics:

- Dependency injection
- Live post processing
- Aspect oriented programming
- Macros

Dependency injection

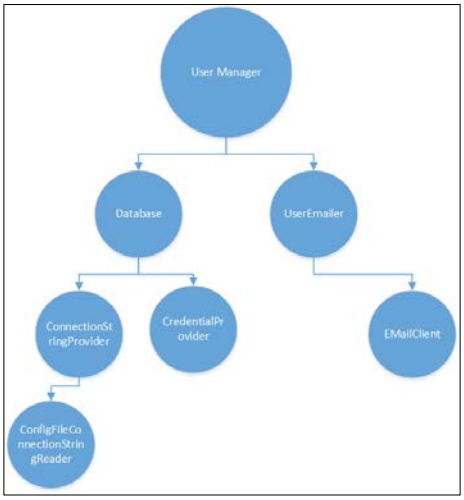
One of the topics we've been talking about continuously during this book is the importance of making your code modular. Small classes are easier to test, provide better reuse, and promote better collaboration for teams. Modular, loosely coupled code is easier to maintain, as changes can be limited. You may remember the example of a ripstop we used earlier.

With modular code of this sort we see a lot of inversion of control. Classes have functionality inserted into them through passing additional classes by their creators. This moves the responsibility for how some portions of the child class work to the parent. For small projects, this is a pretty reasonable approach. As projects get more complicated and dependency graphs get more complicated, manually injecting the functionality becomes more and more difficult. We are still creating objects all over the code base, passing them into created objects so the coupling problem still exists, we've just shifted it up a level.

If we think of object creation as a service, then a solution to this problem presents itself. We can defer the object creation to a central location. This allows us to change the implementations for a given interface in one place, simply and easily. It also allows us to control object lifetime so that we can reuse objects or recreate them every time they are used. If we need to replace one implementation of an interface with another implementation, then we can be confident that we need to only change it in one location. Because the new implementation still fulfils the contract, that is the interface, then all the classes that make use of the interface can remain ignorant of the change.

What's more is that by centralizing object creation it becomes easier to construct objects that depend on other objects. If we look at a dependency graph for a module such as the `UserManager` variable, it is clear that it has a number of dependencies. These dependencies may have additional dependencies and so forth. To build a `UserManager` variable, we not only need to pass in the database, but also `ConnectionStringProvider`, `CredentialProvider`, and `ConfigFileConnectionStringReader`. Goodness, that is going to be a lot of work to create instances of all of these. If we, instead, register implementations of each of these interfaces in a registry, then we need only go to the registry to look up how to make them. This can be automated and the dependencies automatically get injected to all dependencies without a need to explicitly create any of them. This method of solving dependencies is commonly referred to as 'solving the transitive closure'.

A dependency injection framework handles the responsibility of constructing objects. On application set up the dependency injection framework is primed with a combination of names and objects. From this, it creates a registry or a container. When constructing an object through the container, the container looks at the signature of the constructor and attempts to satisfy the arguments on the constructor. Here is an illustration of a dependency graph:



In more statically typed languages such as C# or Java, dependency injection frameworks are commonplace. They usually work by using reflection, a method of using code to extract structural information from other code. When building the container, one specifies an interface and one or more concrete classes that can satisfy the interface. Of course using interfaces and reflection to perform dependency injection requires that the language support both interfaces and introspection.

There is no way to do this in JavaScript. JavaScript has neither direct introspection nor a traditional object inheritance model. A common approach is to use variable names to solve the dependency problem. Consider a class that has a constructor like so:

```
var UserManager = (function () {
  function UserManager(database, userEmailer) {
    this.database = database;
    this.userEmailer = userEmailer;
  }
  return UserManager;
})();
```

The constructor takes two arguments that are very specifically named. When we construct this class through the dependency injection, these two arguments are satisfied by looking through the names registered with the container and passing them into the constructor. However, without introspection how can we extract the names of the parameters so we know what to pass into the constructor?

The solution is actually amazingly simple. The original text of any function in JavaScript is available by simply calling `toString` on it. So, for the constructor given in the preceding code, we can do just do this:

```
UserManager.toString()
```

Now we can parse the string returned to extract the names of the parameters. Care must be taken to parse the text correctly, but it is possible. The popular JavaScript framework, Angular, actually uses this method to do its dependency injection. The result remains relatively preformat. The parsing really only needs to be done once and the results cached, so no additional penalty is incurred.

I won't go through how to actually implement the dependency injection, as it is rather tedious. When parsing the function, you can either parse it using a string-matching algorithm or build a lexer and parser for the JavaScript grammar. The first solution seems easier but it is likely a better decision to try to build up a simple syntax tree for the code into which you're injecting. Fortunately, the entire method body can be treated as a single token, so it is vastly easier than building a fully-fledged parser.

If you're willing to impose a different syntax on the user of your dependency injection framework then you can even go so far as to create your own syntax. The Angular 2.0 dependency injection framework, `di.js`, supports a custom syntax for denoting both places where objects should be injected and for denoting which objects satisfy some requirement.

Using it as a class into which some code needs to be injected, looks like this code, taken from the `di.js` examples page:

```
@Inject(CoffeeMaker, Skillet, Stove, Fridge, Dishwasher)
export class Kitchen {
  constructor(coffeeMaker, skillet, stove, fridge, dishwasher) {
    this.coffeeMaker = coffeeMaker;
    this.skillet = skillet;
    this.stove = stove;
    this.fridge = fridge;
    this.dishwasher = dishwasher;
  }
}
```

The `CoffeeMaker` instance might look like the following code:

```
@Provide(CoffeeMaker)
@Inject(Filter, Container)
export class BodumCoffeeMaker{
  constructor(filter, container){
  …
  }
}
```

You might have also noticed that this example makes use of the `class` keyword. This is because the project is very forward looking and requires the use of `traceur.js` to provide for ES6 class support. We'll learn about `traceur.js` file in the next chapter.

Live post processing

It should be apparent now that running `toString` over a function in JavaScript is a valid way to perform tasks. It seems odd but, really, writing code that emits other code is as old as Lisp or possibly older. When I first came across how dependency injection works in AngularJS, I was both disgusted at the hack and impressed by the ingenuity of the solution.

If it is possible to do dependency injection by interpreting code on the fly, then what more could we do with it? The answer is: quite a lot. The first thing that comes to mind is that you could write domain specific languages.

We talked about DSLs in *Chapter 5*, *Behavioral Patterns*, and even created a very simple one. With the ability to load and rewrite JavaScript, we can take advantage of a syntax that is close to JavaScript but not wholly compatible. When interpreting the DSL, our interpreter would write out additional tokens needed to convert the code to actual JavaScript.

One of the nice features of TypeScript that I've always liked is that parameters to the constructors that are marked as public are automatically transformed into properties on the object. For instance, the TypeScript code that follows:

```
class Axe{
  constructor(public handleLength, public headHeight){}
}
```

Compiles to the following code:

```
var Axe = (function () {
  function Axe(handleLength, headHeight) {
    this.handleLength = handleLength;
    this.headHeight = headHeight;
  }
  return Axe;
})();
```

We could do something similar in our DSL. Starting with the `Axe` definition that follows:

```
class Axe{
  constructor(handleLength, /*public*/ headHeight){}
}
```

We've used a comment here to denote that `headHeight` should be public. Unlike the TypeScript version, we would like our source code to be valid JavaScript. Because comments are included in the `toString` function this works just fine.

The next thing to do is to actually emit new JavaScript from this. I've taken a naïve approach and used regular expressions. This approach would quickly get out of hand and probably only works with the well-formed JavaScript in the `Axe` class:

```
function publicParameters(func){
  var stringRepresentation = func.toString();
  var parameterString = stringRepresentation.match(/^function .*\
    ((.*)\)/)[1];
  var parameters = parameterString.split(",");
  var setterString = "";
  for(var i = 0; i < parameters.length; i++){
    if(parameters[i].indexOf("public") >= 0){
```

```
        var parameterName = parameters[i].split('/')[parameters[i].
          split('/').length-1].trim();
        setterString += "this." +  parameterName + " = " + parameterName
          + ";\n";
      }
    }
    var functionParts = stringRepresentation.match(/(^.*{)([\s\S]*)/);
    return functionParts[1] + setterString + functionParts[2];
  }

  console.log(publicParameters(Axe));
```

Here we extract the parameters to the function and check for those that have the `public` annotation. The result of this function can be passed back into eval for use in the current object or written out to a file if we're using this function in a pre-processor. Typically use of eval in JavaScript is discouraged.

There are tons of different things that can be done using this sort of processing. Even without string post-processing there are some interesting programming concept we can explore by just wrapping methods.

Aspect oriented programming

Modularity of software is a great feature, the majority of this book has been about modularity and its advantages. However, there are some features of software that span the entire system. Security is a great example of this.

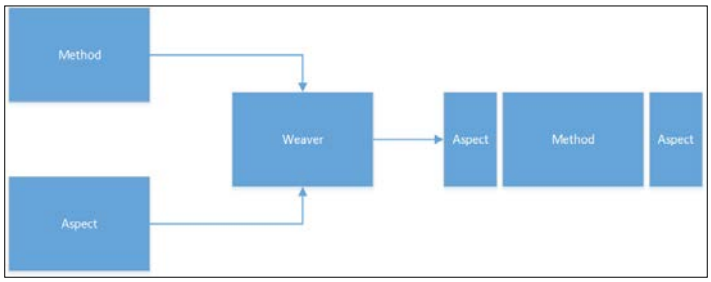
We would like to have similar security code in all the modules of the application to check that people are, in fact, authorized to perform some action. So if we have a function of the sort:

```
var GoldTransfer = (function () {
  function GoldTransfer() {
  }
  GoldTransfer.prototype.SendPaymentOfGold = function (amountOfGold,
    destination) {
    var user = Security.GetCurrentUser();
    if (Security.IsAuthorized(user, "SendPaymentOfGold")) {
      //send actual payment
    } else {
      return { success: 0, message: "Unauthorized" };
    }
  };
  return GoldTransfer;
})();
```

We can see that there is a fair bit of code in place to check if a user is authorized. This same boilerplate code is used elsewhere in the application. In fact, with this being a high security application, the security checks are in place in every public function. All is well until we need to make a change to the common security code. This change needs to take place in every single public function in the application. We can refactor our application all we want, but the truth remains: we need to have at least some code in each of the public methods to perform a security check. This is known as a cross-cutting concern.

There are other instances of cross-cutting concerns in most large applications. Logging is a great example, as is database access and performance instrumenting. **Aspect oriented programming** (**AOP**) presents a way to minimize the repeated code through a process known as **weaving**.

An aspect is a piece of code that can intercept method calls and change them. On the .Net platform there is a tool called PostSharp that does aspect weaving and, on the Java platform, one called AspectJ. These tools hook into the build pipeline and modify the code after it has been transformed into instructions. This allows code to be injected wherever needed. The source code appears unchanged but the compiled output now includes calls to the aspect. Aspects solve the cross cutting concern by being injected into existing code. Here you can see the application of an aspect to a method through a weaver:



Of course we don't have the luxury of a design-time compile step in most JavaScript workflows. Fortunately, we've already seen some approaches that would allow us to implement cross cuts using JavaScript. The first thing we need is the wrapping of methods that we saw in the testing chapter. The second is the `tostring` abilities from earlier in this chapter.

There are some AOP libraries already in existence for JavaScript that may be a good bet to explore. However, we can implement a simple interceptor here. First let's decide on the grammar for requesting injection. We'll use the same idea of comments from earlier to denote methods that require interception. We'll just make the first line in the method a comment that reads `aspect(<name of aspect>).`

To start we'll take a slightly modified version of our same `GoldTransfer` class from earlier:

```
class GoldTransfer {
  SendPaymentOfGold(amountOfGold, destination) {
    var user = Security.GetCurrentUser();
    if (Security.IsAuthorized(user, "SendPaymentOfGold")) {
    }
    else {
     return { success: 0, message: "Unauthorized" };
    }
  }
}
```

We've stripped out all the security stuff that used to exist in it and added a console log so we can see that it actually works. Next we'll need an aspect to weave into it:

```
class ToWeaveIn {
   BeforeCall() {
    console.log("Before!");
  }
  AfterCall() {
    console.log("After!");
  }
}
```

For this we use a simple class that has a `BeforeCall` and `AfterCall` method, one which is called before and one which is called after the original method. We don't need to use eval in this case so the interceptions are safer:

```
function weave(toWeave, toWeaveIn, toWeaveInName) {
  for (var property in toWeave.prototype) {
    var stringRepresentation = toWeave.prototype[property].toString();
    console.log(stringRepresentation);
    if (stringRepresentation.indexOf("@aspect(" + toWeaveInName + ")")
      >= 0) {
      toWeave.prototype[property + "_wrapped"] = toWeave.
        prototype[property];
      toWeave.prototype[property] = function () {
      toWeaveIn.BeforeCall();
      toWeave.prototype[property + "_wrapped"]();
      toWeaveIn.AfterCall();
    };
    }
  }
}
```

This interceptor can easily be modified to a shortcut and return something before the main method body is called. It can also be changed so that the output of the function can be modified by simply tracking the output from the wrapped method and then modifying it in the `AfterCall` method.

This is a fairly lightweight example of AOP. There are some frameworks in existence for JavaScript AOP, but perhaps the best approach is to make use of a precompiler or macro language.

Mixins

As we saw much earlier in this book, the inheritance pattern for JavaScript is different from the typical pattern seen in languages like C# and Java. JavaScript uses prototype inheritance that allows adding functions to a class quite easily and from multiple sources. Prototype inheritance allows for adding methods from multiple sources in a similar fashion to the much-maligned multiple-inheritance. The primary criticism of multiple inheritance is that it is difficult to understand which overload of a method will be called in a situation. This problem is somewhat alleviated in a prototype inheritance model. Thus we can feel comfortable using the approach of adding functionality from several sources, which is known as mixins.

A mixin is a chunk of code which can be added to existing classes to expand their functionality. They make the most sense in scenarios where the functions need to be shared between disparate classes where an inheritance relationship is too strong.

Let's imagine a scenario where this sort of functionality would be handy. In the land of Westeros, death is not always as permanent as in our world. However, those who return from the dead may not be exactly as they were when they were alive. While much of the functionality is shared between `Person` and `ReanimatedPerson`, they are not close enough to have an inheritance relationship. In this code you can see the `extend` function of underscore used to add mixins to our two people classes. It is possible to do this without `underscore` but, as mentioned earlier, there are some complex edge cases around `extends` which make using a library handy:

```
var _ = require("underscore");
export class Person{
}
export class ReanimatedPerson{
}
export class RideHorseMixin{
  public Ride(){
```

```
      console.log("I'm on a horse!");
    }
  }

  var person = new Person();
  var reanimatedPerson = new ReanimatedPerson();
  _.extend(person, new RideHorseMixin());
  _.extend(reanimatedPerson, new RideHorseMixin());

  person.Ride();
  reanimatedPerson.Ride();
```

Mixins provide a mechanism to share functionality between diverse objects but do pollute the prototype structure.

Macros

Preprocessing code through macros is not a new idea. It was, and probably still is, very popular for C and C++. In fact, if you take a look at some of the source code for the Gnu utilities for Linux they are written almost entirely in macros. Macros are notorious for being hard to understand and debug. For a time, newly-created languages like Java and C# did not support macros for exactly this reason.

That being said, even more recent languages like Rust and Julia have brought the idea of macros back. These languages were influenced by the macros from the Scheme language, a dialect of Lisp. The difference between C macros and Lisp/Scheme macros is that the C versions are textual while the Lisp/Scheme ones are structural. This means that C macros are just glorified find/replace tools while Scheme macros are aware of the **abstract syntax tree** (**AST**) around them, allowing them to be much more powerful.

The AST for Scheme is a far simpler construct than that of JavaScript. Nonetheless, there is a very interesting project called `Sweet.js` that tries to create structural macros for JavaScript.

`Sweet.js` plugs into the JavaScript build pipeline and modified JavaScript source code using one or more macros. There are a number of fully-fledged JavaScript transpilers, that is compilers that emit JavaScript. These compilers are problematic for sharing code between multiple projects. Their code is so different that there is no real way to share it. `Sweet.js` supports multiple macros being expanded in a single step. This allows for much better code sharing. The reusable bits are a smaller size and more easy to run together.

A simple example of `Sweet.js` is as follows:

```
let var = macro {
  rule { [$var (,) ...] = $obj:expr } => {
    var i = 0;
    var arr = $obj;
    $(var $var = arr[i++]) (;) ...
  }

  rule { $id } => {
    var $id
  }
}
```

The macro here provides ECMAScript-2015-style deconstructors that split an array into tree fields. The macro matches an array assignment and also regular assignment. For regular assignment the macro simply returns the identity, while for assignment of an array it will explode the text and replace it.

For instance, if you run it over the following:

```
var [foo, bar, baz] = arr;
```

Then, the result will be the following:

```
var i = 0;
var arr$2 = arr;
var foo = arr$2[i++];
var bar = arr$2[i++];
var baz = arr$2[i++];
```

This is just one example macro. The power of macros is really quite spectacular. Macros can create an entirely new language or change very minor things. They can be easily plugged in to fit any sided requirement.

Tips and tricks

Using name-based dependency injection allows for conflicts between names. In order to avoid conflicts it may be worth prefacing your injected arguments with a special character. For instance, AngularJS uses the `$` sign to denote an injected term.

Several times in this chapter I've mentioned the JavaScript build pipeline. It may seem odd that we have to build an interpreted language. However, there are certain optimizations and process improvements that may result from building JavaScript. There are a number of tools that can be used to help building JavaScript. Tools such as Grunt and Gulp are specifically designed to perform JavaScript and web tasks but you can also make use of traditional build tools such as Rake, Ant, or even Make.

Summary

In this chapter we covered a number of advanced JavaScript patterns. Of these patterns it's my belief that dependency injection and macros are the most useful to us. You may not necessarily want to use them on every project. When approaching problems simply being aware of the possible solutions may change your approach to the problem.

Throughout this book I have talked extensively about the next versions of JavaScript. However, you don't need to wait until some future time to make use of many of these tools. Today, there are ways to compile newer versions of JavaScript down to the current version of JavaScript. The final chapter will explore a number of these tools and techniques.

14
ECMAScript-2015/2016 Solutions Today

I cannot count the number of times I have mentioned upcoming versions of JavaScript in this book, rest assured that it's a large number. It is somewhat frustrating that the language is not keeping pace with the requirements of application developers. Many of the approaches we've discussed become unnecessary with a newer version of JavaScript. There are, however, some ways to get the next version of JavaScript working today.

In this chapter we'll look at a couple of these, specifically:

- Typescript
- BabelJS

TypeScript

There is no shortage of languages that compile to JavaScript. CoffeeScript is perhaps the best known example of one of these languages, although the Google web toolkit that compiles Java to JavaScript was also once very popular. Never ones to be left behind or use somebody else's solution, Microsoft released a language called TypeScript in 2012. It is designed to be a superset of JavaScript in the same way that C++ is a superset of C. This means that all syntactically valid JavaScript code is also syntactically valid TypeScript code.

Microsoft itself is making heavy use of TypeScript in some of its larger web properties. Both Office 365 and Visual Studio Online have significant code bases written in TypeScript. These projects actually predate TypeScript by a significant margin. The transition from JavaScript to TypeScript was reportedly quite easy due to the fact that it is a superset of JavaScript.

One of the design goals for TypeScript was to make it as compatible as possible with ECMAScript-2015 and future versions. This means that TypeScript supports some, although certainly not all, of the features of ECMAScript-2016, as well as a healthy chunk of ECMAScript-2015. Two significant features from ECMAScript-2016 which are partially supported by Typescript are decorators and async/await.

Decorators

In an earlier chapter we explored **aspect oriented programming** (**AOP**). With AOP we wrap function with interceptors. Decorators offer an easy way of doing this. Let's say that we have a class which dispatches messages in Westeros. Obviously there are no phones or internet there, so messages are dispatched via crows. It would be very helpful if we could spy on these messages. Our `CrowMessenger` class looks like the following:

```
class CrowMessenger {
  @spy
  public SendMessage(message: string) {
    console.log(`Send message is ${message}`);
  }
}
var c = new CrowMessenger();
var r = c.SendMessage("Attack at dawn");
```

You may note the `@spy` annotation on the `SendMessage` method. This is simply another function which intercepts and wraps the function. Inside of the spy we have access to the function descriptor. As you can see in the following code, we take the descriptor and manipulate it to capture the argument sent to the `CrowMessenger` class:

```
function spy(target: any, key: string, descriptor?: any) {
  if(descriptor === undefined) {
    descriptor = Object.getOwnPropertyDescriptor(target, key);
  }
  var originalMethod = descriptor.value;

  descriptor.value =  function (...args: any[]) {
    var arguments = args.map(a => JSON.stringify(a)).join();
    var result = originalMethod.apply(this, args);
    console.log(`Message sent was: ${arguments}`);
    return result;
  }
  return descriptor;
}
```

Spys would obviously be very useful for testing functions. Not only can we spy on the values here but we could replace the input and output to the function. Consider the following:

```
descriptor.value =  function (...args: any[]) {
  var arguments = args.map(a => JSON.stringify(a)).join();
  var result = "Retreat at once";
  console.log(`Message sent was: ${arguments}`);
  return result;
}
```

Decorators can be used for purposes other than AOP. For instance, you could annotate the properties on an object as serializable and use the annotations to control custom JSON serialization. It is my suspicion that decorators will become more useful and powerful as decorators become supported. Already Angular 2.0 is making extensive use of decorators.

Async/Await

In *Chapter 7*, *Reactive Programming*, we spoke about how the callback nature of JavaScript programming makes code very confusing. Nowhere is this more apparent than trying to chain together a series of asynchronous events. We rapidly fall into a trap of code, which looks like the following:

```
$.post("someurl", function(){
  $.post("someotherurl", function(){
    $.get("yetanotherurl", function(){
      navigator.geolocation.getCurrentPosition(function(location){
        ...
      })
    })
  })
})
```

Not only is this code difficult to read, it is nearly impossible to understand. The async/await syntax, which is borrowed from C#, allows for writing your code in a much more succinct fashion. Behind the scenes generators are used (or abused, if you like) to create the impression of true async/await. Let's look at an example. In the preceding code we made use of the geolocation API which returns the location of a client. It is asynchronous as it performs some IO with the user's machine to get a real world location. Our specification calls for us to get the user's location, post it back to the server, and then get an image:

```
navigator.geolocation.getCurrentPosition(function(location){
  $.post("/post/url", function(result){
    $.get("/get/url", function(){
```

```
    });
  });
});
```

If we now introduce async/await, this can become the following:

```
async function getPosition(){
  return await navigator.geolocation.getCurrentPosition();
}
async function postUrl(geoLocationResult){
  return await $.post("/post/url");
}
async function getUrl(postResult){
  return await $.get("/get/url");
}
async function performAction(){
  var position = await getPosition();
  var postResult = await postUrl(position);
  var getResult = await getUrl(postResult);
}
```

This code assumes that all `async` responses return promises which are a construct that contains a status and a result. As it stands, most `async` operations do not return promises but there are libraries and utilities to convert callbacks to promises. As you can see, the syntax is much cleaner and easier to follow than the callback mess.

Typing

As well as the ECMAScript-2016 features we've mentioned in the preceding section, TypeScript has a rather intriguing typing system incorporated into it. One of the nicest parts of JavaScript is that it is a dynamically typed language. We've seen, repeatedly, how, not being burdened by types has saved us time and code. The typing system in TypeScript allows you to use as much or as little typing as you deem to be necessary. You can give variables a type by declaring them with the following syntax:

```
var a_number: number;
var a_string: string;
var an_html_element: HTMLElement;
```

Once a variable has a type assigned to it, the TypeScript compiler will use that not only to check that variable's usage, but also to infer what other types may be derived from that class. For example, consider the following code:

```
var numbers: Array<number> = [];
numbers.push(7);
```

```
numbers.push(9);
var unknown = numbers.pop();
```

Here, the TypeScript compiler will know that `unknown` is a number. If you attempt to use it as something else, say as the following string:

```
console.log(unknown.substr(0,1));
```

Then the compiler will throw an error. However, you don't need to assign a type to any variable. This means that you can tune the degree to which the type checking is run. While it sounds odd, it is actually a brilliant solution for introducing the rigour of type checking without losing the pliability of JavaScript. The typing is only enforced during compilation, once the code is compiled to JavaScript, any hint that there was typing information associated with a field disappears. As a result, the emitted JavaScript is actually very clean.

If you're interested in typing systems and know words like contravariant and can discuss the various levels of gradual typing, then TypeScript's typing system may be well worth your time to investigate.

All the examples in this book were originally written in TypeScript and then compiled to JavaScript. This was done to improve the accuracy of the code and generally to save me from messing up quite so frequently. I'm horribly biased but I think that TypeScript is really well done and certainly better than writing pure JavaScript.

There is no support for typing in future versions of JavaScript. Thus, even with all the changes coming to future versions of JavaScript, I still believe that TypeScript has its place in providing compile time type checking. I never cease to be amazed by the number of times that the type checker has saved me from making silly mistakes when writing TypeScript.

BabelJS

An alternative to TypeScript is to use the BabelJS compiler. This is an open source project ECMAScript-2015 and beyond to the equivalent ECMAScript 5 JavaScript. A lot of the changes put in place for ECMAScript-2015 are syntactic niceties, so they can actually be represented in ECMAScript 5 JavaScript, although not as succinctly or as pleasantly. We've seen that already using class-like structures in ES 5. BabelJS is written in JavaScript, which means that the compilation from ECMAScript-2015 to ES 5 is possible directly on a web page. Of course, as seems to be the trend with compilers, the source code for BabelJS makes use of ES 6 constructs, so BabelJS must be used to compile BabelJS.

At the time of writing, the list of ES6 functions that are supported by BabelJS are extensive:

- Arrow functions
- Classes
- Computed property names
- Default parameters
- Destructuring assignment
- Iterators and for of
- Generator comprehension
- Generators
- Modules
- Numeric literals
- Property method assignment
- Object initializer shorthand
- Rest parameters
- Spread
- Template literals
- Promises

BabelJS is multi-purpose JavaScript compiler, so compiling ES-2015 code is simply one of the many things it can do. There are numerous plugins which provide a wide array of interesting functionality. For instance, the "Inline environmental variable" plugin inserts compile time variables, allowing for conditional compilation based on environments.

There is already a fair bit of documentation available on how each of these features work so we won't go over all of them.

Setting up Babel JS is a fairly simple exercise if you already have node and npm installed:

```
npm install -g babel-cli
```

This will create a BabelJS binary which can do compilation like so:

```
babel  input.js --o output.js
```

For most use cases you'll want to investigate using build tools such as Gulp or Grunt, which can compile many files at once and perform any number of post-compilation steps.

Classes

By now you should be getting sick of reading about different ways to make classes in JavaScript. Unfortunately for you I'm the one writing this book so let's look at one final example. We'll use the same castle example from earlier.

Modules within files are not supported in BabelJS. Instead, files are treated as modules, which allows for dynamic loading of modules in a fashion not unlike `require.js`. Thus we'll drop the module definition from our castle and stick to just the classes. One other feature that exists in TypeScript and not ES 6 is prefacing a parameter with `public` to make it a public property on a class. Instead we make use of the `export` directive.

Once we've made these changes, the source ES6 file looks like the following:

```
export class BaseStructure {
  constructor() {
    console.log("Structure built");
  }
}

export class Castle extends BaseStructure {
  constructor(name){
    this.name = name;
    super();
  }
  Build(){
    console.log("Castle built: " + this.name);
  }
}
```

The resulting ES 5 JavaScript looks like the following:

```
"use strict";

var _createClass = function () { function defineProperties(target,
props) { for (var i = 0; i < props.length; i++) { var descriptor
= props[i]; descriptor.enumerable = descriptor.enumerable ||
false; descriptor.configurable = true; if ("value" in descriptor)
descriptor.writable = true; Object.defineProperty(target, descriptor.
key, descriptor); } } return function (Constructor, protoProps,
staticProps) { if (protoProps) defineProperties(Constructor.
prototype, protoProps); if (staticProps) defineProperties(Constructor,
staticProps); return Constructor; }; }();
```

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

function _possibleConstructorReturn(self, call) { if (!self) { throw
  new ReferenceError("this hasn't been initialised - super() hasn't
  been called"); } return call && (typeof call === "object" || typeof
  call === "function") ? call : self; }

function _inherits(subClass, superClass) { if (typeof superClass
  !== "function" && superClass !== null) { throw new TypeError("Super
  expression must either be null or a function, not " + typeof
  superClass); } subClass.prototype = Object.create(superClass &&
  superClass.prototype, { constructor: { value: subClass, enumerable:
  false, writable: true, configurable: true } }); if (superClass)
  Object.setPrototypeOf ? Object.setPrototypeOf(subClass, superClass)
  : subClass.__proto__ = superClass; }

function _classCallCheck(instance, Constructor) { if (!(instance
  instanceof Constructor)) { throw new TypeError("Cannot call a class
  as a function"); } }

var BaseStructure = exports.BaseStructure = function BaseStructure() {
  _classCallCheck(this, BaseStructure);
  console.log("Structure built");
};

var Castle = exports.Castle = function (_BaseStructure) {
  _inherits(Castle, _BaseStructure);
  function Castle(name) {
    _classCallCheck(this, Castle);
    var _this = _possibleConstructorReturn(this, Object.
      getPrototypeOf(Castle).call(this));
    _this.name = name;
    return _this;
  }
  _createClass(Castle, [{
    key: "Build",
    value: function Build() {
      console.log("Castle built: " + this.name);
    }
  }]);
  return Castle;
}(BaseStructure);
```

Right away it is apparent that the code produced by BabelJS is not as clean as the code from TypeScript. You may also have noticed that there are some helper functions employed to handle inheritance scenarios. There are also a number of mentions of `"use strict";`. This is an instruction to the JavaScript engine that it should run in strict mode.

Strict mode prevents a number of dangerous JavaScript practices. For instance, in some JavaScript interpreters it is legal to use a variable without declaring it first:

```
x = 22;
```

This will throw an error if `x` has not previously been declared:

```
var x = 22;
```

Duplicating properties in objects is disallowed, as well as double declaring a parameter. There are a number of other practises that `"use strict";` will treat as errors. I like to think of `"use strict";` as being similar to treating all warnings as errors. It isn't, perhaps, as complete as `-Werror` in GCC but it is still a good idea to use strict mode on new JavaScript code bases. BabelJS simply enforces that for you.

Default parameters

Not a huge feature but a real nicety in ES 6 is the introduction of default parameters. It has always been possible to call a function in JavaScript without specifying all the parameters. Parameters are simply populated from left to right until there are no more values and all remaining parameters are given undefined.

Default parameters allow setting a value other than undefined for parameters that aren't filled out:

```
function CreateFeast(meat, drink = "wine"){
  console.log("The meat is: " + meat);
  console.log("The drink is: " + drink);
}
CreateFeast("Boar", "Beer");
CreateFeast("Venison");
```

This will output the following:

```
The meat is: Boar
The drink is: Beer
The meat is: Venison
The drink is: wine
```

The JavaScript code produced is actually very simple:

```
"use strict";
function CreateFeast(meat) {
  var drink = arguments.length <= 1 || arguments[1] === undefined ?
    "wine" : arguments[1];
  console.log("The meat is: " + meat);
  console.log("The drink is: " + drink);
}
CreateFeast("Boar", "Beer");
CreateFeast("Venison");
```

Template literals

On the surface, template literals seem to be a solution for the lack of string interpolation in JavaScript. In some languages, such as Ruby and Python, you can inject substitutions from the surrounding code directly into a string without having to pass them into some sort of string formatting function. For instance, in Ruby you can do the following:

```
name= "Stannis";
print "The one true king is ${name}"
```

This will bind the `${name}` parameter to the name from the surrounding scope.

ES6 supports template literals that allow something similar in JavaScript:

```
var name = "Stannis";
console.log(`The one true king is ${name}`);
```

It may be difficult to see but that string is actually surrounded by backticks and not quotation marks. Tokens to bind to the scope are denoted by `${}`. Within the braces you can put complex expressions such as:

```
var army1Size = 5000;
var army2Size = 3578;
console.log(`The surviving army will be ${army1Size > army2Size ?
  "Army 1": "Army 2"}`);
```

The BabelJS compiled version of this code simply substitutes appending strings for the string interpolation:

```
var army1Size = 5000;
var army2Size = 3578;
console.log(("The surviving army will be " + (army1Size > army2Size ?
  "Army 1" : "Army 2")));
```

Template literals also solve a number of other problems. New line characters inside of a template literal are legal, meaning that you can use template literals to create multiline strings.

With the multiline string idea in mind, it seems like template literals might be useful for building domain specific languages: a topic we've seen a number of times already. The DSL can be embedded in a template literal and then values from outside plugged in. An example might be using it to hold HTML strings (certainly a DSL) and inserting values in from a model. These could, perhaps, take the place of some of the template tools in use today.

Block bindings with let

The scoping of variables in JavaScript is weird. If you define a variable inside a block, say inside an `if` statement, then that variable is still available outside of the block. For example, see the following code:

```
if(true)
{
  var outside = 9;
}
console.log(outside);
```

This code will print `9`, even though the variable outside is clearly out of scope. At least it is out of scope if you assume that JavaScript is like other C-syntax languages and supports block level scoping. The scoping in JavaScript is actually function level. Variables declared in code blocks like those found attached to `if` and `for` loop statements are hoisted to the beginning of the function. This means that they remain in scope for the entirety of the function.

ES 6 introduces a new keyword, `let`, which scopes variables to the block level. This sort of variable is ideal for use in loops or to maintain proper variable values inside an `if` statement. Traceur implements support for block scoped variables. However, the support is experimental at the moment due to performance implications.

Consider the following code:

```
if(true)
{
  var outside = 9;
  et inside = 7;
}
console.log(outside);
console.log(inside);
```

This will compile to the following:

```
var inside$__0;
if (true) {
  var outside = 9;
  inside$__0 = 7;
}
console.log(outside);
console.log(inside);
```

You can see that the inner variable is replaced with a renamed one. Once outside the block, the variable is no longer replaced. Running this code will report that inside is undefined when the `console.log` method occurs.

In production

BabelJS is a very powerful tool for replicating many of the structures and features of the next version of JavaScript today. However, the code generated is never going to be quite as efficient as having native support for the constructs. It may be worth benchmarking the generated code to ensure that it continues to meet the performance requirements of your project.

Tips and tricks

There are two excellent libraries for working with collections functionally in JavaScript: Underscore.js and Lo-Dash. Used in combination with TypeScript or BabelJS they have a very pleasant syntax and provide immense power.

For instance, finding all the members of a collection that satisfy a condition using Underscore looks like the following:

```
_.filter(collection, (item) => item.Id > 3);
```

This code will find all the items where the ID is greater than `3`.

Either of these libraries is one of the first things I add to a new project. Underscore is actually bundled with backbone.js, an MVVM framework.

Tasks for Grunt and Gulp exist for compiling code written in TypeScript or BabelJS. There is, of course, also good support for TypeScript in much of Microsoft's development tool chain, although BabelJS is currently not supported directly.

Summary

As the functionality of JavaScript expands, the need for third party frameworks and even transpilers starts to drop off. The language itself replaces many of these tools. The end game for tools like jQuery is that they are no longer required as they have been absorbed into the ecosystem. For many years the velocity of web browsers has been unable to keep pace with the rate of change of people's desires.

There is a large effort behind the next version of AngularJS but great efforts are being made to align the new components with the upcoming web component standards. Web components won't fully replace AngularJS but Angular will end up simply enhancing web components.

Of course the idea that there won't be a need for any frameworks or tools is ridiculous. There is always going to be a new method of solving a problem and new libraries and frameworks will show up. The opinions of people on how to solve problems is also going to differ. That's why there is space in the market for the wide variety of MVVM frameworks that exist.

Working with JavaScript can be a much more pleasant experience if you make use of ES6 constructs. There are a couple of possible approaches to doing so, which of these is best suited to your specific problem is a matter for closer investigation.

Module 3

Functional Programming in JavaScript

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

1
The Powers of JavaScript's Functional Side – a Demonstration

Introduction

For decades, functional programming has been the darling of computer science aficionados, prized for its mathematical purity and puzzling nature that kept it hidden in dusty computer labs occupied by data scientists and PhD hopefuls. But now, it is going through a resurgence, thanks to modern languages such as **Python**, **Julia**, **Ruby**, **Clojure** and—last but not least—**JavaScipt**.

JavaScript, you say? The web's scripting language? Yes!

JavaScript has proven to be an important technology that isn't going away for quite a while. This is largely due to the fact that it is capable of being reborn and extended with new frameworks and libraries, such as **backbone.js**, **jQuery**, **Dojo**, **underscore.js**, and many more. *This is directly related to JavaScript's true identity as a functional programming language*. An understanding of functional programming with JavaScript will be welcome and useful for a long time for programmers of any skill level.

Why so? Functional programming is very powerful, robust, and elegant. It is useful and efficient on large data structures. It can be very advantageous to use JavaScript—a client-side scripting language, as a functional means to manipulate the DOM, sort API responses or perform other tasks on increasingly complex websites.

In this book, you will learn everything you need to know about functional programming with JavaScript: how to empower your JavaScript web applications with functional programming, how to unlock JavaScript's hidden powers, and how to write better code that is both more powerful and—because it is smaller—easier to maintain, faster to download, and takes less overhead. You will also learn the core concepts of functional programming, how to apply them to JavaScript, how to side-step the caveats and issues that may arise when using JavaScript as a functional language, and how to mix functional programming with object-oriented programming in JavaScript.

But before we begin, let's perform an experiment.

The demonstration

Perhaps a quick demonstration will be the best way to introduce functional programming with JavaScript. We will perform the same task using JavaScript—once using traditional, native methods, and once with functional programming. Then, we will compare the two methods.

The application – an e-commerce website

In pursuit of a real-world application, let's say we need an e-commerce web application for a mail-order coffee bean company. They sell several types of coffee and in different quantities, both of which affect the price.

Imperative methods

First, let's go with the procedural route. To keep this demonstration down to earth, we'll have to create objects that hold the data. This allows the ability to fetch the values from a database if we need to. But for now, we'll assume they're statically defined:

```
// create some objects to store the data.
var columbian = {
  name: 'columbian',
  basePrice: 5
};
var frenchRoast = {
  name: 'french roast',
  basePrice: 8
};
var decaf = {
  name: 'decaf',
```

```
  basePrice: 6
};

// we'll use a helper function to calculate the cost
// according to the size and print it to an HTML list
function printPrice(coffee, size) {
  if (size == 'small') {
    var price = coffee.basePrice + 2;
  }
  else if (size == 'medium') {
    var price = coffee.basePrice + 4;
  }
  else {
    var price = coffee.basePrice + 6;
  }

// create the new html list item
  var node = document.createElement("li");
  var label = coffee.name + ' ' + size;
  var textnode = document.createTextNode(label+' price: $'+price);
  node.appendChild(textnode);
  document.getElementById('products').appendChild(node);
}

// now all we need to do is call the printPrice function
// for every single combination of coffee type and size
printPrice(columbian, 'small');
printPrice(columbian, 'medium');
printPrice(columbian, 'large');
printPrice(frenchRoast, 'small');
printPrice(frenchRoast, 'medium');
printPrice(frenchRoast, 'large');
printPrice(decaf, 'small');
printPrice(decaf, 'medium');
printPrice(decaf, 'large');
```



Downloading the example code

You can download example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

As you can see, this code is very basic. What if there were many more coffee styles than just the three we have here? What if there were 20? 50? What if, in addition to size, there were organic and non-organic options. That could increase the lines of code extremely quickly!

Using this method, we are telling the machine what to print for each coffee type and for each size. This is fundamentally what is wrong with imperative code.

Functional programming

While imperative code tells the machine, step-by-step, what it needs to do to solve the problem, functional programming instead seeks to describe the problem mathematically so that the machine can do the rest.

With a more functional approach, the same application can be written as follows:

```
// separate the data and logic from the interface
var printPrice = function(price, label) {
  var node = document.createElement("li");
  var textnode = document.createTextNode(label+' price: $'+price);
  node.appendChild(textnode);
  document.getElementById('products 2').appendChild(node);
}

// create function objects for each type of coffee
var columbian = function(){
  this.name = 'columbian';
  this.basePrice = 5;
};
var frenchRoast = function(){
  this.name = 'french roast';
  this.basePrice = 8;
};
var decaf = function(){
  this.name = 'decaf';
  this.basePrice = 6;
};

// create object literals for the different sizes
var small = {
  getPrice: function(){return this.basePrice + 2},
  getLabel: function(){return this.name + ' small'}
};
var medium = {
```

```
  getPrice: function(){return this.basePrice + 4},
  getLabel: function(){return this.name + ' medium'}
};
var large = {
  getPrice: function(){return this.basePrice + 6},
  getLabel: function(){return this.name + ' large'}
};

// put all the coffee types and sizes into arrays
var coffeeTypes = [columbian, frenchRoast, decaf];
var coffeeSizes = [small, medium, large];

// build new objects that are combinations of the above
// and put them into a new array
var coffees = coffeeTypes.reduce(function(previous, current) {
  var newCoffee = coffeeSizes.map(function(mixin) {
    // `plusmix` function for functional mixins, see Ch.7
    var newCoffeeObj = plusMixin(current, mixin);
    return new newCoffeeObj();
  });
  return previous.concat(newCoffee);
},[]);

// we've now defined how to get the price and label for each
// coffee type and size combination, now we can just print them
coffees.forEach(function(coffee){
  printPrice(coffee.getPrice(),coffee.getLabel());
});
```

The first thing that should be obvious is that it is much more modular. This makes adding a new size or a new coffee type as simple as shown in the following code snippet:

```
var peruvian = function(){
  this.name = 'peruvian';
  this.basePrice = 11;
};

var extraLarge = {
  getPrice: function(){return this.basePrice + 10},
  getLabel: function(){return this.name + ' extra large'}
};

coffeeTypes.push(Peruvian);
coffeeSizes.push(extraLarge);
```

Arrays of coffee objects and size objects are "mixed" together,—that is, their methods and member variables are combined—with a custom function called `plusMixin` (see *Chapter 7*, *Functional and Object-oriented Programming in JavaScript*). The coffee type classes contain the member variables and the sizes contain methods to calculate the name and price. The "mixing" happens within a `map` operation, which applies a pure function to each element in an array and returns a new function inside a `reduce()` operation—another higher-order function similar to the `map` function, except that all the elements in the array are combined into one. Finally, the new array of all possible combinations of types and sizes is iterated through with the `forEach()` method The `forEach()` method is yet another higher-order function that applies a callback function to each object in an array. In this example, we provide it as an anonymous function that instantiates the objects and calls the `printPrice()` function with the object's `getPrice()` and `getLabel()` methods as arguments.

Actually, we could make this example even more functional by removing the `coffees` variable and chaining the functions together—another little trick in functional programming.

```
coffeeTypes.reduce(function(previous, current) {
  var newCoffee = coffeeSizes.map(function(mixin) {
    // `plusMixin` function for functional mixins, see Ch.7
    var newCoffeeObj = plusMixin(current, mixin);
    return new newCoffeeObj();
  });
  return previous.concat(newCoffee);
},[]).forEach(function(coffee) {
  printPrice(coffee.getPrice(),coffee.getLabel());
});
```

Also, the control flow is not as top-to-bottom as the imperative code was. In functional programming, the `map()` function and other higher-order functions take the place of `for` and `while` loops and very little importance is placed on the order of execution. This makes it a little trickier for newcomers to the paradigm to read the code but, once you get the hang of it, it's not hard at all to follow and you'll see that it is much better.

This example barely touched on what functional programming can do in JavaScript. Throughout this book, you will see even more powerful examples of the functional approach.

Summary

First, the benefits of adopting a functional style are clear.

Second, don't be scared of functional programming. Yes, it is often thought of as pure logic in the form of computer language, but we don't need to understand **Lambda calculus** to be able to apply it to everyday tasks. The fact is, by allowing our programs to be broken down into smaller pieces, they're easier to understand, simpler to maintain, and more reliable. `map()` and `reduce()` function's are lesser-known built-in functions in JavaScript, but we'll look at them.

JavaScript is a scripting language, interactive and approachable. No compiling is necessary. We don't even need to download any development software, your favorite browser works as the interpreter and as the development environment.

Interested? Alright, let's get started!

2
Fundamentals of Functional Programming

By now, you've seen a small glimpse of what functional programming can do. But what exactly is functional programming? What makes one language functional and not another? What makes one programming style functional and not another?

In this chapter, we will first answer these questions and then cover the core concepts of functional programming:

- Using functions and arrays for control flow
- Writing pure functions, anonymous functions, recursive functions, and more
- Passing functions around like objects
- Utilizing the `map()`, `filter()`, and `reduce()` functions

Functional programming languages

Functional programming languages are languages that facilitate the functional programming paradigm. At the risk of oversimplifying, we could say that, if a language includes the features required for functional programming, then it is a functional language—as simple as that. In most cases, it's the programming style that truly determines whether a program is functional or not.

What makes a language functional?

Functional programming cannot be performed in C. Functional programming cannot be performed in Java (without a lot of cumbersome workarounds for "almost" functional programming). Those and many more languages simply don't contain the constructs to support it. They are purely object-oriented and strictly non-functional languages.

At the same time, object-oriented programming cannot be performed on purely functional languages, such as **Scheme**, **Haskell**, and **Lisp,** just to name a few.

However, there are certain languages that support both models. Python is a famous example, but there are others: Ruby, Julia, and—here's the one we're interested in—JavaScript. How can these languages support two design patterns that are very different from each other? They contain the features required for both programming paradigms. However, in the case of JavaScript, the functional features are somewhat hidden.

But really, it's a little more involved than that. So what makes a language functional?

Characteristic	**Imperative**	**Functional**
Programming Style	Perform step-by-step tasks and manage changes in state	Define what the problem is and what data transformations are needed to achieve the solution
State Changes	Important	Non-existent
Order of Execution	Important	Not as important
Primary Flow Control	Loops, conditionals, and function calls	Function calls and recursion
Primary Manipulation Unit	Structures and class objects	Functions as first-class objects and data sets

The syntax of the language must allow for certain design patterns, such as an inferred type system, and the ability to use anonymous functions. Essentially, the language must implement Lambda calculus. Also, the interpreter's evaluation strategy should be non-strict and call-by-need (also known as deferred execution), which allows for immutable data structures and non-strict, lazy evaluation.

Advantages

You could say that the profound enlightenment you experience when you finally "get it" will make learning functional programming worth it. An experience such as this will make you a better programmer for the rest of your life, whether you actually become a full-time functional programmer or not.

But we're not talking about learning to meditate; we're talking about learning an extremely useful tool that will make you a better programmer.

Formally speaking, what exactly are the practical advantages of using functional programming?

Cleaner code

Functional programs are cleaner, simpler, and smaller. This simplifies debugging, testing, and maintenance.

For example, let's say we need a function that converts a two-dimensional array into a one-dimensional array. Using only imperative techniques, we could write it the following way:

```
function merge2dArrayIntoOne(arrays) {
  var count = arrays.length;
  var merged = new Array(count);
  var c = 0;
  for (var i = 0; i < count; ++i) {
    for (var j = 0, jlen = arrays[i].length; j < jlen; ++j) {
      merged[c++] = arrays[i][j];
    }
  }
  return merged
}
```

And using functional techniques, it could be written as follows:

```
varmerge2dArrayIntoOne2 = function(arrays) {
  return arrays.reduce( function(p,n){
    return p.concat(n);
  });
};
```

Both of these functions take the same input and return the same output. However, the functional example is much more concise and clean.

Modularity

Functional programming forces large problems to be broken down into smaller instances of the same problem to be solved. This means that the code is more modular. Programs that are modular are clearly specified, easier to debug, and simpler to maintain. Testing is easier because each piece of modular code can potentially be checked for correctness.

Reusability

Functional programs share a variety of common helper functions, due to the modularity of functional programming. You'll find that many of these functions can be reused for a variety of different applications.

Many of the most common functions will be covered later in this chapter. However, as you work as a functional programmer, you will inevitably compile your own library of little functions that can be used over and over again. For example, a well-designed function that searches through the lines of a configuration file could also be used to search through a hash table.

Reduced coupling

Coupling is the amount of dependency between modules in a program. Because the functional programmer works to write first-class, higher-order, pure functions that are completely independent of each other with no side effects on global variables, coupling is greatly reduced. Certainly, functions will unavoidably rely on each other. But modifying one function will not change another, so long as the one-to-one mapping of inputs to outputs remains correct.

Mathematically correct

This last one is on a more theoretical level. Thanks to its roots in Lambda calculus, functional programs can be mathematically proven to be correct. This is a big advantage for researchers who need to prove the growth rate, time complexity, and mathematical correctness of a program.

Let's look at Fibonacci's sequence. Although it's rarely used for anything other than a proof-of-concept, it illustrates this concept quite well. The standard way of evaluating a Fibonacci sequence is to create a recursive function that expresses `fibonnaci(n) = fibonnaci(n-2) + fibonnaci(n-1)` with a base case to `return 1 when n < 2`, which makes it possible to stop the recursion and begin adding up the values returned at each step in the recursive call stack.

This describes the intermediary steps involved in calculating the sequence.

```
var fibonacci = function(n) {
  if (n < 2) {
    return 1;
  }
  else {
    return fibonacci(n - 2) + fibonacci(n - 1);
  }
}
console.log( fibonacci(8) );
// Output: 34
```

However, with the help of a library that implements a lazy execution strategy, an indefinite sequence can be generated that states the *mathematical equation* that defines the entire sequence of numbers. Only as many numbers as needed will be computed.

```
var fibonacci2 = Lazy.generate(function() {
  var x = 1,
  y = 1;
  return function() {
    var prev = x;
    x = y;
    y += prev;
    return prev;
  };
}());

console.log(fibonacci2.length());// Output: undefined

console.log(fibonacci2.take(12).toArray());// Output: [1, 1, 2, 3, 5,
8, 13, 21, 34, 55, 89, 144]

var fibonacci3 = Lazy.generate(function() {
  var x = 1,
  y = 1;
  return function() {
    var prev = x;
    x = y;
    y += prev;
    return prev;
  };
}());

console.log(fibonacci3.take(9).reverse().first(1).toArray());//
Output: [34]
```

The second example is clearly more mathematically sound. It relies on the `Lazy.js` library of JavaScript. There are other libraries that can help here as well, such as `Sloth.js` and `wu.js`. These will be covered in *Chapter 3*, *Setting Up the Functional Programming Environment*.

Functional programming in a nonfunctional world

Can functional and nonfunctional programming be mixed together? Although this is the subject of *Chapter 7*, *Functional & Object-oriented Programming in JavaScript*, it is important to get a few things straight before we go any further.

This book is not intended to teach you how to implement an entire application that strictly adheres to the rigors of pure functional programming. Such applications are rarely appropriate outside Academia. Rather, this book will teach you how to use functional programming design strategies within your applications to complement the necessary imperative code.

For example, if you need the first four words that only contain letters out of some text, they could naively be written like this:

```
var words = [], count = 0;
text = myString.split(' ');
for (i=0; count<4, i<text.length; i++) {
  if (!text[i].match(/[0-9]/)) {
    words = words.concat(text[i]);
    count++;
  }
}
console.log(words);
```

In contrast, a functional programmer might write them as follows:

```
var words = [];
var words = myString.split(' ').filter(function(x){
  return (! x.match(/[1-9]+/));
}).slice(0,4);
console.log(words);
```

Or, with a library of functional programming utilities, they can be simplified even further:

```
var words = toSequence(myString).match(/[a-zA-Z]+/).first(4);
```

The key to identifying functions that can be written in a more functional way is to look for loops and temporary variables, such as `words` and `count` instances in the preceding example. We can usually do away with both temporary variables and loops by replacing them with higher-order functions, which we will explore later in this chapter.

Is JavaScript a functional programming language?

There is one last question we must ask ourselves. Is JavaScript a functional language or a non-functional language?

JavaScript is arguably the world's most popular and least understood functional programming language. JavaScript is a functional programming language in C-like clothing. Its syntax is undeniably C-like, meaning it uses C's block syntax and in-fix ordering. And it's one of the worst named languages in existence. It doesn't take a lot of imagination to see how so many people can confuse JavaScript as being related to Java; somehow, its name implies that it should be! But in reality it has very little in common with Java. And, to really cement the idea that JavaScript is an object-oriented language, libraries and frameworks such as Dojo and **ease.js** have been hard at work attempting to abstract it and make it suitable for object-oriented programming. JavaScript came of age in the 1990s when OOP was all the buzz, and we've been told that JavaScript is object-oriented because we want it to be so badly. But it is not.

Its true identity is much more aligned with its ancestors: Scheme and Lisp, two classic functional languages. JavaScript is a functional language, all the way. Its functions are first-class and can be nested, it has closures and compositions, and it allows for currying and monads. All of these are key to functional programming. Here are a few more reasons why JavaScript is a functional language:

- JavaScript's lexical grammar includes the ability to pass functions as arguments, has an inferred type system, and allows for anonymous functions, higher-order functions, closures and more. These facts are paramount to achieving the structure and behavior of functional programming.

- It is not a pure object-oriented language, with most object-oriented design patterns achieved by copying the Prototype object, a weak model for object-oriented programming. **European Computer Manufacturers Association Script** (**ECMAScript**), JavaScript's formal and standardized specifications for implementation, states the following in specification 4.2.1:

  > *"ECMAScript does not contain proper classes such as those in C++, Smalltalk, or Java, but rather, supports constructors which create objects. In a class-based object-oriented language, in general, state is carried by instances, methods are carried by classes, and inheritance is only of structure and behavior. In ECMAScript, the state and methods are carried by objects, and structure, behavior and state are all inherited."*

- It is an interpreted language. Sometimes called "engines", JavaScript interpreters often closely resemble Scheme interpreters. Both are dynamic, both have flexible datatypes that easily combine and transform, both evaluate the code into blocks of expressions, and both treat functions similarly.

That being said, it is true that JavaScript is not a pure functional language. What's lacking is lazy evaluation and built-in immutable data. This is because most interpreters are call-by-name and not call-by-need. JavaScript also isn't very good with recursion due to the way it handles tail calls. However, all of these issues can be mitigated with a little bit of attention. Non-strict evaluation, required for infinite sequences and lazy evaluation, can be achieved with a library called `Lazy.js`. Immutable data can be achieved simply by programming technique, but this requires more programmer discipline rather than relying on the language to take care of it. And recursive tail call elimination can be achieved with a method called **Trampolining**. These issues will be addressed in *Chapter 6*, *Advanced Topics & Pitfalls in JavaScript*.

Many debates have been waged over whether or not JavaScript is a functional language, an object-oriented language, both, or neither. And this won't be the last debate.

In the end, functional programming is way of writing cleaner code through clever ways of mutating, combining, and using functions. And JavaScript provides an excellent medium for this approach. If you really want to use JavaScript to its full potential, you must learn how to use it as a functional language.

Working with functions

Sometimes, the elegant implementation is a function. Not a method. Not a class. Not a framework. Just a function.

-John Carmack, lead programmer of the Doom video game

Functional programming is all about decomposing a problem into a set of functions. Often, functions are chained together, nested within each other, passed around, and treated as first-class citizens. If you've used frameworks such as jQuery and Node.js, you've probably used some of these techniques, you just didn't realize it!

Let's start with a little JavaScript dilemma.

Say we need to compile a list of values that are assigned to generic objects. The objects could be anything: dates, HTML objects, and so on.

```
var
  obj1 = {value: 1},
  obj2 = {value: 2},
  obj3 = {value: 3};

var values = [];
function accumulate(obj) {
  values.push(obj.value);
}
accumulate(obj1);
accumulate(obj2);
console.log(values); // Output: [obj1.value, obj2.value]
```

It works but it's volatile. Any code can modify the `values` object without calling the `accumulate()` function. And if we forget to assign the empty set, `[]`, to the `values` instance then the code will not work at all.

But if the variable is declared inside the function, it can't be mutated by any rogue lines of code.

```
function accumulate2(obj) {
  var values = [];
  values.push(obj.value);
  return values;
}
console.log(accumulate2(obj1)); // Returns: [obj1.value]
console.log(accumulate2(obj2)); // Returns: [obj2.value]
console.log(accumulate2(obj3)); // Returns: [obj3.value]
```

It does not work! Only the value of the object last passed in is returned.

We could possibly solve this with a nested function inside the first function.

```
var ValueAccumulator = function(obj) {
  var values = []
  var accumulate = function() {
    values.push(obj.value);
  };
  accumulate();
  return values;
};
```

But it's the same issue, and now we cannot reach the `accumulate` function or the `values` variable.

What we need is a self-invoking function.

Self-invoking functions and closures

What if we could return a function expression that in-turn returns the `values` array? Variables declared in a function are available to any code within the function, including self-invoking functions.

By using a self-invoking function, our dilemma is solved.

```
var ValueAccumulator = function() {
  var values = [];
  var accumulate = function(obj) {
    if (obj) {
      values.push(obj.value);
      return values;
    }
    else {
      return values;
    }
  };
  return accumulate;
};

//This allows us to do this:
var accumulator = ValueAccumulator();
accumulator(obj1);
accumulator(obj2);
console.log(accumulator());
// Output: [obj1.value, obj2.value]
```

It's all about variable scoping. The `values` variable is available to the inner `accumulate()` function, even when code outside the scope calls the functions. This is called a closure.



Closures in JavaScript are functions that have access to the parent scope, even when the parent function has closed.

Closures are a feature of all functional languages. Traditional imperative languages do not allow them.

Higher-order functions

Self-invoking functions are actually a form of higher-order functions. Higher-order functions are functions that either take another function as the input or return a function as the output.

Higher-order functions are not common in traditional programming. While an imperative programmer might use a loop to iterate an array, a functional programmer would take another approach entirely. By using a higher-order function, the array can be worked on by applying that function to each item in the array to create a new array.

This is the central idea of the functional programming paradigm. What higher-order functions allow is the ability to pass logic to other functions, just like objects.

Functions are treated as first-class citizens in JavaScript, a distinction JavaScript shares with Scheme, Haskell, and the other classic functional languages. This may sound bizarre, but all this really means is that functions are treated as primitives, just like numbers and objects. If numbers and objects can be passed around, so can functions.

To see this in action, let's use a higher-order function with our `ValueAccumulator()` function from the previous section:

```
// using forEach() to iterate through an array and call a
// callback function, accumulator, for each item
var accumulator2 = ValueAccumulator();
var objects = [obj1, obj2, obj3]; // could be huge array of objects
objects.forEach(accumulator2);
console.log(accumulator2());
```

Pure functions

Pure functions return a value computed using only the inputs passed to it. Outside variables and global states may not be used and there may be no side effects. In other words, it must not mutate the variables passed to it for input. Therefore, pure functions are only used for their returned value.

A simple example of this is a math function. The `Math.sqrt(4)` function will always return `2`, does not use any hidden information such as settings or state, and will never inflict any side effects.

Pure functions are the true interpretation of the mathematical term for 'function', a relation between inputs and an output. They are simple to think about and are readily re-usable. Because they are totally independent, pure functions are more capable of being used again and again.

To illustrate this, compare the following non-pure function to the pure one.

```
// function that prints a message to the center of the screen
var printCenter = function(str) {
  var elem = document.createElement("div");
  elem.textContent = str;
  elem.style.position = 'absolute';
  elem.style.top = window.innerHeight/2+"px";
  elem.style.left = window.innerWidth/2+"px";
  document.body.appendChild(elem);
};
printCenter('hello world');
// pure function that accomplishes the same thing
var printSomewhere = function(str, height, width) {
  var elem = document.createElement("div");
  elem.textContent = str;
  elem.style.position = 'absolute';
  elem.style.top = height;
  elem.style.left = width;
  return elem;
};
document.body.appendChild(
  printSomewhere('hello world',
  window.innerHeight/2)+10+"px",
  window.innerWidth/2)+10+"px")
);
```

While the non-pure function relies on the state of the window object to compute the height and width, the pure, self-sufficient function instead asks that those values be passed in. What this actually does is allow the message to be printed anywhere, and this makes the function much more versatile.

And while the non-pure function may seem like the easier option because it performs the appending itself instead of returning an element, the pure function `printSomewhere()` and its returned value play better with other functional programming design techniques.

```
var messages = ['Hi', 'Hello', 'Sup', 'Hey', 'Hola'];
messages.map(function(s,i){
  return printSomewhere(s, 100*i*10, 100*i*10);
}).forEach(function(element) {
  document.body.appendChild(element);
});
```



When the functions are pure and don't rely on state or environment, then we don't care about when or where they actually get computed. We'll see this later with lazy evaluation.

Anonymous functions

Another benefit of treating functions as first-class objects is the advent of anonymous functions.

As the name might imply, anonymous functions are functions without names. But they are more than that. What they allow is the ability to define ad-hoc logic, on-the-spot and as needed. Usually, it's for the benefit of convenience; if the function is only referred to once, then a variable name doesn't need to be wasted on it.

Some examples of anonymous functions are as follows:

```
// The standard way to write anonymous functions
function(){return "hello world"};

// Anonymous function assigned to variable
var anon = function(x,y){return x+y};

// Anonymous function used in place of a named callback function,
// this is one of the more common uses of anonymous functions.
setInterval(function(){console.log(new Date().getTime())}, 1000);
```

```
// Output:  1413249010672, 1413249010673, 1413249010674, ...

// Without wrapping it in an anonymous function, it immediately
// execute once and then return undefined as the callback:
setInterval(console.log(new Date().getTime()), 1000)
// Output:  1413249010671
```

A more involved example of anonymous functions used within higher-order functions:

```
function powersOf(x) {
  return function(y) {
    // this is an anonymous function!
    return Math.pow(x,y);
  };
}
powerOfTwo = powersOf(2);
console.log(powerOfTwo(1)); // 2
console.log(powerOfTwo(2)); // 4
console.log(powerOfTwo(3)); // 8

powerOfThree = powersOf(3);
console.log(powerOfThree(3));  // 9
console.log(powerOfThree(10)); // 59049
```

The function that is returned doesn't need to be named; it can't be used anywhere outside the `powersOf()` function, and so it is an anonymous function.

Remember our accumulator function? It can be re-written using anonymous functions.

```
var
  obj1 = {value: 1},
  obj2 = {value: 2},
  obj3 = {value: 3};

var values = (function() {
  // anonymous function
  var values = [];
  return function(obj) {
    // another anonymous function!
    if (obj) {
      values.push(obj.value);
      return values;
    }
    else {
```

```
        return values;
      }
    }
})(); // make it self-executing
console.log(values(obj1)); // Returns: [obj.value]
console.log(values(obj2)); // Returns: [obj.value, obj2.value]
```

Right on! A pure, high-order, anonymous function. How did we ever get so lucky? Actually, it's more than that. It's also *self-executing* as indicated by the structure, `(function(){...})();`. The pair of parentheses following the anonymous function causes the function to be called right away. In the above example, the `values` instance is assigned to the output of the self-executing function call.



Anonymous functions are more than just syntactical sugar. They are the embodiment of Lambda calculus. Stay with me on this… Lambda calculus was invented long before computers or computer languages. It was just a mathematical notion for reasoning about functions. Remarkably, it was discovered that—despite the fact that it only defines three kinds of expressions: variable references, function calls, and *anonymous functions*—it was Turing-complete. Today, Lambda calculus lies at the core of all functional languages if you know how to find it, including JavaScript.

For this reason, anonymous functions are often called lambda expressions.

One drawback to anonymous functions remains. They're difficult to identify in call stacks, which makes debugging trickier. They should be used sparingly.

Method chains

Chaining methods together in JavaScript is quit common. If you've used jQuery, you've likely performed this technique. It's sometimes called the "Builder Pattern".

It's a technique that is used to simplify code where multiple functions are applied to an object one after another.

```
// Instead of applying the functions one per line...
arr = [1,2,3,4];
arr1 = arr.reverse();
arr2 = arr1.concat([5,6]);
arr3 = arr2.map(Math.sqrt);
```

```
// ...they can be chained together into a one-liner
console.log([1,2,3,4].reverse().concat([5,6]).map(Math.sqrt));
// parentheses may be used to illustrate
console.log((((1,2,3,4]).reverse()).concat([5,6])).map(Math.sqrt)
);
```

This only works when the functions are methods of the object being worked on. If you created your own function that, for example, takes two arrays and returns an array with the two arrays zipped together, you must declare it as a member of the `Array.prototype` object. Take a look at the following code snippet:

```
Array.prototype.zip = function(arr2) {
  // ...
}
```

This would allow us to the following:

```
arr.zip([11,12,13,14).map(function(n){return n*2});
// Output: 2, 22, 4, 24, 6, 26, 8, 28
```

Recursion

Recursion is likely the most famous functional programming technique. If you don't know by now, a recursive function is a function that calls itself.

When a functions calls *itself*, something strange happens. It acts both as a loop, in that it executes the same code multiple times, and as a function stack.

Recursive functions must be very careful to avoid an infinite loop (rather, infinite recursion in this case). So just like loops, a condition must be used to know when to stop. This is called the base case.

An example is as follows:

```
var foo = function(n) {
  if (n < 0) {
    // base case
    return 'hello';
  }
  else {
    // recursive case
    foo(n-1);
  }
}
console.log(foo(5));
```

It's possible to convert any loop to a recursive algorithm and any recursive algorithm to a loop. But recursive algorithms are more appropriate, almost necessary, for situations that differ greatly from those where loops are appropriate.

A good example is tree traversal. While it's not too hard to traverse a tree using a recursive function, a loop would be much more complex and would need to maintain a stack. And that would go against the spirit of functional programming.

```
var getLeafs = function(node) {
  if (node.childNodes.length == 0) {
    // base case
    return node.innerText;
  }
  else {
    // recursive case:
    return node.childNodes.map(getLeafs);
  }
}
```

Divide and conquer

Recursion is more than an interesting way to iterate without `for` and `while` loops. An algorithm design, known as divide and conquer, recursively breaks problems down into smaller instances of the same problem until they're small enough to solve.

The historical example of this is the Euclidan algorithm for finding the greatest common denominator for two numbers.

```
function gcd(a, b) {
  if (b == 0) {
    // base case (conquer)
    return a;
  }
  else {
    // recursive case (divide)
    return gcd(b, a % b);
  }
}

console.log(gcd(12,8));
console.log(gcd(100,20));
```

So in theory, divide and conquer works quite eloquently, but does it have any use in the real world? Yes! The JavaScript function for sorting arrays is not very good. Not only does it sort the array in place, which means that the data is not immutable, but it is unreliable and inflexible. With divide and conquer, we can do better.

The merge sort algorithm uses the divide and conquer recursive algorithm design to efficiently sort an array by recursively dividing the array into smaller sub-arrays and then merging them together.

The full implementation in JavaScript is about 40 lines of code. However, pseudo-code is as follows:

```
var mergeSort = function(arr){
  if (arr.length < 2) {
    // base case: 0 or 1 item arrays don't need sorting
    return items;
  }
  else {
    // recursive case: divide the array, sort, then merge
    var middle = Math.floor(arr.length / 2);
    // divide
    var left = mergeSort(arr.slice(0, middle));
    var right = mergeSort(arr.slice(middle));
    // conquer
    // merge is a helper function that returns a new array
    // of the two arrays merged together
    return merge(left, right);
  }
}
```

Lazy evaluation

Lazy evaluation, also known as non-strict evaluation, call-by-need and deffered execution, is an evaluation strategy that waits until the value is needed to compute the result of a function and is particularly useful for functional programming. It's clear that a line of code that states `x = func()` is calling for `x` to be assigned to the returned value by `func()`. But what `x` actually equates to does not matter until it is needed. Waiting to call `func()` until `x` is needed is known as lazy evaluation.

This strategy can result in a major increase in performance, especially when used with method chains and arrays, the favorite program flow techniques of the functional programmer.

One exciting benefit of lazy evaluation is the existence of infinite series. Because nothing is actually computed until it can't be delayed any further, it's possible to do this:

```
// wishful JavaScript pseudocode:
var infinateNums = range(1 to infinity);
var tenPrimes = infinateNums.getPrimeNumbers().first(10);
```

This opens the door for many possibilities: asynchronous execution, parallelization, and composition, just to name a few.

However, there's one problem: JavaScript does not perform Lazy evaluation on its own. That being said, there exist libraries for JavaScript that simulate lazy evaluation very well. That is the subject of *Chapter 3*, *Setting Up the Functional Programming Environment*.

The functional programmer's toolkit

If you've looked closely at the few examples presented so far, you'll notice a few methods being used that you may not be familiar with. They are the `map()`, `filter()`, and `reduce()` functions, and they are crucial to every functional program of any language. They enable you to remove loops and statements, resulting in cleaner code.

`The map()`, `filter()`, and `reduce()` functions make up the core of the functional programmer's toolkit, a collection of pure, higher-order functions that are the workhorses of the functional method. In fact, they're the epitome of what a pure function and what a higher-order function should be like; they take a function as input and return an output with zero side effects.

While they're standard for browsers that implement ECMAScript 5.1, they only work on arrays. Each time it's called, a new array is created and returned. The existing array is not modified. But there's more, *they take functions as inputs*, often in the form of anonymous functions referred to as callback functions; they iterate over the array and apply the function to each item in the array!

```
myArray = [1,2,3,4];
newArray = myArray.map(function(x) {return x*2});
console.log(myArray);  // Output: [1,2,3,4]
console.log(newArray); // Output: [2,4,6,8]
```

One more thing. Because they only work on arrays, they do not work on other iterable data structures, like certain objects. Fret not, libraries such as `underscore.js`, `Lazy.js`, `stream.js`, and many more all implement their own `map()`, `filter()`, and `reduce()` methods that are more versatile.

Callbacks

If you've never worked with callbacks before, you might find the concept a little puzzling. This is especially true in JavaScript, given the several different ways that JavaScript allows you to declare functions.

A `callback()` function is used for passing to other functions for them to use. It's a way to pass logic just as you would pass an object:

```
var myArray = [1,2,3];
function myCallback(x){return x+1};
console.log(myArray.map(myCallback));
```

To make it simpler for easy tasks, anonymous functions can be used:

```
console.log(myArray.map(function(x){return x+1}));
```

They are not only used in functional programming, they are used for many things in JavaScript. Purely for example, here's a `callback()` function used in an AJAX call made with jQuery:

```
function myCallback(xhr){
  console.log(xhr.status);
  return true;
}
$.ajax(myURI).done(myCallback);
```

Notice that only the name of the function was used. And because we're not calling the callback and are only passing the name of it, it would be wrong to write this:

```
$.ajax(myURI).fail(myCallback(xhr));
// or
$.ajax(myURI).fail(myCallback());
```

What would happen if we did call the callback? In that case, the `myCallback(xhr)` method would try to execute—'undefined' would be printed to the console and it would return `True`. When the `ajax()` call completes, it will have 'true' as the name of the callback function to use, and that will throw an error.

What this also means is that we cannot specify what arguments are passed to the callback functions. If we need different parameters from what the `ajax()` call will pass to it, we can wrap the callback function in an anonymous function:

```
function myCallback(status){
  console.log(status);
  return true;
}
$.ajax(myURI).done(function(xhr){myCallback(xhr.status)});
```

Array.prototype.map()

The `map()` function is the ringleader of the bunch. It simply applies the callback function on each item in the array.



Syntax: `arr.map(callback [, thisArg]);`

Parameters:

- `callback()`: This function produces an element for the new array, receiving these arguments:
  - `currentValue`: This argument gives the current element being processed in the array
  - `index`: This argument gives the index of the current element in the array
  - `array`: This argument gives the array being processed
- `thisArg()`: This function is optional. The value is used as `this` when executing `callback`.

Examples:

```
var
  integers = [1,-0,9,-8,3],
  numbers = [1,2,3,4],
  str = 'hello world how ya doing?';
// map integers to their absolute values
console.log(integers.map(Math.abs));

// multiply an array of numbers by their position in the array
```

```
console.log(numbers.map(function(x, i){return x*i}) );

// Capitalize every other word in a string.
console.log(str.split(' ').map(function(s, i){
  if (i%2 == 0) {
    return s.toUpperCase();
  }
  else {
    return s;
  }
}) );
```



While the `Array.prototype.map` method is a standard method for the Array object in JavaScript, it can be easily extended to your custom objects as well.

```
MyObject.prototype.map = function(f) {
  return new MyObject(f(this.value));
};
```

Array.prototype.filter()

The `filter()` function is used to take elements out of an array. The callback must return `True` (to include the item in the new array) or `False` (to drop it). Something similar could be achieved by using the `map()` function and returning a `null` value for items you want dropped, but the `filter()` function will delete the item from the new array instead of inserting a `null` value in its place.



Syntax: `arr.filter(callback [, thisArg]);`

Parameters:

- `callback()`: This function is used to test each element in the array. Return `True` to keep the element, `False` otherwise. With these parameters:
  - `currentValue`: This parameter gives the current element being processed in the array
  - `index`: This parameter gives the index of the current element in the array

- `array`: This parameter gives the array being processed.
- `thisArg()`: This function is optional. Value is used as `this` when executing `callback`.

Examples:

```
var myarray = [1,2,3,4]
words = 'hello 123 world how 345 ya doing'.split(' ');
re = '[a-zA-Z]';
// remove all negative numbers
console.log([-2,-1,0,1,2].filter(function(x){return x>0}));
// remove null values after a map operation
console.log(words.filter(function(s){
  return s.match(re);
}) );
// remove random objects from an array
console.log(myarray.filter(function(){
  return Math.floor(Math.random()*2)})
);
```

Array.prototype.reduce()

Sometimes called fold, the `reduce()` function is used to accumulate all the values of the array into one. The callback needs to return the logic to be performed to combine the objects. In the case of numbers, they're usually added together to get a sum or multiplied together to get a product. In the case of strings, the strings are often appended together.



Syntax: `arr.reduce(callback [, initialValue]);`

Parameters:

- `callback()`: This function combines two objects into one, which is returned. With these parameters:
  - `previousValue`: This parameter gives the value previously returned from the last invocation of the callback, or the `initialValue`, if supplied
  - `currentValue`: This parameter gives the current element being processed in the array

    - `index`: This parameter gives the index of the current element in the array
    - `array`: This parameter gives the array being processed

- `initialValue()`: This function is optional. Object to use as the first argument to the first call of the `callback`.

Examples:

```
var numbers = [1,2,3,4];
// sum up all the values of an array
console.log([1,2,3,4,5].reduce(function(x,y){return x+y}, 0));
// sum up all the values of an array
console.log([1,2,3,4,5].reduce(function(x,y){return x+y}, 0));

// find the largest number
console.log(numbers.reduce(function(a,b){
  return Math.max(a,b)}) // max takes two arguments
);
```

Honorable mentions

The `map()`, `filter()`, and `reduce()` functions are not alone in our toolbox of helper functions. There exist many more functions that can be plugged into nearly any functional application.

Array.prototype.forEach

Essentially the non-pure version of `map()`, `forEach()` iterates over an array and applies a `callback()` function over each item. However, it doesn't return anything. It's a cleaner way of performing a `for` loop.



Syntax: `arr.forEach(callback [, thisArg]);`

Parameters:

- `callback()`: This function is to be performed for each value of the array. With these parameters:
    - `currentValue`: This parameter gives the current element being processed in the array
    - `index`: This parameter gives the index of the current element in the array
    - `array`: This parameter gives the array being processed
- `thisArg`: This function is optional. Value is used as `this` when executing `callback`.

Examples:

```
var arr = [1,2,3];
var nodes = arr.map(function(x) {
  var elem = document.createElement("div");
  elem.textContent = x;
  return elem;
});

// log the value of each item
arr.forEach(function(x){console.log(x)});

// append nodes to the DOM
nodes.forEach(function(x){document.body.appendChild(x)});
```

Array.prototype.concat

When working with arrays instead of `for` and `while` loops, often you will need to join multiple arrays together. Another built-in JavaScript function, `concat()`, takes care of this for us. The `concat()` function returns a new array and leaves the old arrays untouched. It can join as many arrays as you pass to it.

```
console.log([1, 2, 3].concat(['a','b','c']) // concatenate two
arrays);
// Output: [1, 2, 3, 'a','b','c']
```

The original array is untouched. It returns a new array with both arrays concatenated together. This also means that the `concat()` function can be chained together.

```
var arr1 = [1,2,3];
var arr2 = [4,5,6];
var arr3 = [7,8,9];
var x = arr1.concat(arr2, arr3);
var y = arr1.concat(arr2).concat(arr3));
var z = arr1.concat(arr2.concat(arr3)));
console.log(x);
console.log(y);
console.log(z);
```

Variables `x`, `y` and `z` all contain `[1,2,3,4,5,6,7,8,9]`.

Array.prototype.reverse

Another native JavaScript function helps with array transformations. The `reverse()` function inverts an array, such that the first element is now the last and the last is now the first.

However, it does not return a new array; instead it mutates the array in place. We can do better. Here's an implementation of a pure method for reversing an array:

```
var invert = function(arr) {
  return arr.map(function(x, i, a) {
    return a[a.length - (i+1)];
  });
};
var q = invert([1,2,3,4]);
console.log( q );
```

Array.prototype.sort

Much like our `map()`, `filter()`, and `reduce()` methods, the `sort()` method takes a `callback()` function that defines how the objects within an array should be sorted. But, like the `reverse()` function, it mutates the array in place. And that's no bueno.

```
arr = [200, 12, 56, 7, 344];
console.log(arr.sort(function(a,b){return a-b}) );
// arr is now: [7, 12, 56, 200, 344];
```

We could write a pure `sort()` function that doesn't mutate the array, but sorting algorithms is the source of much grief. Significantly large arrays that need to be sorted really should be organized in data structures that are designed just for that: quickStort, mergeSort, bubbleSort, and so on.

Array.prototype.every and Array.prototype.some

The `Array.prototype.every()` and `Array.prototype.some()` functions are both pure and high-order functions that are methods of the `Array` object and are used to test the elements of an array against a `callback()` function that must return a Boolean representing the respective input. The `every()` function returns `True` if the `callback()` function returns `True` for every element in the array, and the `some()` function returns `True` if some elements in the array are `True`.

Example:

```
function isNumber(n) {
  return !isNaN(parseFloat(n)) && isFinite(n);
}

console.log([1, 2, 3, 4].every(isNumber)); // Return: true
console.log([1, 2, 'a'].every(isNumber)); // Return: false
console.log([1, 2, 'a'].some(isNumber)); // Return: true
```

Summary

In order to develop an understanding of functional programming, this chapter covered a fairly broad range of topics. First we analyzed what it means for a programming language to be functional, then we evaluated JavaScript for its functional programming capabilities. Next, we applied the core concepts of functional programming using JavaScript and showcased some of JavaScript's built-in functions for functional programming.

Although JavaScript does have a few tools for functional programming, its functional core remains mostly hidden and much is to be desired. In the next chapter, we will explore several libraries for JavaScript that expose its functional underbelly.

3

Setting Up the Functional Programming Environment

Introduction

Do we need to know advanced math—category theory, Lambda calculus, polymorphisms—just to write applications with functional programming? Do we need to reinvent the wheel? The short answer to both these questions is *no*.

In this chapter, we will do our best to survey everything that can impact the way we write our functional applications in JavaScript.

- Libraries
- Toolkits
- Development environments
- Functional language that compiles to JavaScript
- And more

Please understand that the current landscape of functional libraries for JavaScript is a very fluid one. Like all aspects of computer programming, the community can change in a heartbeat; new libraries can be adopted and old ones can be abandoned. For instance, during the writing process of this very book, the popular and stable `Node.js` platform for I/O has been forked by its open source community. Its future is vague.

Therefore, the most important concept to be gained from this chapter is not how to use the current libraries for functional programming, but how to use any library that enhances JavaScript's functional programming method. This chapter will not focus on just one or two libraries, but will explore as many as possible with the goal of surveying all the many styles of functional programming that exist within JavaScript.

Functional libraries for JavaScript

It's been said that every functional programmer writes their own library of functions, and functional JavaScript programmers are no exception. With today's open source code-sharing platforms such as GitHub, Bower, and NPM, it's easier to share, collaborate, and grow these libraries. Many libraries exist for functional programming with JavaScript, ranging from tiny toolkits to monolithic module libraries.

Each library promotes its own style of functional programming. From a rigid, math-based style to a relaxed, informal style, each library is different but they all share one common feature: they all have abstract JavaScript functional capabilities to increase code re-use, readability, and robustness.

At the time of writing, however, a single library has not established itself as the de-facto standard. Some might argue that `underscore.js` is the one but, as you'll see in the following section, it might be advisable to avoid `underscore.js`.

Underscore.js

Underscore has become the standard functional JavaScript library in the eyes of many. It is mature, stable, and was created by *Jeremy Ashkenas*, the man behind the `Backbone.js` and `CoffeeScript` libraries. Underscore is actually a reimplementation of Ruby's `Enumerable` module, which explains why CoffeeScript was also influenced by Ruby.

Similar to jQuery, Underscore doesn't modify native JavaScript objects and instead uses a symbol to define its own object: the underscore character "_". So, using Underscore would work like this:

```
var x = _.map([1,2,3], Math.sqrt); // Underscore's map function
console.log(x.toString());
```

We've already seen JavaScrip's native `map()` method for the `Array` object, which works like this:

```
var x = [1,2,3].map(Math.sqrt);
```

The difference is that, in Underscore, both the `Array` object and the `callback()` function are passed as parameters to the Underscore object's `map()` method (`_.map`), as opposed to passing only the callback to the array's native `map()` method (`Array.prototype.map`).

But there's way more than just `map()` and other built-in functions to Underscore. It's full of super handy functions such as `find()`, `invoke()`, `pluck()`, `sortyBy()`, `groupBy()`, and more.

```
var greetings = [{origin: 'spanish', value: 'hola'},
{origin: 'english', value: 'hello'}];
console.log(_.pluck(greetings, 'value')  );
// Grabs an object's property.
// Returns: ['hola', 'hello']
console.log(_.find(greetings, function(s) {return s.origin ==
'spanish';}));
// Looks for the first obj that passes the truth test
// Returns: {origin: 'spanish', value: 'hola'}
greetings = greetings.concat(_.object(['origin','value'],
['french','bonjour']));
console.log(greetings);
// _.object creates an object literal from two merged arrays
// Returns: [{origin: 'spanish', value: 'hola'},
//{origin: 'english', value: 'hello'},
//{origin: 'french', value: 'bonjour'}]
```

And it provides a way of chaining methods together:

```
var g = _.chain(greetings)
  .sortBy(function(x) {return x.value.length})
  .pluck('origin')
  .map(function(x){return x.charAt(0).toUpperCase()+x.slice(1)})
  .reduce(function(x, y){return x + ' ' + y}, '')
  .value();
// Applies the functions
// Returns: 'Spanish English French'
console.log(g);
```



The `_.chain()` method returns a wrapped object that holds all the Underscore functions. The `_.value` method is then used to extract the value of the wrapped object. Wrapped objects are also very useful for mixing Underscore with object-oriented programming.

Despite its ease of use and adaptation by the community, the `underscore.js` library has been criticized for forcing you to write overly verbose code and for encouraging the wrong patterns. Underscore's structure may not be ideal or even function!

Until version 1.7.0, released shortly after Brian Lonsdorf's talk entitled *Hey Underscore, you're doing it wrong!*, landed on YouTube, Underscore explicitly prevented us from extending functions such as `map()`, `reduce()`, `filter()`, and more.

```
_.prototype.map = function(obj, iterate, [context]) {
  if (Array.prototype.map && obj.map === Array.prototype.map)
  return obj.map(iterate, context);
  // ...
};
```



You can watch the video of Brian Lonsdorf's talk at `www.youtube.com/watch?v=m3svKOdZij`.

Map, in terms of category theory, is a homomorphic functor interface (more on this in *Chapter 5*, *Category Theory*). And we should be able to define `map` as a functor for whatever we need it for. So that's not very functional of Underscore.

And because JavaScript doesn't have built-in immutable data, a functional library should be careful to not allow its helper functions to mutate the objects passed to it. A good example of this problem is shown below. The intention of the snippet is to return a new `selected` list with one option set as the default. But what actually happens is that the `selected` list is mutated in place.

```
function getSelectedOptions(id, value) {
  options = document.querySelectorAll('#' + id + ' option');
  var newOptions = _.map(options, function(opt){
    if (opt.text == value) {
      opt.selected = true;
      opt.text += ' (this is the default)';
    }
    else {
      opt.selected = false;
    }
    return opt;
  });
  return newOptions;
}
var optionsHelp = getSelectedOptions('timezones', 'Chicago');
```

We would have to insert the line `opt = opt.cloneNode();` to the `callback()` function to make a copy of each object within the list being passed to the function. Underscore's `map()` function cheats to boost performance, but it is at the cost of functional *feng shui*. The native `Array.prototype.map()` function wouldn't require this because it makes a copy, but it also doesn't work on `nodelist` collections.

Underscore may be less than ideal for mathematically-correct, functional programming, but it was never intended to extend or transform JavaScript into a pure functional language. It defines itself as *a JavaScript library that provides a whole mess of useful functional programming helpers*. It may be a little more than a spurious collection of functional-like helpers, but it's no serious functional library either.

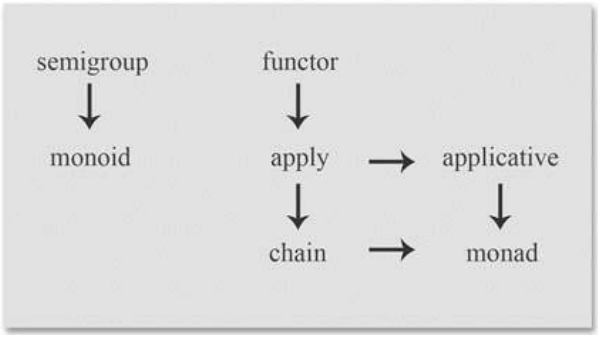
Is there a better library out there? Perhaps one that is based on mathematics?

Fantasy Land

Sometimes, the truth is stranger than fiction.

Fantasy Land is a collection of functional base libraries and a formal specification for how to implement "algebraic structures" in JavaScript. More specifically, Fantasy Land specifies the interoperability of common algebraic structures, or algebras for short: monads, monoids, setoids, functors, chains, and more. Their names may sound scary, but they're just a set of values, a set of operators, and some laws it must obey. In other words, they're just objects.

Here's how it works. Each algebra is a separate Fantasy Land specification and may have dependencies on other algebras that need to be implemented.



Some of the algebra specifications are:

- Setoids:
    - Implement the reflexivity, symmetry and transitivity laws
    - Define the `equals()` method
- Semigroups
    - Implement the associativity law
    - Define the `concat()` method
- Monoid
    - Implement right identity and left identity
    - Define the `empty()` method
- Functor
    - Implement the identity and composition laws
    - Define the `map()` method

The list goes on and on.

We don't necessarily need to know exactly what each algebra is for but it certainly helps, especially if you're writing your own library that conforms to the specifications. It's not just abstract nonsense, it outlines a means of implementing a high-level abstraction called category theory. A full explanation of category theory can be found in *Chapter 5*, *Category Theory*.

Fantasy Land doesn't just tell us how to implement functional programming, it does provide a set of functional modules for JavaScript. However, many are incomplete and documentation is pretty sparse. But Fantasy Land isn't the only library out there to implement its open source specifications. Others have too, namely: **Bilby.js**.

Bilby.js

What the heck is a bilby? No, it's not a mythical creature that might exist in Fantasy Land. It exists here on Earth as a freaky/cute cross between a mouse and a rabbit. Nonetheless, `bibly.js` library is compliant with Fantasy Land specifications.

In fact, `bilby.js` is a serious functional library. As its documentation states, it is, *Serious, meaning it applies category theory to enable highly abstract code. Functional, meaning it enables referentially transparent programs*. Wow, that is pretty serious. The documentation located at `http://bilby.brianmckenna.org/` goes on to say that it provides:

- Immutable multi-methods for ad-hoc polymorphism
- Functional data structures
- Operator overloading for functional syntax
- Automated specification testing (**ScalaCheck**, **QuickCheck**)

By far the most mature library that conforms to the Fantasy Land specifications for algebraic structures, `Bilby.js` is a great resource for fully committing to the functional style.

Let's try an example:

```
// environments in bilby are immutable structure for multimethods
var shapes1 = bilby.environment()
  // can define methods
  .method(
    'area', // methods take a name
    function(a){return typeof(a) == 'rect'}, // a predicate
    function(a){return a.x * a.y} // and an implementation
  )
  // and properties, like methods with predicates that always
  // return true
  .property(
     'name',   // takes a name
     'shape'); // and a function
// now we can overload it
var shapes2 = shapes1
  .method(
    'area', function(a){return typeof(a) == 'circle'},
    function(a){return a.r * a.r * Math.PI} );
var shapes3 = shapes2
  .method(
    'area', function(a){return typeof(a) == 'triangle'},
    function(a){return a.height * a.base / 2} );

// and now we can do something like this
var objs = [{type:'circle', r:5}, {type:'rect', x:2, y:3}];
var areas = objs.map(shapes3.area);

// and this
var totalArea = objs.map(shapes3.area).reduce(add);
```

This is category theory and ad-hoc polymorphism in action. Again, category theory will be covered in full in *Chapter 5*, *Category Theory*.



Category theory is a recently invigorated branch of mathematics that functional programmers use to maximize the abstraction and usefulness of their code. *But there is a major drawback: it's difficult to conceptualize and quickly get started with.*

The truth is that Bilby and Fantasy Land are really stretching the possibilities of functional programming in JavaScript. Although it's exciting to see the evolution of computer science, the world may just not be ready for the kind of hard-core functional style that Bibly and Fantasy Land are pushing.

Maybe such a grandiose library on the bleeding-edge of functional JavaScript is not our thing. After all, we set out to explore the functional techniques that complement JavaScript, not to build functional programming dogma. Let's turn our attention to another new library, `Lazy.js`.

Lazy.js

Lazy is a utility library more along the lines of the `underscore.js` library but with a lazy evaluation strategy. Because of this, Lazy makes the impossible possible by functionally computing results of series that won't be available with immediate interpretation. It also boasts a significant performance boost.

The `Lazy.js` library is still very young. But it has a lot of momentum and community enthusiasm behind it.

The idea is that, in Lazy, everything is a sequence that we can iterate over. Owing to the way the library controls the order in which methods are applied, many really cool things can be achieved: asynchronous iteration (parallel programming), infinite sequences, functional reactive programming, and more.

The following examples show off a bit of everything:

```
// Get the first eight lines of a song's lyrics
var lyrics = "Lorem ipsum dolor sit amet, consectetur adipiscing eli
// Without Lazy, the entire string is first split into lines
console.log(lyrics.split('\n').slice(0,3));

// With Lazy, the text is only split into the first 8 lines
// The lyrics can even be infinitely long!
console.log(Lazy(lyrics).split('\n').take(3));
```

```
//First 10 squares that are evenly divisible by 3
var oneTo1000 = Lazy.range(1, 1000).toArray();
var sequence = Lazy(oneTo1000)
  .map(function(x) { return x * x; })
  .filter(function(x) { return x % 3 === 0; })
  .take(10)
  .each(function(x) { console.log(x); });

// asynchronous iteration over an infinite sequence
var asyncSequence = Lazy.generate(function(x){return x++})
  .async(100) // 0.100s intervals between elements
  .take(20) // only compute the first 20
  .each(function(e) { // begin iterating over the sequence
    console.log(new Date().getMilliseconds() + ": " + e);
  });
```

More examples and use-cases are covered in *Chapter 4*, *Implementing Functional Programming Techniques in JavaScript*.

But its not entirely correct to fully credit the `Lazy.js` library with this idea. One of its predecessors, the `Bacon.js` library, works in much the same way.

Bacon.js

The logo of `Bacon.js` library is as follows:



The mustachioed hipster of functional programming libraries, `Bacon.js` is itself a library for *functional reactive programming*. Functional reactive programming just means that functional design patterns are used to represent values that are reactive and always changing, like the position of the mouse on the screen, or the price of a company's stock. In the same way that Lazy can get away with creating infinite sequences by not calculating the value until it's needed, Bacon can avoid having to calculate ever-changing values until the very last second.

What are called sequences in Lazy are known as EventStreams and Properties in Bacon because they're more suited for working with events (`onmouseover`, `onkeydown`, and so on) and reactive properties (scroll position, mouse position, toggles, and so on).

```
Bacon.fromEventTarget(document.body, "click")
  .onValue(function() { alert("Bacon!") });
```

Bacon is a little bit older than Lazy but its feature set is about half the size and its community enthusiasm is about equal.

Honorable mentions

There are simply too many libraries out there to do them all justice within the scope of this book. Let's look at a few more libraries for functional programming in JavaScript.

- `Functional`
    - Possibly the first library for functional programming in JavaScript, `Functional` is a library that includes comprehensive higher-order function support as well as `string` lambdas
- `wu.js`
    - Especially prized for its `curryable()` function, `wu.js` library is a very nice Library for functional programming. It was the first library (that I know of) to implement lazy evaluation, getting the ball rolling for `Bacon.js`, `Lazy.js` and other libraries
    - Yes, it is named after the infamous rap group *Wu Tang Clan*
- `sloth.js`
    - Very similar to the `Lazy.js` libraries, but much smaller
- `stream.js`
    - The `stream.js` library supports infinite streams and not much else
    - Absolutely tiny in size
- `Lo-Dash.js`
    - As the name might imply, the `lo-dash.js` library was inspired by the `underscore.js` library
    - Highly optimized

- `Sugar`
  - `Sugar` is a support library for functional programming techniques in JavaScript, like Underscore, but with some key differences in how it's implemented.
  - Instead of doing `_.pluck(myObjs, 'value')` in Underscore, it's just `myObjs.map('value')` in Sugar. This means that it modifies native JavaScript objects, so there is a small risk of it not playing nicely with other libraries that do the same such as Prototype.
  - Very good documentation, unit tests, analyzers, and more.
- `from.js`
  - A new functional library and **LINQ** (**Language Integrated Query**) engine for JavaScript that supports most of the same LINQ functions that .NET provides
  - 100% lazy evaluation and supports lambda expressions
  - Very young but documentation is excellent
- JSLINQ
  - Another functional LINQ engine for JavaScript
  - Much older and more mature than `from.js` library
- `Boiler.js`
  - Another utility library that extends JavaScript's functional methods to more primitives: strings, numbers, objects, collections and arrays
- **Folktale**
  - Like the `Bilby.js` library, Folktale is another new library that implements the Fantasy Land specifications. And like its forefather, Folktale is also a collection of libraries for functional programming in JavaScript. It's very young but could have a bright future.
- **jQuery**
  - Surprised to see jQuery mentioned here? Although jQuery is not a tool used to perform functional programming, it nevertheless is functional itself. jQuery might be one of the most widely used libraries that has its roots in functional programming.
  - The jQuery object is actually a monad. jQuery uses the monadic laws to enable method chaining:

```
$('#mydiv').fadeIn().css('left': 50).alert('hi!');
```

A full explanation of this can be found in *Chapter 7, Functional and Object-oriented Programming in JavaScript*.

- And some of its methods are higher-order:
  ```
  $('li').css('left': function(index){return index*50});
  ```
- As of jQuery 1.8, the `deferred.then` parameter implements a functional concept known as Promises.
- jQuery is an abstraction layer, mainly for the DOM. It's not a framework or a toolkit, just a way to use abstraction to increase code-reuse and reduce ugly code. And isn't that what functional programming is all about?

Development and production environments

It does not matter in terms of programming style what type of environment the application is being developed in and will be deployed in. But it does matter to the libraries a lot.

Browsers

The majority of JavaScript applications are designed to run on the client side, that is, in the client's browser. Browser-based environments are excellent for development because browsers are ubiquitous, you can work on the code right on your local machine, the interpreter is the browser's JavaScript engine, and all browsers have a developer console. Firefox's FireBug provides very useful error messages and allows for break-points and more, but it's often helpful to run the same code in Chrome and Safari to cross-reference the error output. Even Internet Explorer contains developer tools.

The problem with browsers is that they evaluate JavaScript differently! Though it's not common, it is possible to write code that returns very different results in different browsers. But usually the differences are in the way they treat the document object model and not how prototypes and functions work. Obviously, `Math.sqrt(4)` method returns `2` to all browsers and shells. But the `scrollLeft` method depends on the browser's layout policies.

Writing browser-specific code is a waste of time, and that's another reason why libraries should be used.

Server-side JavaScript

The `Node.js` library has become the standard platform for creating server-side and network-based applications. Can functional programming be used for server-side application programming? Yes! Ok, but do there exist any functional libraries that are designed for this performance-critical environment? The answer to that is also: yes.

All the functional libraries outlined in this chapter will work in the `Node.js` library, and many depend on the `browserify.js` module to work with browser elements.

A functional use case in the server-side environment

In our brave new world of network systems, server-side application developers are often concerned with concurrency, and rightly so. The classic example is an application that allows multiple users to modify the same file. But if they try to modify it at the same time, you will get into an ugly mess. This is the *maintenance of state* problem that has plagued programmers for decades.

Assume the following scenario:

1. One morning, Adam opens a report for editing but he doesn't save it before leaving for lunch.
2. Billy opens the same report, adds his notes, and then saves it.
3. Adam comes back from lunch, adds his notes to the report, and then saves it, unknowingly overwriting Billy's notes.
4. The next day, Billy finds out that his notes are missing. His boss yells at him; everybody gets mad and they gang up on the misguided application developer who unfairly loses his job.

For a long time, the solution to this problem was to create a state about the file. Toggle a lock status to *on* when someone begins editing it, which prevents others from being able to edit it, and then toggle it to *off* once they save it. In our scenario, Billy would not be able to do his work until Adam gets back from lunch. And if it's never saved (if, say, Adam decided to quit his job in the middle of the lunch break), then no one will ever be able to edit it.

This is where functional programming's ideas about immutable data and state (or lack thereof) can really be put to work. Instead of having users modify the file directly, with a functional approach they would modify a copy of the file, which is a new revision. If they go to save the revision and a new revision already exists, then we know that someone else has already modified the old one. Crisis averted.

Now the scenario from before would unfold like this:

1. One morning, Adam opens a report for editing. But he doesn't save it before going to lunch.
2. Billy opens the same report, adds his notes, and saves it as a new revision.
3. Adam returns from lunch to add his notes. When he attempts to save the new revision, the application tells him that a newer revision now exists.
4. Adam opens the new revisions, adds his notes to it, and saves another new revision.
5. By looking at the revision history, the boss sees that everything is working smoothly. Everyone is happy and the application developer gets a promotion and a raise.

This is known as *event sourcing*. There is no explicit state to be maintained, only events. The process is much cleaner and there is a clear history of events that can be reviewed.

This idea and many others are why functional programming in server-side environments is on the rise.

CLI

Although web and the `node.js` library are the two main JavaScript environments, some pragmatic and adventurous users are finding ways to use JavaScript in the command line.

Using JavaScript as a **Command Line Interface** (**CLI**) scripting language might be one of the best opportunities to apply function programming. Imagine being able to use lazy evaluation when searching for local files or to rewrite an entire bash script into a functional JavaScript one-liner.

Using functional libraries with other JavaScript modules

Web applications are made up of all sorts of things: frameworks, libraries, APIs and more. They can work along side each other as dependents, plugins, or just as coexisting objects.

- `Backbone.js`
  - An **MVP** (**model-view-provider**) framework with a RESTful JSON interface
  - Requires the `underscore.js` library, Backbone's only hard dependency
- jQuery
  - The `Bacon.js` library has bindings for mixing with jQuery
  - Underscore and jQuery complement each other very well
- Prototype JavaScript Framework
  - Provides JavaScript with collection functions in the manner closest to Ruby's Enumerable
- `Sugar.js`
  - Modifies native objects and their methods
  - Must be careful when mixing with other libraries, especially Prototype

Functional languages that compile into JavaScript

Sometimes the thick veneer of C-like syntax over JavaScript's inner functionality can be enough to make you want to switch to another functional language. Well, you can!

- Clojure and ClojureScript
  - Closure is a modern Lisp implementation and a full-featured functional language
  - ClojureScript trans-compiles Clojure into JavaScript
- CoffeeScript
  - CoffeeScript is the name of both a functional language and a compiler for trans-compiling the language into JavaScript
  - 1-to-1 mapping between expressions in CoffeeScript and expression in JavaScript

There are many more out there, including **Pyjs**, **Roy**, **TypeScript**, **UHC** and more.

Summary

Which library you choose to use depends on what your needs are. Need functional reactive programming to handle events and dynamic values? Use the `Bacon.js` library. Only need infinite streams and nothing else? Use the `stream.js` library. Want to complement jQuery with functional helpers? Try the `underscore.js` library. Need a structured environment for serious ad hoc polymorphism? Check out the `bilby.js` library. Need a well-rounded tool for functional programming? Use the `Lazy.js` library. Not happy with any of these options? Write your own!

Any library is only as good as the way it's used. Although a few of the libraries outlined in this chapter have a few flaws, most faults occur somewhere between the keyboard and the chair. It's up to you to use the libraries correctly and to suit your needs.

And if we're importing code libraries into our JavaScript environment, then maybe we can import ideas and principles too. Maybe we can channel *The Zen of Python*, by *Tim Peter*:

> *Beautiful is better than ugly*
> *Explicit is better than implicit.*
> *Simple is better than complex.*
> *Complex is better than complicated.*
> *Flat is better than nested.*
> *Sparse is better than dense.*
> *Readability counts.*
> *Special cases aren't special enough to break the rules.*
> *Although practicality beats purity.*
> *Errors should never pass silently.*
> *Unless explicitly silenced.*
> *In the face of ambiguity, refuse the temptation to guess.*
> *There should be one – and preferably only one – obvious way to do it.*
> *Although that way may not be obvious at first unless you're Dutch.*
> *Now is better than never.*
> *Although never is often better than "right" now.*
> *If the implementation is hard to explain, it's a bad idea.*
> *If the implementation is easy to explain, it may be a good idea.*
> *Namespaces are one honking great idea – let's do more of those!*

4
Implementing Functional Programming Techniques in JavaScript

Hold on to your hats because we're really going to get into the functional mind-set now.

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

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

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

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

- Data generic programming
- Mostly functional programming
- Functional reactive programming and more

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

Partial function application and currying

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

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

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

Function manipulation

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

Apply, call, and the this keyword

In pure functional languages, functions are not invoked; they're applied. JavaScript works the same way and even provides utilities for manually calling and applying functions. And it's all about the `this` keyword, which, of course, is the object that the function is a member of.

The `call()` function lets you define the `this` keyword as the first argument. It works as follows:

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

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

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

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

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

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

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



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

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

Binding arguments

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

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

```
function Drum(){
  this.noise = 'boom';
  this.duration = 1000;
  this.goBoom = function(){console.log(this.noise)};
}
var drum = new Drum();
setInterval(drum.goBoom.bind(drum), drum.duration);
```

This solves a lot of problems in object-oriented frameworks, such as Dojo, specifically the problems of maintaining the state when using classes that define their own handler functions. But we can use the `bind()` function for functional programming too.



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

Function factories

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

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

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

Then we can use this to create more generic functions:

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

And it can work on the other argument too:

```
function bindSecondArg(func, b) {
  return function(a) {
    return func(a, b);
  };
}
var squareOf = bindSecondArg(Math.pow, 2);
var cubeOf = bindSecondArg(Math.pow, 3);
console.log(squareOf(3)); // 9
console.log(squareOf(4)); // 16
console.log(cubeOf(3));   // 27
console.log(cubeOf(4));   // 64
```

The ability to create generic functions is very important in functional programming. But there's a clever trick to making this process even more generalized. The `bindFirstArg()` function itself takes two arguments, the first being a function. If we pass the `bindFirstArg` function as a function to itself, we can create *bindable* functions. This can be best described with the following example:

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

This is why they're called function factories.

Partial application

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

What we need is partial application.



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

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

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

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

Partial application from the left

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

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

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



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

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

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

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

```
console.log(nums2hex(100, 200, 255, 0, 123)); // '64c8ff007b'

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

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

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

Partial application from the right

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

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

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

var someShadesOfGreen = nums2hex.partialApplyRight(255, 0);
console.log(shadesOfGreen(123));   // '7bff00'
console.log(shadesOfGreen(100));   // '64ff00'
```

Partial application has allowed us to take a very generic function and extract more specific functions out of it. But the biggest flaw in this method is that the way in which the arguments are passed, as in how many and in what order, can be ambiguous. And ambiguity is never a good thing in programming. There's a better way to do this: currying.

Currying

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

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

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

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

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

The `numArgs` argument lets us optionally specify the number of arguments the function being curried needs if it's not explicitly defined.

Let's look at how to use it within our hexadecimal application. We'll write a function that converts RGB values to a hexadecimal string that is appropriate for HTML:

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

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

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

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

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

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

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

All of this isn't just for the benefit of function factories and code reuse. Currying and partial application play into a bigger pattern known as composition.

Function composition

Finally, we have arrived at function composition.

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



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

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

Compose

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

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

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

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

In math, the composition of the `f` and `g` variables is defined as `f(g(x))`.
In JavaScript, this can be written as:

```
var compose = function(f, g) {
  return function(x) {
    return f(g(x));
  };
};
```

But if we left it at that, we would lose track of the `this` keyword, among other problems. The solution is to use the `apply()` and `call()` utilities. Compared to curry, the `compose()` polyfill is quite simple.

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

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

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

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

Sequence – compose in reverse

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

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

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

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

```
var sequences = function1.sequence(function2).sequence(function3);
console.log( sequences('count') ); // returns 'count 1 2 3'
```

Compositions versus chains

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

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

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

function floorSqrt3(num) {
  return [num].map(Math.sqrt).map(Math.floor).toString();
}
var floorSqrt4 = String.compose(Math.floor).compose(Math.sqrt);
var floorSqrt5 = Math.sqrt.sequence(Math.floor).sequence(String);

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

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

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



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

- The third approach is a chain of array functions, notably the `map` function. This works fairly well, but it is not mathematically correct.
- Here's our `compose()` function in action. All methods are forced to be unary, pure functions that encourage the use of better, simpler, and smaller functions that do one thing and do it well.
- The last approach uses the `compose()` function in reverse sequence, which is just as valid.

Programming with compose

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

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



Type Signatures are used to explicitly declare what types of input the function accepts and what type it outputs. They were first used by Haskell, which actually used them in the function definitions to be used by the compiler. But, in JavaScript, we just put them in a code comment. They look something like this: `foo :: arg1 -> argN -> output`

Examples:

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

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

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

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

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

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

// arrayToString :: [Char] -> String
```

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

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

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

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

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

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

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

Now let's compose some of them together.

```
var lighterColor = arrayToString
  .compose(raiseColorHexes)
  .compose(stringToArray)
  var darkerColor = arrayToString
  .compose(lowerColorHexes)
  .compose(stringToArray)

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

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

```
// component2hex :: Ints -> Int
function componentToHex(c) {
  var hex = c.toString(16);
  return hex.length == 1 ? "0" + hex : hex;
```

```
}

// nums2hex :: Ints* -> Int
function nums2hex() {
  return Array.prototype.map.call(arguments,
  componentToHex).join('');
}
```

First we need to make the curried and partial-applied functions, then we can compose them to our other composed functions.

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

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

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

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

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

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

// and use them like this:
console.log( lighterRedNumSteps(5) ); // Return: 'ff5555'
console.log( lighterRedNumSteps(2) ); // Return: 'ff2222'
```

In the same way, we could easily create more functions for creating lighter and darker blues, greens, grays, purples, anything you want. *This is a really great way to construct an API.*

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

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

Mostly functional programming

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

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

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

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

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

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

```
function Receptor(name, available){
  this.name = name;
  this.available = available; // mutable state
```

```
    this.render = function(){
      output = '<li>';
      output += this.available ?
        this.name + ' is available' :
        this.name + ' is not available';
      output += '</li>';
      return output;
    }
  }
  var me = new Receptor;
  var receptors = app.getReceptors().push(me);
  app.container.innerHTML = receptors.map(function(r){
    return r.render();
  }).join('');
```

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

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

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

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

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

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

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

Handling events

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

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

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

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

Easy as pie.



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

Our boss likes the application so far, but she points out that if an employee never logs out before leaving for the day without closing the page, then the application says the employee is still available.

To figure out if an employee is active on the website, we can monitor the keyboard and mouse events. Let's say they're considered to be unavailable after 30 minutes of no activity.

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

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

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

Functional reactive programming

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

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

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

And we'll use the `Bacon.js` library, which will allow us to quickly develop **Functional Reactive Programming** (**FRP**) applications. The application will only be used one day out of the year (Election Day), and our boss thinks it should take a proportional amount of time. With functional programming and a library such as `Bacon.js`, we'll get it done in half the time.

But first, we're going to need some objects to represent the voting regions, such as states, provinces, districts, and so on.

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

  // return an HTML representation
  this.render = function(){
    var lis = this.parties.map(function(p){
      return '<li>' + p.name + ': ' + p.votes + '</li>';
    });
    var output = '<h2>' + this.name + '</h2>';
    output += '<ul>' + lis.join('') + '</ul>';
    output += 'Percent reported: ' + this.percent;
    return output;
  }
}
function getRegions(data) {
  return JSON.parse(data).map(function(obj){
    return new Region(obj.name, obj.percent, obj.parties);
  });
}
var url = 'http://api.server.com/election-data?format=json';
var data = jQuery.ajax(url);
var regions = getRegions(data);
app.container.innerHTML = regions.map(function(r){
  return r.render();
}).join('');
```

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

Reactivity

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

```
var eventStream = Bacon.fromPoll(10000, function(){
  return Bacon.Next;
});
var subscriber = eventStream.subscribe(function(){
  var url = 'http://api.server.com/election-data?format=json';
```

```
    var data = jQuery.ajax(url);
    var newRegions = getRegions(data);
    container.innerHTML = newRegions.map(function(r){
      return r.render();
    }).join('');
  });
```

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

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

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

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

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

Putting it all together

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

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

```
// create the eventStream out side of the functions
var eventStream = Bacon.onPromise(jQuery.ajax(url));
var subscribe = null;
```

```
var url = 'http://api.server.com/election-data?format=json';

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

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

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

```
      container.innerHTML = newRegions.map(function(r) {
        return r.render();
      }).join('');
    });
  });
```

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

Summary

JavaScript is a beautiful language.

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

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

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

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

5
Category Theory

Thomas Watson was famously quoted as saying, "I think there is a world market for maybe five computers". That was in 1948. Back then, everybody knew that computers would only be used for two things: math and engineering. Not even the biggest minds in tech could predict that, one day, computers would be able to translate Spanish to English, or simulate entire weather systems. At the time, the fastest machine was IBM's SSEC, clocking in at 50 multiplications per second, the display terminal wasn't due until 15 years later and multiple-processing meant multiple user terminals sharing a single processor. The transistor changed everything, but tech's visionaries still missed the mark. Ken Olson made another famously foolish prediction when, in 1977, he said "There is no reason anyone would want a computer in their home".

It seams obvious to us now that computers are not just for scientists and engineers, but that's hindsight. The idea that machines can do more than just math was anything but intuitive 70 years ago. Watson didn't just fail to realize how computers could transform a society, he failed to realize the transformative and evolving powers of mathematics.

But the potential of computers and math was not lost on everybody. John McCarthy invented **Lisp** in 1958, a revolutionary algorithm-based language that ushered in a new era in computing. Since its inception, Lisp was instrumental in the idea of using abstraction layers—compilers, interpreters, virtualization—to push forward the progression of computers from hardcore math machines to what they are today.

From Lisp came **Scheme**, a direct ancestor of JavaScript. Now that brings us full circle. If computers are, at their core, machines that just do math, then it stands to reason that a math-based programming paradigm would excel.

The term "math" is being used here not to describe the "number crunching" that computers can obviously do, but to describe *discrete mathematics*: the study of discrete, mathematical structures such as statements in logic or the instructions of a computer language. By treating code as a discrete mathematical structure, we can apply concepts and ideas in math to it. This is what has made functional programming so instrumental in artificial intelligence, graph search, pattern recognition and other big challenges in computer science.

In this chapter, we will experiment with some of these concepts and their applications in everyday programming challenges. They will include:

- Category theory
- Morphisms
- Functors
- Maybes
- Promises
- Lenses
- Function composition

With these concepts, we'll be able to write entire libraries and APIs very easily and safely. And we'll go from explaining category theory to formally implementing it in JavaScript.

Category theory

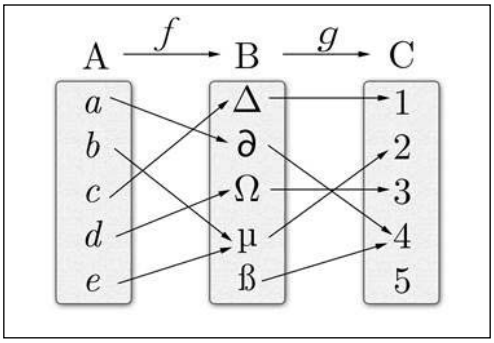
Category theory is the theoretical concept that empowers function composition. Category theory and function composition go together like engine displacement and horsepower, like NASA and the space shuttle, like good beer and a mug to pour it in. Basically, you can't have one without the other.

Category theory in a nutshell

Category theory really isn't too difficult a concept. Its place in math is large enough to fill up an entire graduate-level college course, but its place in computer programming can be summed up quite easily.

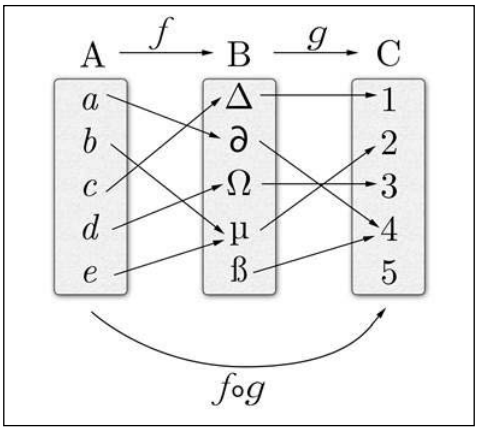
Einstein once said, "If you can't explain it to a 6-year-old, you don't know it yourself". Thus, in the spirit of explaining it to a 6-year-old, *category theory is just connecting the dots*. Although it may be grossly over-simplifying category theory, it does do a good job of explaining what we need to know in a straightforward manner.

First you'll need to know some terminology. **Categories** are just sets with the same type. In JavaScript, they're arrays or objects that contain variables that are explicitly declared as numbers, strings, Booleans, dates, nodes, and so on. **Morphisms** are pure functions that, when given a specific set of inputs, always return the same output. **Homomorphic operations** are restricted to a single category, while **polymorphic operations** can operate on multiple categories. For example, the homomorphic function *multiplication* only works on numbers, but the polymorphic function addition can work on strings too.

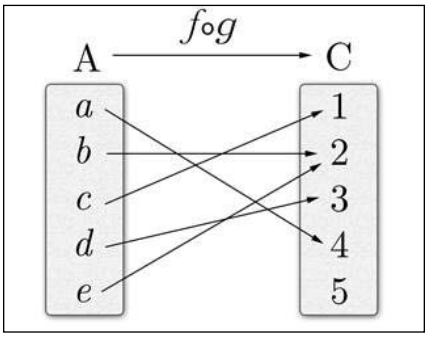


The following diagram shows three categories—A, B, and C—and two morphisms—f and g.

Category theory tells us that, when we have two morphisms where the category of the first one is the expected input of the other, then they can be *composed* to the following:



The *f o g* symbol is the composition of morphisms *f* and *g*. Now we can just connect the dots.



And that's all it really is, just connecting dots.

Type safety

Let's connect some dots. Categories contain two things:

1. Objects (in JavaScript, types).
2. Morphisms (in JavaScript, pure functions that only work on types).

These are the terms given to category theory by mathematicians, so there is some unfortunate nomenclature overloading with our JavaScript terminology. **Objects** in category theory are more like variables with an explicit data type and not collections of properties and values like in the JavaScript definition of objects. **Morphisms** are just pure functions that use those types.

So applying the idea of category theory to JavaScript is pretty easy. Using category theory in JavaScript means working with one certain data type per category. Data types are numbers, strings, arrays, dates, objects, Booleans, and so on. But, with no strict type system in JavaScript, things can go awry. So we'll have to implement our own method of ensuring that the data is correct.

There are four primitive data types in JavaScript: numbers, strings, Booleans, and functions. We can create *type safety functions* that either return the variable or throw an error. *This fulfils the object axiom of categories*.

```
var str = function(s) {
  if (typeof s === "string") {
    return s;
  }
```

```
  else {
    throw new TypeError("Error: String expected, " + typeof s + "
    given.");
  }
}
var num = function(n) {
  if (typeof n === "number") {
    return n;
  }
  else {
    throw new TypeError("Error: Number expected, " + typeof n + "
    given.");
  }
}
var bool = function(b) {
  if (typeof b === "boolean") {
    return b;
  }
  else {
    throw new TypeError("Error: Boolean expected, " + typeof b + "
    given.");
  }
}
var func = function(f) {
  if (typeof f === "function") {
    return f;
  }
  else {
    throw new TypeError("Error: Function expected, " + typeof f +
    " given.");
  }
}
```

However, there's a lot of repeated code here and that isn't very functional. Instead, we can create a function that returns another function that is the type safety function.

```
var typeOf = function(type) {
  return function(x) {
    if (typeof x === type) {
      return x;
    }
    else {
      throw new TypeError("Error: "+type+" expected, "+typeof x+"
      given.");
    }
```

```
    }
}
var str = typeOf('string'),
  num = typeOf('number'),
  func = typeOf('function'),
  bool = typeOf('boolean');
```

Now, we can use them to ensure that our functions behave as expected.

```
// unprotected method:
var x = '24';
x + 1; // will return '241', not 25

// protected method
// plusplus :: Int -> Int
function plusplus(n) {
  return num(n) + 1;
}
plusplus(x); // throws error, preferred over unexpected output
```

Let's look at a meatier example. If we want to check the length of a Unix timestamp that is returned by the JavaScript function `Date.parse()`, not as a string but as a number, then we'll have to use our `str()` function.

```
// timestampLength :: String -> Int
function timestampLength(t) { return num(str(t).length); }
timestampLength(Date.parse('12/31/1999')); // throws error
timestampLength(Date.parse('12/31/1999')
  .toString()); // returns 12
```

Functions like this that explicitly transform one type to another (or to the same type) are called *morphisms*. *This fulfils the morphism axiom of category theory*. These forced type declarations via the type safety functions and the morphisms that use them are everything we need to represent the notion of a category in JavaScript.

Object identities

There's one other important data type: objects.

```
var obj = typeOf('object');
obj(123); // throws error
obj({x:'a'}); // returns {x:'a'}
```

However, objects are different. They can be inherited. Everything that is not a primitive—numbers, strings, Booleans, and functions—is an object, including arrays, dates, elements, and more.

There's no way to know what type of object something is, as in to know what sub-type a JavaScript 'object' is, from the `typeof` keyword, so we'll have to improvise. Objects have a `toString()` function that we can hijack for this purpose.

```
var obj = function(o) {
  if (Object.prototype.toString.call(o)==="[object Object]") {
    return o;
  }
  else {
    throw new TypeError("Error: Object expected, something else
    given.");
  }
}
```

Again, with all the objects out there, we should implement some code re-use.

```
var objectTypeOf = function(name) {
  return function(o) {
    if (Object.prototype.toString.call(o) === "[object "+name+"]")
    {
      return o;
    }
    else {
      throw new TypeError("Error: '+name+' expected, something
      else given.");
    }
  }
}
var obj = objectTypeOf('Object');
var arr = objectTypeOf('Array');
var date = objectTypeOf('Date');
var div = objectTypeOf('HTMLDivElement');
```

These will be very useful for our next topic: functors.

Functors

While morphisms are mappings between types, *functors* are mappings between categories. They can be thought of as functions that lift values out of a container, morph them, and then put them into a new container. The first input is a morphism for the type and the second input is the container.



The type signature for functors looks like this:

```
// myFunctor :: (a -> b) -> f a -> f b
```

This says, "give me a function that takes `a` and returns `b` and a box that contains `a`(s), and I'll return a box that contains `b`(s).

Creating functors

It turns out we already have one functor: `map()`. It grabs the values within the container, an array, and applies a function to it.

```
[1, 4, 9].map(Math.sqrt); // Returns: [1, 2, 3]
```

However, we'll need to write it as a global function and not as a method of the array object. This will allow us to write cleaner, safer code later on.

```
// map :: (a -> b) -> [a] -> [b]
var map = function(f, a) {
  return arr(a).map(func(f));
}
```

This example seems like a contrived wrapper because we're just piggybacking onto the `map()` function. But it serves a purpose. It provides a template for maps of other types.

```
// strmap :: (str -> str) -> str -> str
var strmap = function(f, s) {
  return str(s).split('').map(func(f)).join('');
}

// MyObject#map :: (myValue -> a) -> a
MyObject.prototype.map(f{
  return func(f)(this.myValue);
}
```

Arrays and functors

Arrays are the preferred way to work with data in functional JavaScript.

Is there an easier way to create functors that are already assigned to a morphism? Yes, and it's called `arrayOf`. When you pass in a morphism that expects an integer and returns an array, you get back a morphism that expects an array of integers and returns an array of arrays.

It is not a functor itself, but it allows us to create functors from morphisms.

```
// arrayOf :: (a -> b) -> ([a] -> [b])
var arrayOf = function(f) {
  return function(a) {
    return map(func(f), arr(a));
  }
}
```

Here's how to create functors by using morphism:

```
var plusplusall = arrayOf(plusplus); // plusplus is our morphism
console.log( plusplusall([1,2,3]) ); // returns [2,3,4]
console.log( plusplusall([1,'2',3]) ); // error is thrown
```

The interesting property of the `arrayOf` functor is that it works on type safeties as well. When you pass in the type safety function for strings, you get back a type safety function for an array of strings. The type safeties are treated like the *identity function* morphism. This can be very useful for ensuring that an array contains all the correct types.

```
var strs = arrayOf(str);
console.log( strs(['a','b','c']) ); // returns ['a','b','c']
console.log( strs(['a',2,'c']) ); // throws error
```

Function compositions, revisited

Functions are another type of primitive that we can create a functor for. And that functor is called `fcompose`. We defined functors as something that takes a value from a container and applies a function to it. When that container is a function, we just call it to get its inner value.

We already know what function compositions are, but let's look at what they can do in a category theory-driven environment.

Function compositions are associative. If your high school algebra teacher was like mine, she taught you what the property *is* but not what it can *do*. In practice, compose is what the associative property can do.

$$(a \times b) \times c = a \times (b \times c)$$
$$(f \circ g) \circ h = f \circ (g \circ h)$$

$$f \circ g \neq g \circ f$$

We can do any inner-compose, it doesn't matter how it's grouped. This is not to be confused with the commutative property. *f o g* does not always equal *g o f*. In other words, the reverse of the first word of a string is not the same as the first word of the reverse of a string.

What this all means is that it doesn't matter which functions are applied and in what order, as long as the input of each functions comes from the output of the previous function. But wait, if the function on the right relies on the function on the left, then can't there be only one order of evaluation? Left to right? True, but if it's encapsulated, then we can control it however we feel fit. This is what empowered lazy evaluation in JavaScript.

$$(a \times b) \times c = a \times (b \times c)$$
$$(f \circ g) \circ h = f \circ (g \circ h)$$

Let's rewrite function composition, not as an extension of the function prototype, but as a stand-alone function that will allow us to get more out of it. The basic form is as follows:

```
var fcompose = function(f, g) {
  return function() {
    return f.call(this, g.apply(this, arguments));
  };
};
```

But we'll need it to work on any number of inputs.

```
var fcompose = function() {
  // first make sure all arguments are functions
  var funcs = arrayOf(func)(arguments);

  // return a function that applies all the functions
  return function() {
    var argsOfFuncs = arguments;
    for (var i = funcs.length; i > 0; i -= 1) {
      argsOfFuncs  = [funcs[i].apply(this, args)];
    }
    return args[0];
  };
};

// example:
```

```
var f = fcompose(negate, square, mult2, add1);
f(2); // Returns: -36
```

Now that we've encapsulated the functions, we have control over them. We could rewrite the compose function such that *each function accepts another function as input, stores it, and gives back an object that does the same*. Instead of accepting an array as an input, doing something with it, and then giving back a new array for each operation, we can accept a single array for each element in the source, perform all operations combined (every `map()`, `filter()`, and so on, composed together), and finally store the results in a new array. This is lazy evaluation via function composition. No reason to reinvent the wheel here. Many libraries have a nice implementation of this concept, including the `Lazy.js`, `Bacon.js` and `wu.js` libraries.

There's a lot more we can do as a result of this different model: asynchronous iteration, asynchronous event handling, lazy evaluation, and even automatic parallelization.



Automatic parallelization? There's a word for that in the computer science industry: IMPOSSIBLE. But is it really impossible? The next evolutionary leap in Moore's law might be a compiler that parallelizes our code for us, and could function composition be it?

No, it doesn't quite work that way. The JavaScript engine is what is really doing the parallelization, not automatically but with well thought-out code. Compose just gives the engine the chance to split it into parallel processes. But that in itself is pretty cool.

Monads

Monads are tools that help you compose functions.

Like primitive types, monads are structures that can be used as the containers that functors "reach into". The functors grab the data, do something to it, put it into a new monad, and return it.

There are three monads we'll focus on:

- Maybes
- Promises
- Lenses

So in addition to arrays (map) and functions (compose), we'll have five functors (map, compose, maybe, promise and lens). These are just some of the many other functors and monads that are out there.

Maybes

Maybes allow us to gracefully work with data that might be null and to have defaults. A maybe is a variable that either has some value or it doesn't. And it doesn't matter to the caller.

On its own, it might seem like this is not that big a deal. Everybody knows that null-checks are easily accomplished with an `if-else` statement:

```
if (getUsername() == null ) {
  username = 'Anonymous') {
else {
  username = getUsername();
}
```

But with functional programming, we're breaking away from the procedural, line-by-line way of doing things and instead working with pipelines of functions and data. If we had to break the chain in the middle just to check if the value existed or not, we would have to create temporary variables and write more code. Maybes are just tools to help us keep the logic flowing through the pipeline.

To implement maybes, we'll first need to create some constructors.

```
// the Maybe monad constructor, empty for now
var Maybe = function(){};

// the None instance, a wrapper for an object with no value
var None = function(){};
None.prototype = Object.create(Maybe.prototype);
None.prototype.toString = function(){return 'None';};

// now we can write the `none` function
// saves us from having to write `new None()` all the time
var none = function(){return new None()};

// and the Just instance, a wrapper for an object with a value
var Just = function(x){return this.x = x;};
Just.prototype = Object.create(Maybe.prototype);
Just.prototype.toString = function(){return "Just "+this.x;};
var just = function(x) {return new Just(x)};
```

Finally, we can write the `maybe` function. It returns a new function that either returns nothing or a maybe. *It is a functor.*

```
var maybe = function(m){
  if (m instanceof None) {
```

```
    return m;
  }
  else if (m instanceof Just) {
    return just(m.x);
  }
  else {
    throw new TypeError("Error: Just or None expected, " +
    m.toString() + " given.");
  }
}
```

And we can also create a functor generator just like we did with arrays.

```
var maybeOf = function(f){
  return function(m) {
    if (m instanceof None) {
      return m;
    }
    else if (m instanceof Just) {
      return just(f(m.x));
    }
    else {
      throw new TypeError("Error: Just or None expected, " +
      m.toString() + " given.");
    }
  }
}
```

So `Maybe` is a monad, `maybe` is a functor, and `maybeOf` returns a functor that is already assigned to a morphism.

We'll need one more thing before we can move forward. We'll need to add a method to the `Maybe` monad object that helps us use it more intuitively.

```
Maybe.prototype.orElse = function(y) {
  if (this instanceof Just) {
    return this.x;
  }
  else {
    return y;
  }
}
```

In its raw form, maybes can be used directly.

```
maybe(just(123)).x; // Returns 123
maybeOf(plusplus)(just(123)).x; // Returns 124
maybe(plusplus)(none()).orElse('none'); // returns 'none'
```

Anything that returns a method that is then executed is complicated enough to be begging for trouble. So we can make it a little cleaner by calling on our `curry()` function.

```
maybePlusPlus = maybeOf.curry()(plusplus);
maybePlusPlus(just(123)).x; // returns 123
maybePlusPlus(none()).orElse('none'); // returns none
```

But the real power of maybes will become clear when the dirty business of directly calling the `none()` and `just()` functions is abstracted. We'll do this with an example object `User`, that uses maybes for the username.

```
var User = function(){
  this.username = none(); // initially set to `none`
};
User.prototype.setUsername = function(name) {
  this.username = just(str(name)); // it's now a `just
};
User.prototype.getUsernameMaybe = function() {
  var usernameMaybe = maybeOf.curry()(str);
  return usernameMaybe(this.username).orElse('anonymous');
};

var user = new User();
user.getUsernameMaybe(); // Returns 'anonymous'

user.setUsername('Laura');
user.getUsernameMaybe(); // Returns 'Laura'
```

And now we have a powerful and safe way to define defaults. Keep this `User` object in mind because we'll be using it later on in this chapter.

Promises

The nature of promises is that they remain immune to changing circumstances.
- Frank Underwood, House of Cards

In functional programming, we're often working with pipelines and data flows: chains of functions where each function produces a data type that is consumed by the next. However, many of these functions are asynchronous: readFile, events, AJAX, and so on. Instead of using a continuation-passing style and deeply nested callbacks, how can we modify the return types of these functions to indicate the result? By wrapping them in *promises*.

Promises are like the functional equivalent of callbacks. Obviously, callbacks are not all that functional because, if more than one function is mutating the same data, then there can be race conditions and bugs. Promises solve that problem.

You should use promises to turn this:

```
fs.readFile("file.json", function(err, val) {
  if( err ) {
    console.error("unable to read file");
  }
  else {
    try {
      val = JSON.parse(val);
      console.log(val.success);
    }
    catch( e ) {
      console.error("invalid json in file");
    }
  }
});
```

Into the following code snippet:

```
fs.readFileAsync("file.json").then(JSON.parse)
  .then(function(val) {
    console.log(val.success);
  })
  .catch(SyntaxError, function(e) {
    console.error("invalid json in file");
  })
  .catch(function(e){
    console.error("unable to read file")
  });
```

The preceding code is from the README for *bluebird*: a full featured *Promises/A+* implementation with exceptionally good performance. *Promises/A+* is a specification for implementing promises in JavaScript. Given its current debate within the JavaScript community, we'll leave the implementations up to the *Promises/A+* team, as it is much more complex than maybes.

But here's a partial implementation:

```
// the Promise monad
var Promise = require('bluebird');

// the promise functor
var promise = function(fn, receiver) {
  return function() {
    var slice = Array.prototype.slice,
    args = slice.call(arguments, 0, fn.length - 1),
    promise = new Promise();
    args.push(function() {
      var results = slice.call(arguments),
      error = results.shift();
      if (error) promise.reject(error);
      else promise.resolve.apply(promise, results);
    });
    fn.apply(receiver, args);
    return promise;
  };
};
```

Now we can use the `promise()` functor to transform functions that take callbacks into functions that return promises.

```
var files = ['a.json', 'b.json', 'c.json'];
readFileAsync = promise(fs.readFile);
var data = files
  .map(function(f){
    readFileAsync(f).then(JSON.parse)
  })
  .reduce(function(a,b){
    return $.extend({}, a, b)
  });
```

Lenses

Another reason why programmers really like monads is that they make writing libraries very easy. To explore this, let's extend our `User` object with more functions for getting and setting values but, instead of using getters and setters, we'll use *lenses*.

Lenses are first-class getters and setters. They allow us to not just get and set variables, but also to run functions over it. But instead of mutating the data, they clone and return the new data modified by the function. They force data to be immutable, which is great for security and consistency as well for libraries. They're great for elegant code no matter what the application, so long as the performance-hit of introducing additional array copies is not a critical issue.

Before we write the `lens()` function, let's look at how it works.

```
var first = lens(
  function (a) { return arr(a)[0]; }, // get
  function (a, b) { return [b].concat(arr(a).slice(1)); } // set
);
first([1, 2, 3]); // outputs 1
first.set([1, 2, 3], 5); // outputs [5, 2, 3]
function tenTimes(x) { return x * 10 }
first.modify(tenTimes, [1,2,3]); // outputs [10,2,3]
```

And here's how the `lens()` function works. It returns a function with get, set and mod defined. The `lens()` function itself is a functor.

```
var lens = fuction(get, set) {
  var f = function (a) {return get(a)};
  f.get = function (a) {return get(a)};
  f.set = set;
  f.mod = function (f, a) {return set(a, f(get(a)))};
  return f;
};
```

Let's try an example. We'll extend our `User` object from the previous example.

```
// userName :: User -> str
var userName = lens(
  function (u) {return u.getUsernameMaybe()}, // get
  function (u, v) { // set
    u.setUsername(v);
    return u.getUsernameMaybe();
  }
);

var bob = new User();
bob.setUsername('Bob');
userName.get(bob); // returns 'Bob'
userName.set(bob, 'Bobby'); //return 'Bobby'
userName.get(bob); // returns 'Bobby'
userName.mod(strToUpper, bob); // returns 'BOBBY'
strToUpper.compose(userName.set)(bob, 'robert'); // returns
'ROBERT'
userName.get(bob); // returns 'robert'
```

jQuery is a monad

If you think all this abstract babble about categories, functors, and monads has no real-world application, think again. jQuery, the popular JavaScript library that provides an enhanced interface for working with HTML is, in-fact, a monadic library.

The `jQuery` object is a monad and its methods are functors. Really, they're a special type of functor called *endofunctors*. **Endofunctors** are functors that return the same category as the input, that is, `F :: X -> X`. Each `jQuery` method takes a `jQuery` object and returns a `jQuery` object, which allows methods to be chained, and they will have the type signature `jFunc :: jquery-obj -> jquery-obj`.

```
$('li').add('p.me-too').css('color', 'red').attr({id:'foo'});
```

This is also what empowers jQuery's plugin framework. If the plugin takes a `jQuery` object as input and returns one as output, then it can be inserted into the chain.

Let's look at how jQuery was able to implement this.

Monads are the containers that the functors "reach into" to get the data. In this way, the data can be protected and controlled by the library. jQuery provides access to the underlying data, a wrapped set of HTML elements, via its many methods.

The `jQuery` object itself is written as the result of an anonymous function call.

```
var jQuery = (function () {
  var j = function (selector, context) {
    var jq-obj = new j.fn.init(selector, context);
    return jq-obj;
  };

  j.fn = j.prototype = {
    init: function (selector, context) {
      if (!selector) {
        return this;
      }
    }
  };
  j.fn.init.prototype = j.fn;
  return j;
})();
```

In this highly simplified version of jQuery, it returns a function that defines the `j` object, which is actually just an enhanced `init` constructor.

```
var $ = jQuery(); // the function is returned and assigned to `$`
var x = $('#select-me'); // jQuery object is returned
```

In the same way that functors lift values out of a container, jQuery wraps the HTML elements and provides access to them as opposed to modifying the HTML elements directly.

jQuery doesn't advertise this often, but it has its own `map()` method for lifting the HTML element objects out of the wrapper. Just like the `fmap()` method, the elements are lifted, something is done with them, and then they're placed back into the container. This is how many of jQuery's commands work in the backend.

```
$('li').map(function(index, element) {
  // do something to the element
  return element
});
```

Another library for working with HTML elements, Prototype, does not work like this. Prototype alters the HTML elements directly via helpers. Consequently, it has not faired as well in the JavaScript community.

Implementing categories

It's about time we formally defined category theory as JavaScript objects. Categories are objects (types) and morphisms (functions that only work on those types). It's an extremely high-level, totally-declarative way to program, but it ensures that the code is extremely safe and reliable—perfect for APIs and libraries that are worried about concurrency and type safety.

First, we'll need a function that helps us create morphisms. We'll call it `homoMorph()` because they'll be homomorphisms. It will return a function that expects a function to be passed in and produces the composition of it, based on the inputs. The inputs are the types that the morphism accepts as input and gives as output. Just like our type signatures, that is, `// morph :: num -> num -> [num]`, only the last one is the output.

```
var homoMorph = function( /* input1, input2,..., inputN, output */
) {
  var before =
  checkTypes(arrayOf(func)(Array.prototype.slice.call(arguments,
  0, arguments.length-1)));
```

```
    var after = func(arguments[arguments.length-1])
    return function(middle) {
      return function(args) {
        return after(middle.apply(this, before
        ([].slice.apply(arguments))));
      }
    }
  }

  // now we don't need to add type signature comments
  // because now they're built right into the function declaration
  add = homoMorph(num, num, num)(function(a,b){return a+b})
  add(12,24); // returns 36
  add('a', 'b'); // throws error
  homoMorph(num, num, num)(function(a,b){
    return a+b;
  })(18, 24); // returns 42
```

The `homoMorph()` function is fairly complex. It uses a closure (see *Chapter 2, Fundamentals of Functional Programming*) to return a function that accepts a function and checks its input and output values for type safety. And for that, it relies on a helper function: `checkTypes`, which is defined as follows:

```
var checkTypes = function( typeSafeties ) {
  arrayOf(func)(arr(typeSafeties));
  var argLength = typeSafeties.length;
  return function(args) {
    arr(args);
    if (args.length != argLength) {
      throw new TypeError('Expected '+ argLength + ' arguments');
    }
    var results = [];
    for (var i=0; i<argLength; i++) {
      results[i] = typeSafeties[i](args[i]);
    }
    return results;
  }
}
```

Now let's formally define some homomorphisms.

```
var lensHM = homoMorph(func, func, func)(lens);
var userNameHM = lensHM(
  function (u) {return u.getUsernameMaybe()}, // get
  function (u, v) { // set
```

```
    u.setUsername(v);
    return u.getUsernameMaybe();
  }
)
var strToUpperCase = homoMorph(str, str)(function(s) {
  return s.toUpperCase();
});
var morphFirstLetter = homoMorph(func, str, str)(function(f, s) {
  return f(s[0]).concat(s.slice(1));
});
var capFirstLetter = homoMorph(str, str)(function(s) {
  return morphFirstLetter(strToUpperCase, s)
});
```

Finally, we can bring it on home. The following example includes function composition, lenses, homomorphisms, and more.

```
// homomorphic lenses
var bill = new User();
userNameHM.set(bill, 'William'); // Returns: 'William'
userNameHM.get(bill); // Returns: 'William'

// compose
var capatolizedUsername = fcompose(capFirstLetter,userNameHM.get);
capatolizedUsername(bill, 'bill'); // Returns: 'Bill'

// it's a good idea to use homoMorph on .set and .get too
var getUserName = homoMorph(obj, str)(userNameHM.get);
var setUserName = homoMorph(obj, str, str)(userNameHM.set);
getUserName(bill); // Returns: 'Bill'
setUserName(bill, 'Billy'); // Returns: 'Billy'

// now we can rewrite capatolizeUsername with the new setter
capatolizedUsername = fcompose(capFirstLetter, setUserName);
capatolizedUsername(bill, 'will'); // Returns: 'Will'
getUserName(bill); // Returns: 'will'
```

The preceding code is extremely declarative, safe, reliable, and dependable.



What does it mean for code to be declarative? In imperative programming, we write sequences of instructions that tell the machine how to do what we want. In functional programming, we describe relationships between values that tell the machine what we want it to compute, and the machine figures out the instruction sequences to make it happen. Functional programming is declarative.

Entire libraries and APIs can be constructed this way that allow programmers to write code freely without worrying about concurrency and type safety because those worries are handled in the backend.

Summary

About one in every 2,000 people has a condition known as synesthesia, a neurological phenomenon in which one sensory input bleeds into another. The most common form involves assigning colors with letters. However, there is an even rarer form where sentences and paragraphs are associated with tastes and feelings.

For these people, they don't read word by word, sentence by sentence. They look at the whole page/document/program and get a sense for how it *tastes*—not in the mouth but in the *mind*. Then they put the parts of the text together like the pieces of a puzzle.

This is what it is like to write fully declarative code: code that describes the relationships between values that tells the machine what we want it to compute. The parts of the program are not instructions in line-by-line order. Synesthetics may be able to do it naturally, but with a little practice anyone can learn how to put the relational puzzle pieces together.

In this chapter, we looked at several mathematical concepts that apply to functional programming and how they allow us to build relationships between data. Next, we'll explore recursion and other advanced topics in JavaScript.

6
Advanced Topics and Pitfalls in JavaScript

JavaScript has been called the "assembly language of the web". The analogy (it isn't perfect, but which analogy is?) draws from the fact that JavaScipt is often a target for compilation, namely from **Clojure** and **CoffeeScript**, but also from many other sources such as **pyjamas** (python to JS) and Google Web Kit (Java to JS).

But the analogy also references the foolish idea that JavaScript is as expressive and low-level as x86 assembly. Perhaps this notion stems from the fact that JavaScript has been bashed for its design flaws and oversights ever since it was first shipped with Netscape back in 1995. It was developed and released in a hurry, before it could be fully developed. And because of that, some questionable design choices made its way into JavaScript, the language that soon became the de-facto scripting language of the web. Semicolons were a big mistake. So were its ambiguous methods for defining functions. Is it `var foo = function();` or `function foo();`?

Functional programming is an excellent way to side-step some of these mistakes. By focusing on the fact that JavaScript is truly a functional language, it becomes clear that, in the preceding example about the different ways to declare a function, it's best to declare functions as variables. And that semicolons are mostly just syntactic sugar to make JavaScript appear more C-like.

But always remember the language you are working with. JavaScript, like any other language, has its pitfalls. And, when programming in a style that often skirts the bleeding edge of what's possible, those minor stumbles can become non-recoverable gotchas. Some of these gotchas include:

- Recursion
- Variable scope and closures
- Function declarations vs. function expressions

However, these issues can be overcome with a little attention.

Recursion

Recursion is very important to functional programming in any language. Many functional languages go so far as to require recursion for iteration by not providing `for` and `while` loop statements; this is only possible when tail-call elimination is guaranteed by the language, which is not the case for JavaScript. A quick primer on recursion was given in *Chapter 2*, *Fundamentals of Functional Programming*. But in this section, we'll dig deeper into exactly how recursion works in JavaScript.

Tail recursion

JavaScript's routine for handling recursion is known as *tail recursion*, a stack-based implementation of recursion. This means that, for every recursive call, there is a new frame in the stack.

To illustrate the problems that can arise from this method, let's use the classic recursive algorithm for factorials.

```
var factorial = function(n) {
  if (n == 0) {
    // base case
    return 1;
  }
  else {
    // recursive case
    return n * factorial(n-1);
  }
}
```

The algorithm will call itself `n` times to get the answer. It's literally computing `(1 x 1 x 2 x 3 x … x N)`. That means the time complexity is `O(n)`.



`O(n)`, pronounced "big oh to the n," means that the complexity of the algorithm will grow at a rate of `n` as the size of the input grows, which is leaner growth. `O(n2)` is exponential growth, `O(log(n))` is logarithmic growth, and so on. This notation can be used for time complexity as well as space complexity.

But, because a new frame in the memory stack is allocated for each iteration, the space complexity is also `O(n)`. This is a problem. This means that memory will be consumed at such a rate the memory limit will be exceeded far too easily. On my laptop, `factorial(23456)` returns `Uncaught Error: RangeError: Maximum call stack size exceeded`.

While calculating the factorial of 23,456 is a frivolous endeavor, you can be assured that many problems that are solved with recursion will grow to that size without too much trouble. Consider the case of data trees. The tree could be anything: search applications, file systems, routing tables, and so on. Below is a very simple implementation of the tree traversal function:

```
var traverse = function(node) {
  node.doSomething(); // whatever work needs to be done
  node.childern.forEach(traverse); // many recursive calls
}
```

With just two children per node, both time complexity and space complexity, (in the worst case, where the entire tree must be traversed to find the answer), would be `O(n2)` because there would be two recursive calls each. With many children per node, the complexity would be `O(nm)` where `m` is the number of children. And recursion is the preferred algorithm for tree traversal; a `while` loop would be much more complex and would require the maintenance of a stack.

Exponential growth like this would mean that it would not take a very large tree to throw a `RangeError` exception. There must be a better way.

The Tail-call elimination

We need a way to eliminate the allocation of new stack frames for every recursive call. This is known as *tail-call elimination*.

With tail-call elimination, when a function returns the result of calling itself, the language doesn't actually perform another function call. It turns the whole thing into a loop for you.

OK, so how do we do this? With lazy evaluation. If we could rewrite it to fold over a lazy sequence, such that the function returns a value or it returns the result of calling another function without doing anything with that result, then new stack frames don't need to be allocated.

To put it in "tail recursion form", the factorial function would have to be rewritten such that the inner procedure `fact` calls itself last in the control flow, as shown in the following code snippet:

```
var factorial = function(n) {
  var _fact = function(x, n) {
    if (n == 0) {
      // base case
      return x;
    }
    else {
      // recursive case
      return _fact(n*x, n-1);
    }
  }
  return fact(1, n);
}
```



Instead of having the result produced by the first function in the recursion tail (like in `n * factorial(n-1)`), the result is computed going down the recursion tail (with the call to `_fact(r*n, n-1)`) and is produced by the last function in this tail (with `return r;`). The computation goes only one way down, not on its way up. It's relatively easy to process it as an iteration for the interpreter.

However, *tail-call elimination does not work in JavaScript*. Put the above code into your favorite JavaScript engine and `factorial(24567)` still returns `Uncaught Error: RangeError: Maximum call stack size exceeded` exception. Tail-call elimination is listed as a new feature to be included in the next release of ECMAScript, but it will be some time before all browsers implement it.

JavaScript cannot optimize functions that are put into tail recursion form. It's a feature of the language specification and runtime interpreter, plain and simple. It has to do with how the interpreter acquires resources for stack frames. Some languages will reuse the same stack frame when it doesn't need to remember anything new, like in the preceding function. This is how tail-call elimination reduces both time and space complexity.

Unfortunately, JavaScript does not do this. But if it did, it would reorganize the stack frames from this:

```
call factorial (3)
  call fact (3 1)
    call fact (2 3)
      call fact (1 6)
        call fact (0 6)
        return 6
      return 6
    return 6
  return 6
return 6
```

into the following:

```
call factorial (3)
  call fact (3 1)
  call fact (2 3)
  call fact (1 6)
  call fact (0 6)
  return 6
return 6
```

Trampolining

The solution? A process known as **trampolining**. It's a way to "hack" the concept of tail-call elimination into a program by using **thunks**.



Thunks are, for this purpose, expressions with arguments that wrap anonymous functions with no arguments of their own. For example: `function(str){return function(){console.log(str)}}`. This prevents the expression from being evaluated until a receiving function calls the anonymous function.

A trampoline is a function that takes a function as input and repeatedly executes its returned value until something other than a function is returned. A simple implementation is shown in the following code snippet:

```
var trampoline = function(f) {
  while (f && f instanceof Function) {
    f = f.apply(f.context, f.args);
  }
  return f;
}
```

To actually implement tail-call elimination, we need to use thunks. For this, we can use the `bind()` function that allows us to apply a method to one object with the `this` keyword assigned to another. Internally, it's the same as the `call` keyword, but it's chained to the method and returns a new bound function. The `bind()` function actually does partial application, though in a very limited way.

```
var factorial = function(n) {
  var _fact = function(x, n) {
    if (n == 0) {
      // base case
      return x;
    }
    else {
      // recursive case
      return _fact.bind(null, n*x, n-1);
    }
  }
  return trampoline(_fact.bind(null, 1, n));
}
```

But writing the `fact.bind(null, ...)` method is cumbersome and would confuse anybody reading the code. Instead, let's write our own function for creating thunks. There are a few things the `thunk()` function must do:

- `thunk()` function must emulate the `_fact.bind(null, n*x, n-1)` method that returns a non-evaluated function
- The `thunk()` function should enclose two more functions:
    - For processing the give function, and
    - For processing the function arguments that will be used when the given function is invoked

With that, we're ready to write the function. We only need a few lines of code to write it.

```
var thunk = function (fn) {
  return function() {
    var args = Array.prototype.slice.apply(arguments);
    return function() { return fn.apply(this, args); };
  };
};
```

Now we can use the `thunk()` function in our factorial algorithm like this:

```
var factorial = function(n) {
  var fact = function(x, n) {
    if (n == 0) {
      return x;
    }
    else {
      return thunk(fact)(n * x, n - 1);
    }
  }
  return trampoline(thunk(fact)(1, n));
}
```

But again, we can simplify it just a bit further by defining the `_fact()` function as a `thunk()` function. By defining the inner function as a `thunk()` function, we're relieved of having to use the `thunk()` function both inside the inner function definition and in the return statement.

```
var factorial = function(n) {
  var _fact = thunk(function(x, n) {
    if (n == 0) {
      // base case
      return x;
    }
    else {
      // recursive case
      return _fact(n * x, n - 1);
    }
  });
  return trampoline(_fact(1, n));
}
```

The result is beautiful. What seems like the function `_fact()` being recursively called for a tail-free recursion is almost transparently processed as an iteration!

Finally, let's see how the `trampoline()` and `thunk()` functions work with our more meaningful example of tree traversal. The following is a crude example of how a data tree could be traversed using trampolining and thunks:

```
var treeTraverse = function(trunk) {
  var _traverse = thunk(function(node) {
    node.doSomething();
    node.children.forEach(_traverse);
  }
  trampoline(_traverse(trunk));
}
```

We've solved the issue of tail recursion. But is there an even better way? What if we could simply convert the recursive function to a non-recursive function? Up next, we'll look at how to do just that.

The Y-combinator

The Y-combinator is one of those things in computer science that amaze even the deftest of programming masterminds. Its ability to automatically convert recursive functions to non-recursive functions is why Douglas Crockford calls it "one of the most strange and wonderful artifacts of computer science", and Sussman and Steele once said, "That this manages to work is truly remarkable".

So a truly-remarkable, wonderfully strange artifact of computer science that brings recursive functions to their knees must be massive and complex, right? No, not exactly. Its implementation in JavaScript is only nine, very odd, lines of code. They are as follows:

```
var Y = function(F) {
  return (function (f) {
    return f(f);
  } (function (f) {
    return F(function (x) {
      return f(f)(x);
    });
  }));
}
```

Here's how it works: it finds the "fixed point" of the function passed in as an argument. Fixed points offer another way to think about functions rather than recursion and iteration in the theory of computer programming. And it does this with only the use of anonymous function expressions, function applications, and variable references. Note that `Y` does not reference itself. In fact, all those functions are anonymous.

As you might have guessed, the Y-combinator came out of lambda calculus. It's actually derived with the help of another combinator called the U-combinator. Combinators are special higher-order functions that only use function application and earlier defined combinators to define a result from its input.

To demonstrate the Y-combinator, we'll again turn to the factorial problem, but we need to define the factorial function a little differently. Instead of writing a recursive function, we write a function that returns a function that is the mathematical definition of factorials. Then we can pass this into the Y-combinator.

```
var FactorialGen = function(factorial) {
  return (function(n) {
    if (n == 0) {
      // base case
      return 1;
    }
    else {
      // recursive case
      return n * factorial(n - 1);
    }
  });
};
Factorial = Y(FactorialGen);
Factorial(10); // 3628800
```

However, when we give it a significantly large number, the stack overflows just as if tail recursion without trampolining was used.

```
Factorial(23456); // RangeError: Maximum call stack size exceeded
```

But we can use trampolining with the Y-combinator as in the following:

```
var FactorialGen2 = function (factorial) {
  return function(n) {
    var factorial = thunk(function (x, n) {
      if (n == 0) {
        return x;
      }
      else {
        return factorial(n * x, n - 1);
      }
    });
    return trampoline(factorial(1, n));
  }
};

var Factorial2 = Y(FactorialGen2)
Factorial2(10); // 3628800
Factorial2(23456); // Infinity
```

We can also rearrange the Y-combinator to perform something called memoization.

Memoization

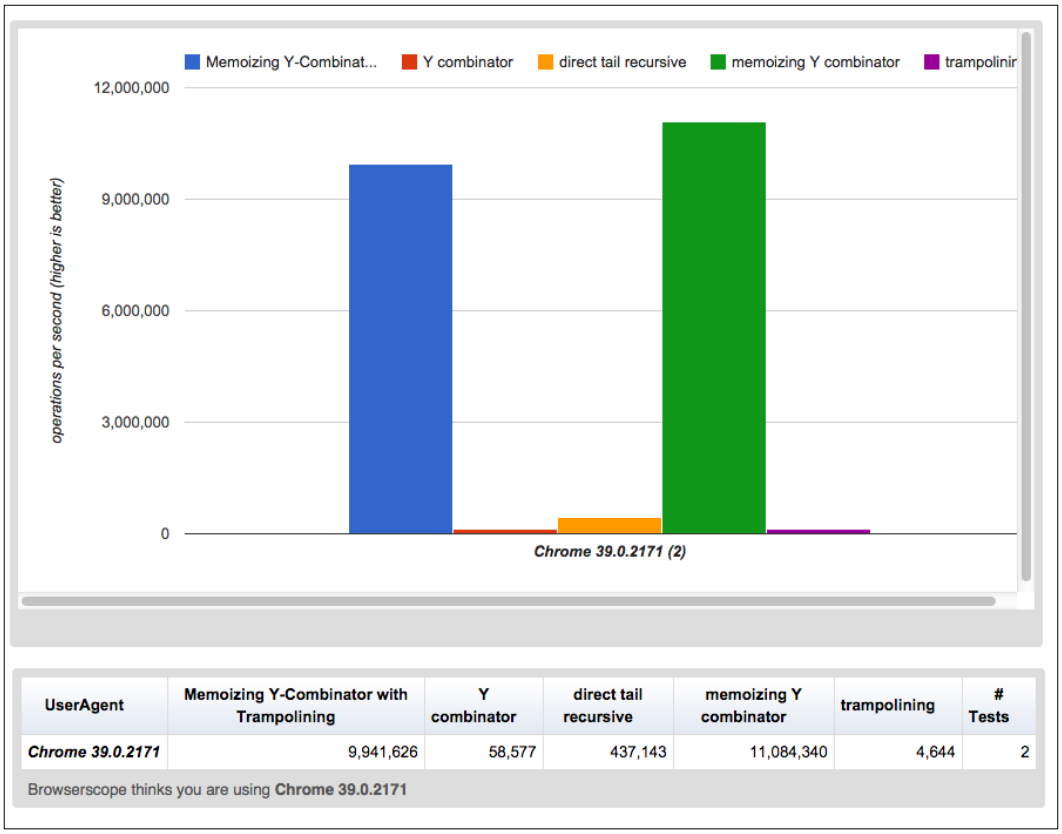
Memoization is the technique of storing the result of expensive function calls. When the function is later called with the same arguments, the stored result is returned rather than computing the result again.

Although the Y-combinator is much faster than recursion, it is still relatively slow. To speed it up, we can create a memoizing fixed-point combinator: a Y-like combinator that caches the results of intermediate function calls.

```
var Ymem = function(F, cache) {
  if (!cache) {
    cache = {} ; // Create a new cache.
  }
  return function(arg) {
    if (cache[arg]) {
      // Answer in cache
      return cache[arg] ;
    }
    // else compute the answer
    var answer = (F(function(n){
      return (Ymem(F,cache))(n);
    }))(arg); // Compute the answer.
    cache[arg] = answer; // Cache the answer.
    return answer;
  };
}
```

So how much faster is it? By using `http://jsperf.com/`, we can compare the performance.

The following results are with random numbers between 1 and 100. We can see that the memoizing Y-combinator is much, much faster. And adding trampolining to it does not slow it down by much. You can view the results and run the tests yourself at this URL: `http://jsperf.com/memoizing-y-combinator-vs-tail-call-optimization/7`.



UserAgent	Memoizing Y-Combinator with Trampolining	Y combinator	direct tail recursive	memoizing Y combinator	trampolining	# Tests
Chrome 39.0.2171	9,941,626	58,577	437,143	11,084,340	4,644	2

Browserscope thinks you are using **Chrome 39.0.2171**

The bottom line is: the most efficient and safest method of performing recursion in JavaScript is to use the memoizing Y-combinator with tail-call elimination via trampolining and thunks.

Variable scope

The scope of variables in JavaScript is not natural. In fact, sometimes it's downright counter-intuitive. They say that JavaScript programmers can be judged by how well they understand scope.

Scope resolutions

First, let's go over the different scope resolutions in JavaScript.

JavaScript uses scope chains to establish the scope of variables. When resolving a variable, it starts at the innermost scope and searches outwards.

Global scope

Variables, functions, and objects defined at this level are available to any code in the entire program. This is the outermost scope.

```
var x = 'hi';
function a() {
  console.log(x);
}
a(); // 'hi'
```

Local scope

Each function described has its own local scope. Any function defined within another function has a nested local scope that is linked to the outer function. Almost always, it's the position in the source that defines the scope.

```
var x = 'hi';
function a() {
  console.log(x);
}
function b() {
  var x = 'hello';
  console.log(x);
}
b(); // hello
a(); // hi
```

Local scope is only for functions and not for any expression statements (`if`, `for`, `while`, and so on), which is different from how most languages treat scope.

```
function c() {
  var y = 'greetings';
  if (true) {
    var y = 'guten tag';
  }
  console.log(y);
}

function d() {
  var y = 'greetings';
  function e() {
    var y = 'guten tag';
  }
```

```
  console.log(y)
}
c(); // 'guten tag'
d(); // 'greetings'
```

In functional programming, this isn't as much of a concern because functions are used more often and expression statements less often. For example:

```
function e(){
  var z = 'namaste';
  [1,2,3].foreach(function(n) {
    var z = 'aloha';
  }
  isTrue(function(){
    var z = 'good morning';
  });
  console.log(z);
}
e(); // 'namaste'
```

Object properties

Object properties have their own scope chains as well.

```
var x = 'hi';
var obj = function(){
  this.x = 'hola';
};
var foo = new obj();
console.log(foo.x); // 'hola'
foo.x = 'bonjour';
console.log(foo.x); // 'bonjour'
```

The object's prototype is further down the scope chain.

```
obj.prototype.x = 'greetings';
obj.prototype.y = 'konnichi ha';
var bar = new obj();
console.log(bar.x); // still prints 'hola'
console.log(bar.y); // 'konnichi ha'
```

This isn't even close to being comprehensive, but these three types of scope are enough to get started.

Closures

One problem with this scope structure is that it leaves no room for private variables. Consider the following code snippet:

```
var name = 'Ford Focus';
var year = '2006';
var millage = 123456;
function getMillage(){
  return millage;
}
function updateMillage(n) {
  millage = n;
}
```

These variables and functions are global, which means it would be too easy for code later down the program to accidentally overwrite them. One solution would be to encapsulate them into a function and call that function immediately after defining it.

```
var car = function(){
  var name = 'Ford Focus';
  var year = '2006';
  var millage = 123456;
  function getMillage(){
    return Millage;
  }
  function updateMillage(n) {
    millage = n;
  }
}();
```

Nothing is happening outside the function, so we ought to discard the function name by making it anonymous.

```
(function(){
  var name = 'Ford Focus';
  var year = '2006';
  var millage = 123456;
  function getMillage(){
    return millage;
  }
  function updateMillage(n) {
    millage = n;
  }
})();
```

To make the functions `getValue()` and `updateMillage()` available outside the anonymous function, we'll need to return them in an object literal as shown in the following code snippet:

```
var car = function(){
  var name = 'Ford Focus';
  var year = '2006';
  var millage = 123456;
  return {
    getMillage: function(){
      return millage;
    },
    updateMillage: function(n) {
      millage = n;
    }
  }
}();
console.log( car.getMillage() ); // works
console.log( car.updateMillage(n) ); // also works
console.log( car.millage ); // undefined
```

This gives us pseudo-private variables, but the problems don't stop there.
The following section explores more issues with variable scope in JavaScript.

Gotchas

Many variable scope nuances can be found throughout JavaScript. The following is by no means a comprehensive list, but it covers the most common cases:

- The following will output 4, not 'undefined' as one would expect:

  ```
  for (var n = 4; false; ) { } console.log(n);
  ```

  This is due to the fact that, in JavaScript, variable definition happens at the beginning of the corresponding scope, not just when it is declared.

- If you define a variable in the outer scope, and then have an `if` statement define a variable inside the function with the same name, even if that `if` branch isn't reached, it is redefined. An example:

  ```
  var x = 1;
  function foo() {
    if (false) {
      var x = 2;
    }
    return x;
  ```

```
}
foo(); // Return value: 'undefined', expected return value:
2
```

Again, this is caused by moving the variable definition at the beginning of the scope with the `undefined` value.

- In the browser, global variables are really stored in the `window` object.

```
window.a = 19;
console.log(a); // Output: 19
```

`a` in the global scope means `a` as an attribute of the current context, so `a===this.a` and `window` object in a browser act as an equivalent of the `this` keyword in the global scope.

The first two examples are a result of a feature of JavaScript known as hoisting, which will be a critical concept in the next section about writing functions.

Function declarations versus function expressions versus the function constructor

What is the difference between these three statements?

```
function foo(n){ return n; }
var foo = function(n){ return n; };
var foo = new Function('n', 'return n');
```

At first glance, they're merely different ways to write the same function. But there's a little more going on here. And if we're to take full advantage of functions in JavaScript in order to manipulate them into a functional programming style, then we'd better be able to get this right. If there is a better way to do something in computer programming, then that one way should be the only way.

Function declarations

Function declarations, sometimes called function statements, define a function by using the `function` keyword.

```
function foo(n) {
  return n;
}
```

Functions that are declared with this syntax are *hoisted* to the top of the current scope. What this actually means is that, even if the function is defined several lines down, JavaScript knows about it and can use it earlier in the scope. For example, the following will correctly print 6 to the console:

```
foo(2,3);
function foo(n, m) {
  console.log(n*m);
}
```

Function expressions

Named functions can also be defined as an expression by defining an anonymous function and assigning it to a variable.

```
var bar = function(n, m) {
  console.log(n*m);
};
```

They are not hoisted like function declarations are. This is because, while function declarations are hoisted, variable declarations are not. For example, this will not work and will throw an error:

```
bar(2,3);
var bar = function(n, m) {
  console.log(n*m);
};
```

In functional programming, we're going to want to use function expressions so we can treat the functions like variables, making them available to be used as callbacks and arguments to higher-order functions such as `map()` functions. Defining functions as expressions makes it more obvious that they're variables assigned to a function. Also, if we're going to write functions in one style, we should write all functions in that style for the sake of consistency and clarity.

The function constructor

JavaScript actually has a third way to create functions: with the `Function()` constructor. Just like function expressions, functions defined with the `Function()` constructor are not hoisted.

```
var func = new Function('n','m','return n+m');
func(2,3); // returns 5
```

But the `Function()` constructor is not only confusing, it is also highly dangerous. No syntax correction can happen, no optimization is possible. It's far easier, safer, and less confusing to write the same function as follows:

```
var func = function(n,m){return n+m};
func(2,3); // returns 5
```

Unpredictable behavior

So the difference is that function declarations are hoisted while function expressions are not. This can cause unexpected things to happen. Consider the following:

```
function foo() {
  return 'hi';
}
console.log(foo());
function foo() {
  return 'hello';
}
```

What's actually printed to the console is `hello`. This is due to the fact that the second definition of the `foo()` function is hoisted to the top, making it the definition that is actually used by the JavaScript interpreter.

While at first this may not seem like a critical difference, in functional programming this can cause mayhem. Consider the following code snippet:

```
if (true) {
  function foo(){console.log('one')};
}
else {
  function foo(){console.log('two')};
}
foo();
```

When the `foo()` function is called, `two` is printed to the console, not `one`!

Finally, there is a way to combine both function expressions and declarations. It works as follows:

```
var foo = function bar(){ console.log('hi'); };
foo(); // 'hi'
bar(); // Error: bar is not defined
```

It makes very little sense to use this method because the name used in the declaration (the `bar()` function in the preceding example) is not available outside the function and causes confusion. It would only be appropriate for recursion, for example:

```
var foo = function factorial(n) {
  if (n == 0) {
    return 1;
  }
else {
    return n * factorial(n-1);
  }
};
foo(5);
```

Summary

JavaScript has been called the "assembly language of the web," because it's as ubiquitous and unavoidable as x86 assembly. It's the only language that runs on all browsers. It's also flawed, yet referring to it as a low-level language is missing the mark.

Instead, think of JavaScript as the raw coffee beans of the web. Sure, some of the beans are damaged and a few are rotten. But if the good ones are selected, roasted, and brewed by a skilled barista, the beans can be transformed into a brilliant jamocha that cannot be had just once and forgotten. It's consumption becomes a daily custom, life without it would be static, harder to perform, and much less exciting. Some even prefer to enhance the brew with plug-ins and add-ons such as cream, sugar, and cocoa, which complement it very well.

One of JavaScript's biggest critics, Douglas Crawford, was quoted as saying "There are certainly a lot of people who refuse to consider the possibility that JavaScript got anything right. I used to be one of those guys. But now I continue to be amazed by the brilliance that is in there".

JavaScript turned out to be pretty awesome.

7
Functional and Object-oriented Programming in JavaScript

You will often hear that JavaScript is a blank language, where blank is either object-oriented, functional, or general-purpose. This book has focused on JavaScript as a functional language and has gone to great lengths to prove that it is. But the truth is that JavaScript is a general-purpose language, meaning it's fully capable of multiple programming styles. Like Python and F#, JavaScript is multi-paradigm. But unlike those languages, JavaScript's OOP side is prototype-based while most other general-purpose languages are class-based.

In this final chapter, we will relate both functional and object-oriented programming to JavaScript, and see how the two paradigms can complement each other and coexist side-by-side. In this chapter the following topics will be covered:

- How can JavaScript be both functional and OOP?
- JavaScript's OOP – using prototypes
- How to mix functional and OOP in JavaScript
- Functional inheritance
- Functional mixins

Better code is the goal. Functional and object-oriented programming are just means to this end.

JavaScript – the multi-paradigm language

If object-oriented programming means treating all variables as objects, and functional programming means treating all functions as variables, then can't functions be treated like objects? In JavaScript, they can.

But saying that functional programming means treating functions as variables is somewhat inaccurate. A better way to put it is: functional programming means treating everything as a value, especially functions.

A better way still to describe functional programming may be to call it declarative. Independent of the imperative branch of programming styles, *declarative programming* expresses the logic of computation required to solve the problem. The computer is told what the problem is rather than the procedure for how to solve it.

Meanwhile, object-oriented programming is derived from the imperative programming style: the computer is given step-by-step instructions for how to solve the problem. OOP mandates that the instructions for computation (methods) and the data they work on (member variables) be organized into units called objects. The only way to access that data is through the object's methods.

So how can these two styles be integrated together?

- The code inside the object's methods is typically written in an imperative style. But what if it was in a functional style? After all, OOP doesn't exclude immutable data and higher-order functions.
- Perhaps a purer way to mix the two would be to treat objects both as functions and as traditional, class-based objects at the same time.
- Maybe we can simply include several ideas from functional programming—such as promises and recursion—into our object-oriented application.
- OOP covers topics such as encapsulation, polymorphism, and abstraction. So does functional programming, it just goes about it in a different way. So maybe we can include several ideas from object-oriented programming in our functional-oriented application.

The point is: OOP and FP can be mixed together and there are several ways to do it. They're not exclusive of each other.

JavaScript's object-oriented implementation – using prototypes

JavaScript is a class-less language. That's not to mean it is less fashionable or more blue-collar than other computer languages; class-less means it doesn't have a class structure in the same way that object-oriented languages do. Instead, it uses prototypes for inheritance.

Although this may be baffling to programmers with backgrounds in C++ and Java, prototype-based inheritance can be much more expressive than traditional inheritance. The following is a brief comparison between the differences between C++ and JavaScript:

C++	**JavaScript**
Strongly typed	Loosely typed
Static	Dynamic
Class-based	Prototype-based
Classes	Functions
Constructors	Functions
Methods	Functions

Inheritance

Before we go much further, let's make sure we fully understand the concept of inheritance in object-oriented programming. Class-based inheritance is demonstrated in the following pseudo-code:

```
class Polygon {
  int numSides;
  function init(n) {
    numSides = n;
  }
}
class Rectangle inherits Polygon {
  int width;
  int length;
  function init(w, l) {
    numSides = 4;
    width = w;
    length = l;
  }
```

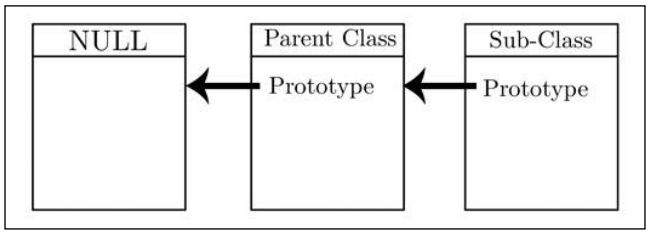
```
    function getArea() {
      return w * l;
    }
  }
  class Square inherits Rectangle {
    function init(s) {
      numSides = 4;
      width = s;
      length = s;
    }
  }
```

The `Polygon` class is the parent class the other classes inherit from. It defines just one member variable, the number of sides, which is set in the `init()` function. The `Rectangle` subclass inherits from the `Polygon` class and adds two more member variables, `length` and `width`, and a method, `getArea()`. It doesn't need to define the `numSides` variable because it was already defined by the class it inherits from, and it also overrides the `init()` function. The `Square` class carries on this chain of inheritance even further by inheriting from the `Rectangle` class for its `getArea()` method. By simply overriding the `init()` function again such that the length and width are the same, the `getArea()` function can remain unchanged and less code needs to be written.

In a traditional OOP language, this is what inheritance is all about. If we wanted to add a color property to all the objects, all we would have to do is add it to the `Polygon` object without having to modify any of the objects that inherit from it.

JavaScript's prototype chain

Inheritance in JavaScript comes down to prototypes. Each object has an internal property known as its prototype, which is a link to another object. That object has a prototype of its own. This pattern can repeat until an object is reached that has `undefined` as its prototype. This is known as the prototype chain, and it's how inheritance works in JavaScript. The following diagram explain the inheritance in JavaScirpt:



When running a search for an object's function definition, JavaScript "walks" the prototype chain until it finds the first definition of a function with the right name. Therefore, overriding it is as simple as providing a new definition on the prototype of the subclass.

Inheritance in JavaScript and the Object.create() method

Just as there are many ways to create objects in JavaScript, there are also many ways to replicate class-based, classical inheritance. But the one preferred way to do it is with the `Object.create()` method.

```
var Polygon = function(n) {
  this.numSides = n;
}

var Rectangle = function(w, l) {
  this.width = w;
  this.length = l;
}

// the Rectangle's prototype is redefined with Object.create
Rectangle.prototype = Object.create(Polygon.prototype);

// it's important to now restore the constructor attribute
// otherwise it stays linked to the Polygon
Rectangle.prototype.constructor = Rectangle;

// now we can continue to define the Rectangle class
Rectangle.prototype.numSides = 4;
Rectangle.prototype.getArea = function() {
  return this.width * this.length;
}

var Square = function(w) {
  this.width = w;
  this.length = w;
}
Square.prototype = Object.create(Rectangle.prototype);
Square.prototype.constructor = Square;

var s = new Square(5);
console.log( s.getArea() ); // 25
```

This syntax may seem unusual to many but, with a little practice, it will become familiar. The `prototype` keyword must be used to gain access to the internal property, `[[Prototype]]`, which all objects have. The `Object.create()` method declares a new object with a specified object for its prototype to inherit from. In this way, classical inheritance can be achieved in JavaScript.



The `Object.create()` method was introduced in ECMAScript 5.1 in 2011, and it was billed as the new and preferred way to create objects. This was just one of many attempts to integrate inheritance into JavaScript. Thankfully, this method works pretty well.

We saw this structure of inheritance when building the `Maybe` classes in *Chapter 5*, *Category Theory*. Here are the `Maybe`, `None`, and `Just` classes, which inherit from each other just like the preceding example.

```
var Maybe = function(){};

var None = function(){};
None.prototype = Object.create(Maybe.prototype);
None.prototype.constructor = None;
None.prototype.toString = function(){return 'None';};

var Just = function(x){this.x = x;};
Just.prototype = Object.create(Maybe.prototype);
Just.prototype.constructor = Just;
Just.prototype.toString = function(){return "Just "+this.x;};
```

This shows that class inheritance in JavaScript can be an enabler of functional programming.

A common mistake is to pass a constructor into `Object.create()` instead of a `prototype` object. This problem is compounded by the fact that an error will not be thrown until the subclass tries to use an inherited member function.

```
Foo.prototype = Object.create(Parent.prototype); // correct
Bar.prototype = Object.create(Parent); // incorrect
Bar.inheritedMethod(); // Error: function is undefined
```

The function won't be found if the `inheritedMethod()` method has been attached to the `Foo.prototype` class. If the `inheritedMethod()` method is attached directly to the instance with `this.inheritedMethod = function(){...}` in the `Bar` constructor, then this use of `Parent` as an argument of `Object.create()` could be correct.

Mixing functional and object-oriented programming in JavaScript

Object-oriented programming has been the dominant programming paradigm for several decades. It is taught in Computer Science 101 classes around the world, while functional programming is not. It is what software architects use to design applications, while functional programming is not. And it makes sense too: OOP makes it easy to conceptualize abstract ideas. It makes it easier to write code.

So, unless you can convince your boss that the application needs to be all functional, we're going to be using functional programming in an object-oriented world. This section will explore ways to do this.

Functional inheritance

Perhaps the most accessible way to apply functional programming to JavaScript applications is to use a mostly functional style within OOP principles, such as inheritance.

To explore how this might work, let's build a simple application that calculates the price of a product. First, we'll need some product classes:

```
var Shirt = function(size) {
  this.size = size;
};

var TShirt = function(size) {
  this.size = size;
};
TShirt.prototype = Object.create(Shirt.prototype);
TShirt.prototype.constructor = TShirt;
TShirt.prototype.getPrice = function(){
  if (this.size == 'small') {
    return 5;
  }
  else {
    return 10;
  }
}

var ExpensiveShirt = function(size) {
  this.size = size;
}
```

```
ExpensiveShirt.prototype = Object.create(Shirt.prototype);
ExpensiveShirt.prototype.constructor = ExpensiveShirt;
ExpensiveShirt.prototype.getPrice = function() {
  if (this.size == 'small') {
    return 20;
  }
  else {
    return 30;
  }
}
```

We can then organize them within a `Store` class as follows:

```
var Store = function(products) {
  this.products = products;
}
Store.prototype.calculateTotal = function(){
  return this.products.reduce(function(sum,product) {
    return sum + product.getPrice();
  }, 10) * TAX; // start with $10 markup, times global TAX var
};

var TAX = 1.08;
var p1 = new TShirt('small');
var p2 = new ExpensiveShirt('large');
var s = new Store([p1,p2]);
console.log(s.calculateTotal()); // Output: 35
```

The `calculateTotal()` method uses the array's `reduce()` function to cleanly sum together the prices of the products.

This works just fine, but what if we need a dynamic way to calculate the markup value? For this, we can turn to a concept called **Strategy Pattern**.

Strategy Pattern

Strategy Pattern is a method for defining a family of interchangeable algorithms. It is used by OOP programmers to manipulate behavior at runtime, but it is based on a few functional programming principles:

- Separation of logic and data
- Composition of functions
- Functions as first-class objects

And a couple of OOP principles as well:

- Encapsulation
- Inheritance

In our example application for calculating product cost, explained previously, let's say we want to give preferential treatment to certain customers, and that the markup will have to be adjusted to reflect this.

So let's create some customer classes:

```
var Customer = function(){};
Customer.prototype.calculateTotal = function(products) {
  return products.reduce(function(total, product) {
    return total + product.getPrice();
  }, 10) * TAX;
};

var RepeatCustomer = function(){};
RepeatCustomer.prototype = Object.create(Customer.prototype);
RepeatCustomer.prototype.constructor = RepeatCustomer;
RepeatCustomer.prototype.calculateTotal = function(products) {
  return products.reduce(function(total, product) {
    return total + product.getPrice();
  }, 5) * TAX;
};

var TaxExemptCustomer = function(){};
TaxExemptCustomer.prototype = Object.create(Customer.prototype);
TaxExemptCustomer.prototype.constructor = TaxExemptCustomer;
TaxExemptCustomer.prototype.calculateTotal = function(products) {
  return products.reduce(function(total, product) {
    return total + product.getPrice();
  }, 10);
};
```

Each `Customer` class encapsulates the algorithm. Now we just need the `Store` class to call the `Customer` class's `calculateTotal()` method.

```
var Store = function(products) {
  this.products = products;
  this.customer = new Customer();
  // bonus exercise: use Maybes from Chapter 5 instead of a
  default customer instance
}
```

```
Store.prototype.setCustomer = function(customer) {
  this.customer = customer;
}
Store.prototype.getTotal = function(){
  return this.customer.calculateTotal(this.products);
};

var p1 = new TShirt('small');
var p2 = new ExpensiveShirt('large');
var s = new Store([p1,p2]);
var c = new TaxExemptCustomer();
s.setCustomer(c);
s.getTotal(); // Output: 45
```

The `Customer` classes do the calculating, the `Product` classes hold the data (the prices), and the `Store` class maintains the context. This achieves a very high level of cohesion and a very good mixture of object-oriented programming and functional programming. JavaScript's high level of expressiveness makes this possible and quite easy.

Mixins

In a nutshell, mixins are classes that can allow other classes to use their methods. The methods are intended to be used solely by other classes, and the `mixin` class itself is never to be instantiated. This helps to avoid inheritance ambiguity. And they're a great means of mixing functional programming with object-oriented programming.

Mixins are implemented differently in each language. Thanks to JavaScript's flexibility and expressiveness, mixins are implemented as objects with only methods. While they can be defined as function objects (that is, `var mixin = function(){...};`), it would be better for the structural discipline of the code to define them as object literals (that is, `var mixin = {...};`). This will help us to distinguish between classes and mixins. After all, mixins should be treated as processes, not objects.

Let's start with declaring some mixins. We'll extend our `Store` application from the previous section, using mixins to expand on the classes.

```
var small = {
  getPrice: function() {
    return this.basePrice + 6;
  },
  getDimensions: function() {
    return [44,63]
  }
```

```
}
var large = {
  getPrice: function() {
    return this.basePrice + 10;
  },
  getDimensions: function() {
    return [64,83]
  }
};
```

We're not limited to just this. Many more mixins can be added, like colors or fabric material. We'll have to rewrite our `Shirt` classes a little bit, as shown in the following code snippet:

```
var Shirt = function() {
  this.basePrice = 1;
};
Shirt.getPrice = function(){
  return this.basePrice;
}
var TShirt = function() {
  this.basePrice = 5;
};
TShirt.prototype = Object.create(Shirt.prototype);
TShirt..prototype.constructor = TShirt;
```

Now we're ready to use mixins.

Classical mixins

You're probably wondering just how these mixins get mixed with the classes. The classical way to do this is by copying the mixin's functions into the receiving object. This can be done with the following extension to the `Shirt` prototype:

```
Shirt.prototype.addMixin = function (mixin) {
  for (var prop in mixin) {
    if (mixin.hasOwnProperty(prop)) {
      this.prototype[prop] = mixin[prop];
    }
  }
};
```

And now the mixins can be added as follows:

```
TShirt.addMixin(small);
var p1 = new TShirt();
console.log( p1.getPrice() ); // Output: 11

TShirt.addMixin(large);
var p2 = new TShirt();
console.log( p2.getPrice() ); // Output: 15
```

However, there is a major problem. When the price of `p1` is calculated again, it comes back as `15`, the price of a large item. It should be the value for a small one!

```
console.log( p1.getPrice() ); // Output: 15
```

The problem is that the `Shirt` object's `prototype.getPrice()` method is getting rewritten every time a mixin is added to it; this is not very functional at all and not what we want.

Functional mixins

There's another way to use mixins, one that is more aligned with functional programming.

Instead of copying the methods of the mixin to the target object, we need to create a new object that is a clone of the target object with the mixin's methods added in. The object must be cloned first, and this is achieved by creating a new object that inherits from it. We'll call this variation `plusMixin`.

```
Shirt.prototype.plusMixin = function(mixin) {
  // create a new object that inherits from the old
  var newObj = this;
  newObj.prototype = Object.create(this.prototype);
  for (var prop in mixin) {
    if (mixin.hasOwnProperty(prop)) {
      newObj.prototype[prop] = mixin[prop];
    }
  }
  return newObj;
};

var SmallTShirt = Tshirt.plusMixin(small); // creates a new class
var smallT = new SmallTShirt();
console.log( smallT.getPrice() );  // Output: 11
```

```
var LargeTShirt = Tshirt.plusMixin(large);
var largeT = new LargeTShirt();
console.log( largeT.getPrice() ); // Output: 15
console.log( smallT.getPrice() ); // Output: 11 (not effected by 2nd
mixin call)
```

Here comes the fun part! Now we can get really functional with the mixins. We can create every possible combination of products and mixins.

```
// in the real world there would be way more products and mixins!
var productClasses = [ExpensiveShirt, Tshirt];
var mixins = [small, medium, large];

// mix them all together
products = productClasses.reduce(function(previous, current) {
  var newProduct = mixins.map(function(mxn) {
    var mixedClass = current.plusMixin(mxn);
    var temp = new mixedClass();
    return temp;
  });
  return previous.concat(newProduct);
},[]);
products.forEach(function(o){console.log(o.getPrice())});
```

To make it more object-oriented, we can rewrite the `Store` object with this functionality. We'll also add a display function to the `Store` object, not the products, to keep the interface logic and the data separated.

```
// the store
var Store = function() {
  productClasses = [ExpensiveShirt, TShirt];
  productMixins = [small, medium, large];
  this.products = productClasses.reduce(function(previous,
  current) {
    var newObjs = productMixins.map(function(mxn) {
      var mixedClass = current.plusMixin(mxn);
      var temp = new mixedClass();
      return temp;
    });
    return previous.concat(newObjs);
  },[]);
}
Store.prototype.displayProducts = function(){
  this.products.forEach(function(p) {
    $('ul#products').append('<li>'+p.getTitle()+':
    $'+p.getPrice()+'</li>');
  });
}
```

And all we have to do is create a `Store` object and call its `displayProducts()` method to generate a list of products and prices!

```
<ul id="products">
  <li>small premium shirt: $16</li>
  <li>medium premium shirt: $18</li>
  <li>large premium shirt: $20</li>
  <li>small t-shirt: $11</li>
  <li>medium t-shirt: $13</li>
  <li>large t-shirt: $15</li>
</ul>
```

These lines need to be added to the `product` classes and mixins to get the preceding output to work:

```
Shirt.prototype.title = 'shirt';
TShirt.prototype.title = 't-shirt';
ExpensiveShirt.prototype.title = 'premium shirt';

// then the mixins got the extra 'getTitle' function:
var small = {
  ...
  getTitle: function() {
    return 'small ' + this.title; // small or medium or large
  }
}
```

And, just like that, we have an e-commerce application that is highly modular and extendable. New shirt styles can be added absurdly easily—just define a new `Shirt` subclass and add to it the `Store` class's array `product` classes. Mixins are added in just the same way. So now when our boss says, "Hey, we have a new type of shirt and a coat, each available in the standard colors, and we need them added to the website before you go home today", we can rest assured that we'll not be staying late!

Summary

JavaScript has a high level of expressiveness. This makes it possible to mix functional and object-oriented programming. Modern JavaScript is not solely OOP or functional—it is a mixture of the two. Concepts such as Strategy Pattern and mixins are perfect for JavaScript's prototype structure, and they help to prove that today's best practices in JavaScript share equal amounts of functional programming and object-oriented programming.

If you were to take away only one thing from this book, I would want it to be how to apply functional programming techniques to real-world applications. And this chapter showed you how to do exactly that.

A

Common Functions for Functional Programming in JavaScript

This Appendix covers common functions for functional programming in JavaScript:

- Array Functions:

```
var flatten = function(arrays) {
  return arrays.reduce( function(p,n){
    return p.concat(n);
  });
};

var invert = function(arr) {
  return arr.map(function(x, i, a) {
    return a[a.length - (i+1)];
  });
};
```

- Binding Functions:

```
var bind = Function.prototype.call.bind(Function.prototype.bind);
var call = bind(Function.prototype.call, Function.prototype.call);
var apply = bind(Function.prototype.call,
Function.prototype.apply);
```

- Category Theory:

```
var checkTypes = function( typeSafeties ) {
  arrayOf(func)(arr(typeSafeties));
```

```
  var argLength = typeSafeties.length;
  return function(args) {
    arr(args);
    if (args.length != argLength) {
      throw new TypeError('Expected '+ argLength + '
      arguments');
    }
    var results = [];
    for (var i=0; i<argLength; i++) {
      results[i] = typeSafeties[i](args[i]);
    }
    return results;
  };
};

var homoMorph = function( /* arg1, arg2, ..., argN, output */ ) {
  var before =
  checkTypes(arrayOf(func)(Array.prototype.slice.call
  (arguments, 0, arguments.length-1)));
  var after = func(arguments[arguments.length-1])
  return function(middle) {
    return function(args) {
      return after(middle.apply(this,
      before([].slice.apply(arguments))));
    };
  };
};
```

- Composition:

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

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

- Currying:

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

- Functors:

```
// map :: (a -> b) -> [a] -> [b]
var map = function(f, a) {
  return arr(a).map(func(f));
}

// strmap :: (str -> str) -> str -> str
var strmap = function(f, s) {
  return str(s).split('').map(func(f)).join('');
}

// fcompose :: (a -> b)* -> (a -> b)
var fcompose = function() {
  var funcs = arrayOf(func)(arguments);
  return function() {
    var argsOfFuncs = arguments;
    for (var i = funcs.length; i > 0; i -= 1) {
      argsOfFuncs  = [funcs[i].apply(this, args)];
    }
    return args[0];
  };
};
```

- Lenses:

```
var lens = function(get, set) {
  var f = function (a) {return get(a)};
  f.get = function (a) {return get(a)};
  f.set = set;
  f.mod = function (f, a) {return set(a, f(get(a)))};
  return f;
};

// usage:
var first = lens(
  function (a) { return arr(a)[0]; }, // get
  function (a, b) { return [b].concat(arr(a).slice(1)); } // set
);
```

- Maybes:

```
var Maybe = function(){};
Maybe.prototype.orElse = function(y) {
  if (this instanceof Just) {
    return this.x;
  }
  else {
    return y;
  }
};

var None = function(){};
None.prototype = Object.create(Maybe.prototype);
None.prototype.toString = function(){return 'None';};
var none = function(){return new None()};
// and the Just instance, a wrapper for an object with a
value
var Just = function(x){return this.x = x;};
Just.prototype = Object.create(Maybe.prototype);
Just.prototype.toString = function(){return "Just
"+this.x;};
var just = function(x) {return new Just(x)};
var maybe = function(m){
  if (m instanceof None) {
    return m;
  }
  else if (m instanceof Just) {
    return just(m.x);
  }
```

```
  else {
    throw new TypeError("Error: Just or None expected, " +
    m.toString() + " given.");
  }
};

var maybeOf = function(f){
  return function(m) {
    if (m instanceof None) {
      return m;
    }
    else if (m instanceof Just) {
      return just(f(m.x));
    }
    else {
      throw new TypeError("Error: Just or None expected, "
      + m.toString() + " given.");
    }
  };
};
```

- Mixins:

```
Object.prototype.plusMixin = function(mixin) {
  var newObj = this;
  newObj.prototype = Object.create(this.prototype);
  newObj.prototype.constructor = newObj;
  for (var prop in mixin) {
    if (mixin.hasOwnProperty(prop)) {
      newObj.prototype[prop] = mixin[prop];
    }
  }
  return newObj;
};
```

- Partial Application:

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

Function.prototype.partialApply = function(){
  var func = this;
  var args = Array.prototype.slice.call(arguments);
```

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

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

- Trampolining:

```
var trampoline = function(f) {
  while (f && f instanceof Function) {
    f = f.apply(f.context, f.args);
  }
  return f;
};

var thunk = function (fn) {
  return function() {
    var args = Array.prototype.slice.apply(arguments);
    return function() { return fn.apply(this, args); };
  };
};
```

- Type Safeties:

```
var typeOf = function(type) {
  return function(x) {
    if (typeof x === type) {
      return x;
    }
    else {
      throw new TypeError("Error: "+type+" expected,
      "+typeof x+" given.");
    }
  };
```

```
};

var str = typeOf('string'),
  num = typeOf('number'),
  func = typeOf('function'),
  bool = typeOf('boolean');

var objectTypeOf = function(name) {
  return function(o) {
    if (Object.prototype.toString.call(o) === "[object
    "+name+"]") {
      return o;
    }
    else {
      throw new TypeError("Error: '+name+' expected,
      something else given.");
    }
  };
};
var obj = objectTypeOf('Object');
var arr = objectTypeOf('Array');
var date = objectTypeOf('Date');
var div = objectTypeOf('HTMLDivElement');

// arrayOf :: (a -> b) -> ([a] -> [b])
var arrayOf = function(f) {
  return function(a) {
    return map(func(f), arr(a));
  }
};
```

- Y-combinator:

```
var Y = function(F) {
  return (function (f) {
    return f(f);
  }(function (f) {
    return F(function (x) {
      return f(f)(x);
    });
  }));
};

// Memoizing Y-Combinator:
var Ymem = function(F, cache) {
```

```
    if (!cache) {
      cache = {} ; // Create a new cache.
    }
    return function(arg) {
      if (cache[arg]) {
        // Answer in cache
        return cache[arg] ;
      }
      // else compute the answer
      var answer = (F(function(n){
        return (Ymem(F,cache))(n);
      }))(arg); // Compute the answer.
      cache[arg] = answer; // Cache the answer.
      return answer;
    };
  };
```

B
Glossary of Terms

This appendix covers some of the important terms that are used in this book:

- **Anonymous function**: A function that has no name and is not bound to any variables. It is also known as a Lambda Expression.
- **Callback**: A function that can be passed to another function to be used in a later event.
- **Category**: In terms of Category Theory, a category is a collection of objects of the same type. In JavaScript, a category can be an array or object that contains objects that are all explicitly declared as numbers, strings, Booleans, dates, objects, and so on.
- **Category Theory**: A concept that organizes mathematical structures into collections of objects and operations on those objects. The data types and functions used in computer programs form the categories used in this book.
- **Closure**: An environment such that functions defined within it can access local variables that are not available outside it.
- **Coupling**: The degree to which each program module relies on each of the other modules. Functional programming reduces the amount of coupling within a program.
- **Currying**: The process of transforming a function with many arguments into a function with one argument that returns another function that can take more arguments, as needed. Formally, a function with *N* arguments can be transformed into a function chain of *N* functions, each with only one argument.
- **Declarative programming**: A programming style that expresses the computational logic required to solve the problem. The computer is told what the problem is rather than the procedure required to solve it.

- **Endofunctor**: A functor that maps a category to itself.
- **Function composition**: The process of combining many functions into one function. The result of each function is passed as an argument to the next, and the result of the last function is the result of the whole composition.
- **Functional language**: A computer language that facilitates functional programming.
- **Functional programming**: A declarative programming paradigm that focuses on treating functions as mathematical expressions and avoids mutable data and changes in state.
- **Functional reactive programming**: A style of functional programming that focuses on reactive elements and variables that change over time in response to events.
- **Functor**: A mapping between categories.
- **Higher-order function**: A function that takes either one or more functions as input, and returns a function as its output.
- **Inheritance**: An object-oriented programming capability that allows one class to inherit member variables and methods from another class.
- **Lambda expressions**: See Anonymous function.
- **Lazy evaluation**: A computer language evaluation strategy that delays the evaluation of an expression until its value is needed. The opposite of this strategy is called eager evaluation or greedy evaluation. Lazy evaluation is also known as call by need.
- **Library**: A set of objects and functions that have a well-defined interface that allows a third-party program to invoke their behavior.
- **Memoization**: The technique of storing the results of expensive function calls. When the function is called later with the same arguments, the stored result is returned rather than computing the result again.
- **Method chain**: A pattern in which many methods are invoked side by side by directly passing the output of one method to the input of the next. This avoids the need to assign the intermediary values to temporary variables.
- **Mixin**: An object that can allow other objects to use its methods. The methods are intended to be used solely by other objects, and the mixin object itself is never to be instantiated.
- **Modularity**: The degree to which a program can be broken down into independent modules of code. Functional programming increases the modularity of programs.
- **Monad**: A structure that provides the encapsulation required by functors.

- **Morphism**: A pure function that only works on a certain category and always returns the same output when given a specific set of inputs. Homomorphic operations are restricted to a single category, while polymorphic operations can operate on multiple categories.
- **Partial application**: The process of binding values to one or more arguments of a function. It returns a partially applied function, which in turn accepts the remaining, unbound arguments.
- **Polyfill**: A function used to augment prototypes with new functions. It allows us to call our new functions as methods of the previous function.
- **Pure function**: A function whose output value depends only on the arguments that are the input to the function. Thus, calling a function, *f*, twice with the same value of an argument, *x*, will produce the same result, *f(x)*,every time.
- **Recursive function**: A function that calls itself. Such functions depend on solutions to smaller instances of the same problem to compute the solution to the larger problem. Like iteration, recursion is another way to repeatedly call the same block of code. But, unlike iteration, recursion requires that the code block define the case in which the repeating code calls should terminate, known as the base case.
- **Reusability**: The degree to which a block of code, usually a function in JavaScript, can be reused in other parts of the same program or in other programs.
- **Self-invoking function**: An anonymous function that is invoked immediately after it has been defined. In JavaScript, this is achieved by placing a pair of parentheses after the function expression.
- **Strategy pattern**: A method used to define a family of interchangeable algorithms.
- **Tail recursion**: A stack-based implementation of recursion. For every recursive call, there is a new frame in the stack.
- **Toolkit**: A small software library that provides a set of functions for the programmer to use. Compared to a library, a toolkit is simpler and requires less coupling with the program that invokes it.
- **Trampolining**: A strategy for recursion that provides tail-call elimination in programming languages that do not provide this feature, such as JavaScript.
- **Y-combinator**: A fixed-point combinator in Lambda calculus that eliminates explicit recursion. When it is given as input to a function that returns a recursive function, the Y-combinator returns the fixed point of that function, which is the transformation from the recursive function to a non-recursive function.

Bibliography

This learning path has been prepared for you to show how functional programming when combined with other techniques makes JavaScript programming more efficient. It comprises of the following Packt products:

- *Mastering JavaScript*, *Ved Antani*
- *Mastering JavaScript Design Patterns - Second Edition*, *Simon Timms*
- *Functional Programming in JavaScript*, *Dan Mantyla*