JavaScript-Mancy

Mastering the Arcane Art of Writing Awesome JavaScript

Jaime González García

JavaScript-mancy

Mastering the Arcane Art of Writing Awesome JavaScript for C# Developers

Jaime González García

This book is for sale at http://leanpub.com/javascriptmancy-mastering-arcane-art-of-writing-awesome-javascript-for-csharp-developers

This version was published on 2019-07-28

Tweet This Book!

Please help Jaime González García by spreading the word about this book on Twitter!

The suggested tweet for this book is:

Master the Arcane Art Of Writing Awesome JavaScript for C# Developers!

The suggested hashtag for this book is #javascriptmancy.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#javascriptmancy

To my beautiful wife Malin and my beloved son Teo

Contents

Prelude

It was during the second age
that the great founder of our order Branden Iech,

first stumbled upon the arcane REPL,
and learnt how to bend the fabric of existence to his very will,

then was that he discovered
there was a mechanism to alter the threads
being woven into The Pattern,

then that we started experiencing the magic of JavaScript

> - Irec Oliett,
> The Origins of JavaScript-Mancy
> Guardian of Chronicles, 7th Age

Imagine… imagine you lived in a world were you could use JavaScript to change the universe around you, to tamper with the threads that compose reality and do anything that you can imagine. Well, welcome to the world of JavaScript-mancy, where wizards, also known as JavaScriptmancers, control the arcane winds of magic wielding JavaScript to and fro and command the very fabric of reality.

We, programmers, sadly do not live in such a world. But we do have a measure of magic [1] in us, **we have the skills and power to create things out of nothingness**. And even if we cannot throw fireballs or levitate (*yet*), we can definitely change/improve/enhance reality and the universe around us with our little creations. Ain't that freaking awesome?

Well, I hope this book inspires you to continue creating, and using this beautiful skill we share, this time, with JavaScript.

A Story About Why I Wrote This Book

I was sitting at the back of the room, with my back straight and fidgetting with my fingers on the table. I was both excited and nervous. It was the first time I had ventured myself to attend to one of the unfrequent meetings of my local .NET user group. *Excited* because it was beyond awesome to be in the presence of so many like-minded individuals, people who loved to code like me, people who were so passionate about software development that were willing to sacrifice their free time to meet and talk about programming. *Nervous* because, of course, I did not want to look nor sound stupid in such a distinguished group of people.

The meetup started discussing *TypeScript* the new superset of JavaScript that promised *Nirvana* for C# developers in this new world of super interactive web applications. TypeScript here, TypeScript there because writing JavaScript sucked… JavaScript was the worst… everybody in the room started sharing their old war stories about writing JavaScript, how bad it was in comparison to C#, and so on…

"Errr… the TypeScript compiler writes beautiful JavaScript" I adventured to say… the room fell silent. People looking astonishingly at each other, uncomprehending, unbelieving… Someone had dared use *beautiful* and *JavaScript* in the same sentence.

This was not the first, nor will be the last time I have encountered such a reaction and feelings towards JavaScript as predominant in the .NET community. JavaScript is not worthy of our consideration. JavaScript is a toy language. JavaScript is unreliable and behaves in weird and unexpected ways. JavaScript developers don't know how to program. JavaScript tooling is horrible…

And every single time I sat muted, thinking to myself, reflecting, racking my brains pondering… How to show and explain that JavaScript is actually awesome? How to share that it is a beautiful language? A rich language that is super fun to write? That's how this book came about.

And let me tell you one little secret. Just some few years ago I felt exactly the same way about JavaScript. And then, all of the sudden, I started using it, with the mind of a beginner, without prejudices, without disdain. It was hard at first, being so fluent in C# I couldn't wrap my head

[1] "Any sufficiently advanced technology is indistinguishable from magic." Arthur C. Clarke. Love that quote :)

around how to achieve the same degree of fluency and expressiveness in JavaScript. Nonetheless I continued forward, and all of the sudden I came to love it.

The problem with JavaScript is that it looks too much like C#, enough to make you confident that you know JavaScript because you know C#. And just when you are all comfortable, trusting and unsuspecting JavaScript smacks you right in the face with a battle hammer, because, in many respects, JavaScript is not at all like C#. It just looks like it on the surface.

JavaScript is indeed a beautiful language, a little rough on the edges, but a beautiful language nonetheless. Trust me. You're in for a treat.

Why Should You Care About JavaScript?

You may be wondering why you need to know JavaScript if you already grok C#.

Well, first and foremost, *JavaScript is super fun to write*. Its lack of ceremony and super fast feedback cycles make it a fun language to program in and ideal for quick prototyping, quick testing of things, tinkering, building stuff and getting results fast. If you haven't been feeling it for programming lately, JavaScript will help you rediscover your passion and love for programming.

JavaScript is the language of the web, if you are doing any sort of web development, you need to understand how to write great JavaScript code and how JavaScript itself works. Even if you are writing a transpiled language like TypeScript or CoffeeScript, they both become JavaScript in the browser and thus knowing JavaScript will make you way more effective.

But *JavaScript is not limited to the web*, during the past few years JavaScript has taken the world by storm[2], you can write JavaScript to make websites, in the backend, to build mobile applications, games and even to control robots and IoT devices, which makes it a true cross-platform language.

JavaScript is a very approachable language, a forgiving one, easy to learn but hard to master. It is minimalistic in its contructs, beautiful, expressive and supports many programming paradigms. If you reflect about JavaScript features you'll see how it is built with simplicity in mind. Ideas such as type coercion (*are "44" and 44 so different after all?*) or being able to declare strings with either single or double quotes are great expressions of that principle.

JavaScript's openness and easy extensibility are the perfect foundations to make it a *fast-evolving language and ecosystem*. As the one language for the web, the language that browsers can understand, it has become the perfect medium for cross-pollination across all software development communities, where .NET developers ideas can meet and intermingle with others from the Ruby and Python communities. This makes knowledge, patterns and ideas spread accross boundaries like never before.

Since no one single entity really controls JavaScript[3], *the community has a great influence in how the language evolves*. With a thriving open source community, and openness and extensibility built

[2] http://githut.info/

[3] The ECMAScript standard in which JavaScript is based is evolved by the TC39 (Technical Committee 39) composed of several companies with strong interest in JavaScript (all major browser vendors) and distinguished members of the community. You can take a look at their GitHub page for a sneak-peek into how they work and what they are working in

within the language, it is the community and the browsers the ones that develop the language and the platform, and the standard bodies the ones that follow and stabilize the trends. When people find JavaScript lacking in some regard, they soon rush to fill in the gap with powerful libraries, tooling and techniques.

But don't just take my word for it. This is what the book is for, to show you.

What is the Goal of This Book?

The goal of this book is to make you fluent in JavaScript, able to express your ideas instantly and build awesome things with it. You'll not only learn the language itself but how to write idiomatic JavaScript. You'll learn both the most common patterns and idioms used in JavaScript today, and also all about the latest version of JavaScript: ECMAScript 6 (also known ECMAScript 2015) and even about the upcoming version ECMAScript 7.

> You can use ECMAScript as a synonym for JavaScript. It is true that we often use ES (short for ECMAScript) and a version number to refer to a specific version of JavaScript and its related set of new features. Particularly when these features haven't yet been implemented by all major browsers vendors. But for all intents and purposes ECMAScript is JavaScript. For instance, you will rarely hear explicit references to ES5.

But we will not stop there because what is a language by itself if you cannot build anything with it. I want to teach you everything you need to be successful and have fun writing JavaScript after you read this book. And that's why we will take one step further and take a glance at the JavaScript ecosystem, the JavaScript community, the rapid prototyping tools, the great tooling involved in building modern JavaScript applications, JavaScript testing and building an app in a modern JavaScript framework: Angular 2.

Why JavaScript-mancy?

Writing code is one of my favorite past times and so is reading fantasy books. For this project I wanted to mix these two passions of mine and try to make something awesome out of it.

In fantasy we usually have the idea of magic, usually very powerful, very obscure and only at the reach of a few dedicated individuals. There's also different schools or types of magic, pyromancy deals with fire magic, allomancy relates to magic triggered by metals, necromancy is all about death magic, raising armies of skeletons and zombies, immortality, etc.

I thought that drawing a parallel between magic and what we programmers do daily would be perfect. Because it is obscure to the untrained mind and requires a lot of work and study to get into, and because we have the power to create things out of nothing.

And therefore, **JavaScript-mancy**, **the arcane art of writing awesome JavaScript**.

Is This Book For You?

I have written this book for you C# developer:

- you that hear about the awesome stuff that is happening in the realm of JavaScript and are curious about it. You who would like to be a part of it, a part of this fast evolving, open and thriving community.
- you that have written JavaScript before, perhaps even do it daily and have been frustrated by it, by not been able to express your ideas in JavaScript, by not being able to get a program do what you wanted it to do, or struggling to do so. After reading this book you'll be able to write JavaScript as naturally as you write C#.
- you that think JavaScript a toy language, a language not capable of doing real software development. You'll come to see an expressive and powerful multiparadigm language suitable for a multitude of scenarios and platforms.

This book is specifically for C# developers because it uses a lot of analogies from the .NET world, C# and static typed languages to teach JavaScript. As a C# developer myself, I understand where the pain points lie and where we struggle the most when trying to learn JavaScript and will use analogies as a bridge between languages. Once you get a basic understanding and fluency in JavaScript I'll expand into JavaScript specific patterns and constructs that are less common in C# and that will blow your mind.

How is The Book Organized?

The book is organized in 3 parts focused in the language, the ecosystem and building your first app in JavaScript.

Part I. Mastering the Art of JavaScript-mancy will teach you to write JavaScript with fluency.

We will start looking briefly at the **basics of JavaScript** to set up the mood of the book and soon we'll discuss the big gotchas where we C# developers usually get stuck and stumble.

After that we will examine **object oriented programming in JavaScript**, studying prototypical inheritance, how to mimic C# (classic) inheritance in JavaScript. We will also look beyond class OOP into mixins, multiple inheritance and stamps where JavaScript takes you into interesting OOP paradigms that we rarely see in the more conventional C#.

We will then dive into **functional programming in JavaScript** and take a journey through LINQ, applicative programming, immutability, generators, combinators and function composition.

Organizing your JavaScript applications will be the next topic with the module pattern, commonJS, AMD (Asynchronous module definition) and ES6 modules.

Finally we will take a look at **Asynchronous programming** in JavaScript with callbacks, promises and reactive programming.

Since adoption of ES6 will take some time to take hold, and you'll probably see a lot of ES5 code for the years to come, we will start every section of the book showing the most common solutions and patterns of writing JavaScript that we use nowadays with ES5. This will be the perfect starting point to understand and showcase the new ES6 features, the problems they try to solve and how they can greatly improve your JavaScript.

In **Part II. Welcome to The Realm Of JavaScript** we'll take a look at the JavaScript ecosystem, following a brief history of the language that will shed some light on why JavaScript is the way it is today, continuing with the node.js revolution and JavaScript as a true cross-platform, cross-domain language.

Part II will continue with **how to setup your JavaScript development environment** to maximize your productivity and minimize your frustration. We will cover modern JavaScript and front-end workflows, JavaScript unit testing, browser dev tools and even take a look a various text editors and IDEs.

We will wrap Part II with a look at the role of **transpiled languages**. Languages like TypeScript, CoffeeScript, even ECMAScript 6, and how they have impacted and will affect JavaScript development in the future.

Part III. Building Your First Modern JavaScript App With Angular 2 will wrap up the book with a practical look at building modern JavaScript applications. Angular 2 is a great framework for this purpose because it takes advantage of all modern web standards, ES6 and has a very compact design that makes writing Angular 2 apps feel like writing vanilla JavaScript. That is, you won't need to spend a lot of time learning convoluted framework concepts, and will focus instead in developing your JavaScript skills to build a real app killing two birds with one stone (Muahahaha!).

In regards to the size and length of each chapter, aside from the introduction, I have kept every chapter small. The idea being that you can learn little by little, acquire a bit of knowledge that you can apply in your daily work, and get a feel of progress and completion from the very start.

Understanding the Code Samples in This Book

How to Run the Code Samples in This Book

For simplicity, I recommend that you start running the code samples in the browser. That's the most straightforward way since you won't need to install anything in your computer. You can either type them as you go in the browser JavaScript console (`F12` for Chrome if you are running windows or `Opt-CMD-J` in a Mac) or with prototyping tools like JsBin[4], jsFiddle[5], CodePen[6] or Plunker[7]. Any of these tools is excellent so you can pick your favorite.

[4] http://jsbin.io
[5] https://jsfiddle.net/
[6] http://codepen.io
[7] http://plnkr.co/

If you don't feel like typing, all the examples are available in jsFiddle/jsBin JavaScriptmancy library: http://bit.ly/javascriptmancy-samples[8].

For testing ECMAScript 6 examples I recommend JsBin[9], jsFiddle[10] or the Babel REPL at https://babeljs.io/repl/[11]. Alternatively there's a very interesting Chrome plugin that you can use to run both ES5 and ES6 examples called [ScratchJS][].

If you like, you can download all the code samples from GitHub[12] and run them locally in your computer using node.js[13].

Also keep an eye out for **javascriptmancy.com**[14] where I'll add interactive exercises in a not too distant future.

A Note About Conventions Used in the Code Samples

The book has three types of code samples. Whenever you see a extract of code like the one below, where statements are preceded by a `›`, I expect you to type the examples in a REPL.

> ### The REPL is Your Friend!
>
> One of the great things about JavaScript is the REPL (Read-Eval-Print-Loop), that is a place where you can type JavaScript code and get the results immediately. A REPL lets you tinker with JavaScript, test whatever you can think of and get immediate feedback about the result. Awesome right?
>
> A couple of good examples of REPLs are a browser's console (`F12` in Chrome/Windows) and node.js (take a look at the appendix to learn how to install node in your computer).

The code after `›` is what you need to type and the expression displayed right afterwards is the expected result:

```
› 2 + 2
// => 4
```

Some expressions that you often write in a REPL like a variable or a function declaration evaluate to `undefined`:

[8] http://bit.ly/javascriptmancy-samples
[9] http://jsbin.io
[10] https://jsfiddle.net/
[11] https://babeljs.io/repl/
[12]
[13] http://www.nodejs.org
[14] http://www.javascriptmancy.com

```
> var hp = 100;
// => undefined
```

Since I find that this just adds unnecessary noise to the examples I'll omit these `undefined` values and I'll just write the meaningful result. For instance:

```
> console.log('yippiiiiiiii')
// => yippiiiiiiii
// => undefined     <==== I will omit this
```

When I have a multiline statement, I will omit the › so you can more easily copy and paste it in a REPL or prototyping tool (*jsBin*, *CodePen*, etc). That way you won't need to remove the unnecessary › before running the sample:

```
let createWater = function (mana){
    return `${mana} liters of water`;
}
```

I expect the examples within a chapter to be run together, so sometimes examples may reference variables from previous examples within the same section. I will attempt to show smallish bits of code at a time for the sake of simplicity.

For more advanced examples the code will look like a program, there will be no › to be found and I'll add a filename for reference. You can either type the content of the files in your favorite editor or download the source directly from GitHub.

CrazyExampleOfDoom.js

```
export class Doom {
  constructor(){
    /* Oh no! You read this...
    /
    /  I am sorry to tell you that in 3 days
    /  at midnight the most horrendous apparition
    /  will come out from your favorite dev machine
    /  and it'll be your demise
    /  that is...
    /  unless you give this book as a gift to
    /  other 3 developers, in that case you are
    /  blessed for ever and ever
    */
  }
}
```

A Note About ECMAScript 5 and ECMAScript 6,7 within The Book

Everything in programming has a reason for existing. That hairy piece of code that you wrote seven months ago, that feature that went into an application, that syntax or construct within a language, *all were or seemed like good ideas at the time*. ES6, ES7 and future versions of JavaScript all try to improve upon the version of JavaScript that we have today. And it helps to understand the pain points they are trying to solve, the context in which they appear and in which they are needed. That's why this book will show you ES5 in conjunction with ES6 and beyond. For it will be much easier to understand new features when you see them as a natural evolution of the needs and pain points of developers today.

How will this translate into the examples within the book? - you may be wondering. Well I'll start in the beginning of the book writing ES5 style code, and slowly but surely, as I go showing you ES6 features, we will transform our ES5 code into ES6. By the end of the book, you yourself will have experienced the journey and have mastered both ES5 and ES6.

Additionally, it is going to take some time for us to start using ES6 to the fullest, and there's surely a ton of web applications that will never be updated to using ES6 features so it will be definitely helpful to know ES5.

A Note About the Use of Generalizations in This Book

Some times in the course of the book I will make generalizations for the sake of simplicity and to provide a better and more continuous learning experience. I will make statements such as:

> *In JavaScript, unlike in C#, you can augment objects with new properties at any point in time*

If you are experienced in C# you may frown at this, cringe, raise your fist to the sky and shout: *Why!? oh Why would he say such a thing!? Does he not know C#!?*. But bear with me. I will write the above not unaware of the fact that C# has the `dynamic` keyword and the `ExpandoObject` class that offer that very functionality, but because the predominant use of C# involves the use of strong types and compile-time type checking. The affirmation above provides a much simpler and clearer explanation about JavaScript than writing:

> *In JavaScript, unlike in C# where you use classes and strong types in 99% of the situations and in a similar way to the use of dynamic and ExpandoObject, you can augment objects with new properties at any point in time*

So instead of focusing on being correct 100% of the time and diving into every little detail, I will try to favor simplicity and only go into detail when it is conductive to understanding JavaScript which is the focus of this book. Nonetheless, I will provide footnotes for anyone that is interested in exploring these topics further.

Do You Have Any Feedback? Found Any Error?

If you have any feedback or have found some error in this book that you would like to report, then don't hesitate to drop me an email at jaime@vintharas.com.

A Final Word From the Author

The goal for this book is to be a holistic book. Holistic enough to give a good overview of the JavaScript language and ecosystem, yet contain enough detail to impart real knowledge about how JavaScript really works. That's a fine line to tread and sometimes I will probably cover too little or too much. If so don't hesitate to let me know. The beauty of a lean published book is that I have much more room to include improvements suggested by you.

There is a hidden goal as well, that is to make it as fun and enjoyable as possible. Therefore the fantasy theme of the whole book, the conversational style, the jokes and the weird sense of humor. Anyways, I have put my heart and soul into this book and hope you really enjoy it!

Jaime, 2015

Part I. Mastering the Arcane Art of JavaScript-mancy

Introduction

For many years JavaScript has been frowned upon and looked down on by many developers due to its quirky nature, obscure behaviors and many WTFs that populate its hairy APIs and operations.

Frown upon no more! For with modern design patterns, libraries, tools and the long awaited ECMAScript 6 (ES6, ES2015) writing JavaScript is now a pleasure.

Join me at the school of JavaScript-mancy as we travel along the modern landscape of writing JavaScript in 2015 and beyond, as we discover the organic evolution of this beautiful language and its thriving ecosystem, and delve in the latest features/spells/incantations of the JavaScript Arcana.

You my friend, can no longer ignore JavaScript. JavaScript is the most deployed language on earth, a beautiful and expressive language that supports many paradigms and which has a thriving community that continuously expands and improves its ecosystem with patterns, libraries, frameworks and tools. You don't want to miss this train.

But JavaScript, though forgiving and specially conceived to be easy to learn, can be either daunting for us that have a strongly-typed mindset and come from languages such as C# or Java or, more often, laughed at as a toy.

For you who consider it daunting and hate working with it worry not! I will show you the most common misconceptions and all the patterns you need to know to become as fluent in JavaScript as you are in C#.

For you who consider it a toy language, something associated not with real programming but with copy-paste coding or scripting to add flare to websites, I will show you all the different patterns and programming paradigms that JavaScript supports and which make it just as good and powerful as C#.

Let's get started!

Once Upon a Time...

Once upon a time, in a faraway land, there was a beautiful hidden island with captivating white sandy beaches, lush green hills and mighty white peaked mountains. The natives called it ***Asturi*** *and, if not for an incredible and unexpected event, it would have remained hidden and forgotten for centuries.*

Some say it was during his early morning walk, some say that it happened in the shower. Be that as it may, ***Branden Iech****, at the time the local eccentric and today considered the greatest Philosopher of antiquity, stumbled upon something that would change the world forever.*

In talking to himself, as both his most beloved companions and his most bitter detractors would attest was a habit of his, he stumbled upon the magic words of JavaScript and the mysterious REPL.

In the years that followed he would teach the magic word and fund the order of JavaScriptmancers bringing a golden age to our civilization. Poor, naive philosopher. For such power wielded by mere humans was meant to be misused, to corrupt their fragile hearts and bring their and our downfall. It's been ten thousand years, ten thousand years of wars, pain and struggle.

It is said that, in the 12th day of the 12th month of the 12th age a hero will rise and bring balance to the world. That happens to be today.

12th Age, Guardian of Chronicles

This book has a story in it. It is a story of a fantasy[15] world where some people can wield JavaScript to affect the world around them, to essentially program the world and bend it to their will. Cool right? The story follows the step of a heroine that comes to this hypothetical world to save it from evil, but of course, she needs to learn JavaScript first. **Care to join her in her quest to learn JavaScript and save the world?**

[15]For those of you that are not fantasy nerds I have included a small glossary at the end of the book where you can check words that you find strange. You should be able to understand the book and examples without the glossary, but I think it'll be more fun if you do

The Essential Ingredients Of JavaScript-Mancy

The importance of the fundamentals cannot be enough overstated,
the gathering of the proper ingredients for an incantation,
the carefulness and caring of the preparations,
the timing and intimate knowledge of the rituals,
everything affects the end result.

To practice is to be,
practice as you want to be,
for do you want to be an Artful Wizard, or a mediocre one?

> - Kely Sompsin,
> Maester of the Guild,
> Fourth Age

An Introduction to JavaScript-Mancy

I expect the reader of this manuscript to be well acquainted with the basics of programming, but I couldn't, in good faith, start the book without some sort of introduction to the arts of JavaScript. This first chapter will therefore take you through the whole breadth of the JavaScript language, albeit in a superficial way. I will show you all the basic features of JavaScript, even those of its latest incarnation ECMAScript 6, and how they differ or are similar to C#. I abhorr starting programming books with page after page on `for` loops and `if` statements so I will attempt to be as brief, interesting and entertaining as I can.

If you feel like you are well versed in the basics of JavaScript (and the new ES6 features) then by all means jump over this chapter, but be aware that there's a story happening in the examples, so you might be interested in taking a look. In any case, here we go...

```
/*
    And so here are we... at the start of a new adventure,
    our heroine sleeping peacefully in the middle of a clearing,
    surrounded by the darkness of a moonless night.

    We will call her... *stranger* since we do not yet know her name...

*/

stranger.says('hmmm... what?! where!?');
stranger.weaves('Console.WriteLine("lux++!")');
// => Uncaught ReferenceError: Console is not defined

stranger.says('hmm?');
stranger.weaves('Console.WriteLine("lux = lux + 1 !!")');
// => Uncaught ReferenceError: Console is not defined

/*

    The stranger curses and looks startled. Well I suppose she looks startled,
    it is hard to see a person's expression in the complete blackness of a
    moonless night as you well know...

*/

randalf.says("I'm afraid that is not going to work here...");
randalf.weaves("lux();");
// => A tiny wisp of light appears out of thin air and
//    illuminates the surroundings
```

```
/*
    Ok, now! THAT, my friend, is what startled looks like!");
*/

randalf.says("Hmm, you are not ready yet...no...no... " +
   "You are going to need to learn some ~~~JavaScript~~~");
```

JavaScript

Experiment JavaScriptmancer!!

You can experiment with all examples in this chapter within this jsBin[16] or downloading the source code from GitHub[17].

JavaScript, as we the guardians of JavaScript-mancy usually call the arts, is a multi-paradigm dynamic programming language. The multi-paradigm bit means that JavaScript lends itself well to different styles (paradigms) of programming like object-oriented or functional programming. The dynamic part means that... well the dynamic part is widely disputed even today... but for all intents and purposes we can say that JavaScript is evaluated as it is executed, there's no compilation step in which variables are bound, types are checked and expressions analyzed for correctness. The JavaScript runtime defers all this work until the code itself is being executed and this allows for a lot more freedom and interesting applications like metaprogramming [18].

JavaScript is also **dynamically typed** so a variable can reference many different types during its lifetime and you can augment objects with new methods or properties at any point in time.

I can, for example, summon a minion from the depths of hell using the `var` keyword and let it be a number:

```
> var minion = 1
> minion
// => 1
```

And I can do some alchemy thereafter and transform the minion into something else, a string, for example:

[16] http://bit.ly/javascriptmancy-basics
[17] https://github.com/vintharas/javascriptmancy-code-samples
[18] This is a gross oversimplification. If you've read some traditional literature about static vs dynamic programming languages you'll be familiar with the idea that static programming languages like C++, Java, C# are compiled and then executed whereas dynamic programming languages like Ruby, Python or JavaScript are not compiled but interpreted on-the-fly as they are executed. The repercussions of this being that a compiled program cannot be changed at runtime (you would need to recompile it), whereas an interpreted one can, since nothing is set in stone until it is run. In today's world however, JavaScript runtimes like V8 (Chrome) or SpiderMonkey (Mozilla) compile JavaScript to machine code, optimize it, and re-optimize it based on heuristics on how the code is executed.

```
> minion = "bunny";
// => "bunny"
```

I can keep doing that for as long as I want (as long as I have enough mana[19] of course), so let's make my `minion` an object:

```
> minion = {name: 'bugs', type: 'bunny'};
// => Object {name: 'bugs', type: 'bunny'}
```

In JavaScript I don't need a class to create an object. I can just create an object with whichever properties I desire and then later on augment [20] it with new properties to my heart's content:

```
> minion.description = 'A mean looking bunny';
> console.log(minion);
// => Object {name: bugs, type: bunny, description: A mean looking bunny}
```

JavaScript Has Some Types

JavaScript supports the following types: `Number`, `String`, `Object`, `Boolean`, `undefined`, `null` and `Symbol`.

As you may have guessed, JavaScript has a single type to represent all numbers. Which is pretty nice if you ask me, not having to ever think about doubles, and shorts, and longs and floats...

```
> var one = 1
> typeof(one)
// => "number"

> var oneAndAHalf = 1.5
> typeof(1.5)
// => "number"
```

There's a string type that works as you would expect any respectable string to behave. It lets you create string literals. Interestingly enough, JavaScript strings support both single (`'`) and double quotes (`"`):

[19] For those of you not familiar with magic, *mana* can be seen as a measure of *magical stamina*. As such, doing magic (like summoning minions) spends one's mana. An empty reservoir of mana means no spellcasting just as a empty reserve of stamina means no more running.

[20] As an article of interest you can augment objects in C# if you use the `dynamic` keyword with `ExpandoObjects`

```
> var text = "Thou shalt not pass!"
> typeof text
// => "string"

> var anotherBitOfText = 'No! Thou Shalt Not Pass!'
> typeof anotherBitOfText
// => "string"
```

JavaScript also has a boolean type to represent `true` and `false` values:

```
> var always = true
> var never = false
> typeof(always)
// => "boolean"
```

And an object type that we can use to create any new custom types:

```
> var skull = {name: 'Skull of Dark Magic'}
> typeof(skull)
// => object
```

JavaScript differs from other languages in that it has two different ways of representing the lack of something. Where C# has `null`, JavaScript has both `null` and `undefined`. Unlike in C#, the default value of anything that hasn't been yet defined is `undefined`, whereas `null` must be set explicitly.

```
> skull.description
// => undefined
> typeof(skull.description)
// => undefined

> skull.description = null;
> typeof(skull.description)
// => object :)
```

This can get even more confusing because of the fact that there's a third possibility. That a variable hasn't been declared:

```
> abracadabra
// => Uncaught ReferenceError: abracadabra is not defined
```

The confusion coming mainly from the error message: *abracadabra is not defined*. You can just think about these variables as *undeclared* instead of *not defined* and stick to the previous definition of `undefined`.

ECMAScript 6 brings a new primitive type, the symbol. Symbols can be seen as constant and immutable tokens that can be used as unique IDs.

```
> var crux = Symbol()
> typeof(crux)
// => symbol
```

Later within the book, we'll see how we can use Symbols to enable new patterns for hiding data in JavaScript.

Everything Within JavaScript Behaves Like an Object

In spite of JavaScript not having the concept of value types or reference types, numbers, strings and booleans behave like C# value types and objects behave like C# reference types. In practice, however, everything within JavaScript can be treated as an object.

Numbers for instance, provide several useful methods:

```
> (1).toString()
// => 1
> (3.14159).toPrecision(3)
// => 3.141
> (5000).toLocaleString('sv-SE')
// => 5 000
```

And so do strings:

```
> "a ghoul".toUpperCase()
// "A GHOUL"
```

Interesting right? If a number is a primitive value type, how come it has methods? Well, what is happening is that, whenever we call a method on a number or other primitive type, the JavaScript runtime is wrapping the primitive value in a special wrapper object. So even though `1` is not an object itself, when JavaScript evaluates `(1).toPrecision(3)` it wraps the value within the `Number` object on-the-fly. You can test the reverse process and instantiate a number using the wrapper object directly:

```
> var number = new Number(1);
// => Number {}
> typeof(number)
// => 'object'
```

Then unwrap the original value with `valueOf`:

```
> number.valueOf()
// => 1
```

Even more remarkable than numbers and strings, **functions behave like objects**. They have their own methods:

```
> var fireBall = function(){ world.spell('A blazing ball of fire materializes from t\
he palm of your hand!');};
> fireBall.apply
// => function apply(){}
```

And you can even add properties to a function:

```
{lang="javascript"}
> fireBall.maxNumberOfCharges = 5
// => 5;
> fireBall.maxNumberOfCharges
// => 5;
```

Let's take a quick look at each one of these types and how they work in JavaScript.

Strings in JavaScript

Strings, like in C#, let you represent text literals.

```
> "hi there"
// => hi there
> "creepy"
// => creepy
> "stop repeating what I say"
// => stop repeating what I say
```

Unlike in C# you can use both single (') and double quotes (") to create a string. Oftentimes you will see one used to escape the other:

```
> "this ain't cool man"
// => this ain't cool man
> 'you think you are so "funny"'
// => you think you are so "funny"
```

Any string has a number of useful methods:

```
> "I am tired of you devious REPL".split(' ');
// => ["I", "am", "tired", "of", "you", "devious", "REPL"]
> "I am tired of you devious REPL".replace('tired', 'ecstatic');
// => I am ecstatic of you devious REPL
> "I am tired of you devious REPL".indexOf('tired');
// => 5
```

ES6 also brings a number of new methods like `startsWith`, `endsWith`, `repeat`:

```
> "Stop REPL!".startsWith("Stop");
// => true
> "Stop REPL!".endsWith("REPL!");
// => true
> "NaN".repeat(10) + " BatMan!!!"
// => NaNNaNNaNNaNNaNNaNNaNNaNNaN BatMan!!!!
> "ha! Now I beat you at your own game!"
// => "ha! Now I beat you at your own game!"
```

Until recently, there was no such thing as C# `String.format` nor `StringBuilder` so most injecting values in strings was done using the `+` operator or `string.concat`:

```
> var target = 'minion';
> "obliterate " + target + " with hellfire!"
// => obliterate minion with hellfire!
> "obliterate ".concat(target, " with hellfire!")
// => obliterate minion with hellfire!
```

Fortunately, ES6 brings *template strings* and a more elegant approach to string interpolation.

Better String Interpolation ES6 Template Strings

The new **ES6 Template Strings** improve greatly the way you can operate with strings. They allow string interpolation based on the variables that exist in the scope where the template is evaluated, thus providing a much cleaner and readable syntax.

In order to create a template string you surround the string literal with backticks and use `${variable-in-scope}` to specify which variable to include within the resulting string:

```
> var spell = 'hellfire';
> `obliterate ${target} with ${spell}!`
// => obliterate minion with hellfire!
```

Template strings also let you easily create multi-line strings.

Where in the past you were forced to make use of string concatenation and the new line character `\n`:

```
> "One ring to rule them all\n" +
  "One ring to find them;\n" +
  "One ring to bring them all\n" +
  "and in the darkness bind them"
// =>  One ring to rull them all
       One ring to find them;
       One ring to bring them all
       and in the darkness bind them
```

ES6 Template strings let you write a multi-line string in a more straightforward fashion:

```
> `One ring to rull them all
  One ring to find them
  One ring to bring them all
  and in the darkness bind them`
// =>  One ring to rull them all
       One ring to find them;
       One ring to bring them all
       and in the darkness bind them
```

Functions in JavaScript

Functions are the most basic building component in JavaScript. As such, they can live more independent lives than methods in C# which are always tied to a class. So, you'll oftentimes see functions alone in the wild:

```
> function obliterate(target){
    console.log(`${target} is obliterated into tiny ashes`);
}
> obliterate('rabid bunny')
// => rabid bunny is obliterated into tiny ashes
```

JavaScript, in a radically different way than C#, lets you call a function with any number of arguments, even if they are not defined in a function's signature:

```
> obliterate('rabid bunny', 'leprechaun', 'yeti')
// => rabid bunny is obliterated into tiny ashes
```

And even with no arguments at all, although depending on the function implementation it may cause some chaos and some mayhem:

```
> obliterate()
// => undefined is obliterated into tiny ashes
```

You can use the very special `arguments` array-like object to get hold of the arguments being passed at runtime to a given function:

```
> function obliterateMany(){
    // ES6 method to convert arguments to an actual array
    var targets = Array.from(arguments).join(', ');
    console.log(`${targets} are obliterated into tiny ashes`);
}
> obliterateMany('Rabid bunny', 'leprechaun', 'yeti')
// => Rabid bunny, leprechaun, yeti are obliterated into tiny ashes
```

Or the finer **ES6 rest syntax** reminescent of C# `params`:

```
> function obliterateMany(...targets){
    console.log(`${targets} are obliterated into tiny ashes`);
}
> obliterateMany('Rabid bunny', 'leprechaun', 'yeti')
// => Rabid bunny, leprechaun, yeti are obliterated into tiny ashes
```

In addition to functions working as... well... functions, they perform many other roles in JavaScript and are oftentimes used as building blocks to achieve higher-order abstractions: they are used as object constructors, to define modules, as a means to achieve data hiding and have many other uses.

ES6 changes this complete reliance on functions a little bit as it provides new higher level constructs that are native to the language, constructs like **ES6 classes** and **ES6 modules** which you'll be able to learn more about within the book. Indeed throughout this manuscript I'll show you both ES5 constructs, the present and the past, and ES6 ones, the present-ish and the future, so you'll feel at home in any JavaScript codebase you happen to work with.

Functions as Values

An interesting and very important aspect of functions in javascript is that they can be treated as values, this is what we mean when we say **functions are first-class citizens** of the language. It means that they are not some special construct that you can only use in certain places, with some special conditions and grammar. Functions are just like any other type in JavaScript, you can store them in variables, you can pass them as arguments to other functions and you can return them from a function.

For instance, let's say you want to create a very special logger that prepends your name to any message that you wish to log:

```
> var log = function(msg){ console.log(msg);}
> var logByRandalf = function (msg, logFn){
    logFn(`Randalf logs: ${msg}`);
}
> logByRandalf('I am logging something, saving it to memory for ever', log);
// => Randalf logs: I am logging something, saving it to memory for ever
```

But that was a little bit awkward, what if we return a function with the new functionality that we desire:

```
> var createLogBySomeone = function (byWho){
    return function(msg){
        return console.log(`${byWho} logs: ${msg}`);
    };
}
> var logByRandalf = createLogBySomeone('Randalf');
> logByRandalf('I am logging something, saving it to memory for ever');
// => Randalf logs: I am logging something, saving it to memory for ever
```

If you feel a little bit confused by this don't worry, we will dive deeper into functional programming, closures and high-order functions later within the book. For now just realize that **functions are values** and you can use them as such.

JavaScript Has Function Scope

Another very interesting aspect of JavaScript that is diametrically opposed to how things work in C# is the scope of variables. JavaScript variables have **function scope and not block scope**. This means that functions define new scopes for variables and not blocks of code (*if statements*, *for loops*, code between `{}`, etc...) which highlights once more the importance of functions in JavaScript.

You can appreciate function scope in all its glory in these examples. First if you declare a single function with an `if` block you can verify how the `if` block doesn't define a new scope as you would expect in C#:

```
> function scopeIsNuts(){ // new scope for scopeIsNuts
    console.log(x); // => undefined
    if (true){
        var x = 1;
    }
    console.log(x); // => 1
}
```

But if we replace the `if` block with a new function `inner`, then we have two scopes:

```
> function outer(){ // new scope for outer
    var x = 0;
    console.log(x); // => 0

    function inner(){ // new scope for inner
        var x = 1;
        console.log(x); // => 1
    }
    inner();

    console.log(x); // => 0
}
```

ES6 let, ES6 const and Block Scope

ES6 attemps to bring an end to the confusion of JavaScript having function scope with the `let` keyword that allows you to create variables with block scope. With ES6 you can either use `var` for function scoped variables or `let` for block scoped ones.

If you rewrite the example we used to illustrate function scope with `let`, you'll obtain a very different result:

```
> function scopeIsNuts(){ // new scope for scopeIsNuts
    console.log(x); // => undefined
    if (true){
        let x = 1;
        console.log(x); // => 1
    }
    console.log(x); // => undefined
}
```

Now the `x` variable only exists within the `if` statement block. Additionally, you can use the `const` keyword to declare constant variables with block scope.

```
> function scopeIsNuts(){ // new scope for scopeIsNuts
    console.log(x); // => undefined
    if (true){
        const x = 1;
        console.log(x); // => 1
        x = 2; // => TypeError
    }
    console.log(x); // => undefined
}
```

ES6 Default Arguments

ES6 finally brings default arguments to JavaScript functions, and they work just like in C#:

```
> function fireBall(target, mana=10){
   var damage = 1.5*mana;
   console.log(`A huge fireball springs from
your fingers and hits the ${target} with ${damage} damage`);
}
> fireBall('troll')
// => A huge fireball springs from your fingers and hits the troll
//    with 15 damage
> fireBall('troll', /* mana */ 50)
// => A huge fireball springs from your fingers and hits the troll
//    with 75 damage
```

ES6 Destructuring

Another nifty ES6 feature is destructuring. Destructuring lets you unwrap any given object into a number of properties and bind them to variables of your choice. You can take advantage of destructuring with any object:

```
> var {hp, defense} = {name: 'conan',
                       description: 'cimmerian barbarian king of thieves',
                       hp: {current: 9000, max: 9000},
                       defense: 100, attack: 400};
> console.log(hp);
// => {current: 9000, max: 9000}
> console.log(defense);
// => 100
```

Even when passing an object to a function:

```
function calculateDamage({attack}, {hp, defense}){
  var effectiveAttackRating = attack - defense + getHpModifier(hp);
  var damage = attackRoll(effectiveAttackRating);
  return damage > 0 ? damage: 0;

  function getHpModifier(hp){
    return hp.current < 0.1*hp.max ? 10 : 0;
  }

  function attackRoll(dice){
    // do some fancy dice rolling
    return dice;
  }
}

var troll = {
  name: 'Aaagghhhh',
  description: 'nasty troll',
  hp: {current: 20000, max: 20000},
  defense: 40, attack: 100
};
var conan = {name: 'conan',
  hp: {current: 200, max: 200},
  defense: 1000, attack: 1000
};
console.log(calculateDamage(troll, conan));
// => 0
// => no troll gonna damage conan
```

ES6 Arrow Functions

Another great feature brought by ES6 are arrow functions which resemble C# lambda expressions. Instead of writing the `obliterate` function as we did before, we can use the arrow function syntax:

```
/*
> function obliterate(target){
    console.log(`${target} is obliterated into tiny ashes`);
}
*/
> let obliterate = target =>
    console.log(`${target} is obliterated into tiny ashes`);
> obliterate('minion');
// => minion is obliterated into tiny ashes
> obliterate('rabid bunny')
// => rabid bunny is obliterated into tiny ashes
```

And if you have a function with more arguments or statements, you can write it just like we do in C#:

```
> let obliterateMany = (...targets) => {
    targets = targets.join(', ');
    console.log(`${targets} are obliterated into tiny ashes`);
};
> obliterateMany('bunny', 'leprechaun', 'yeti');
// => Bunny, leprechaun, yeti are obliterated into tiny ashes
```

We will dive deeper into arrow functions later in the book and see how they not only provide a terser and more readable syntax but also serve a very important function in what regards to safekeeping the value of `this` in JavaScript. (We've got ourselves a very naughty `this` in JavaScript as you'll soon appreciate yourself)

OOP and Objects in JavaScript

JavaScript has great support for object-oriented programming with objects literals, constructor functions, prototypical inheritance, ES6 classes and less orthodox OOP paradigms like mixins and stamps.

Objects in JavaScript are just key/value pairs. The simplest way to create an object is using an object literal:

```
> var scepterOfDestruction = {
    description: 'Scepter of Destruction',
    toString: function() {
        return this.description;
    },
    destruct: function(target) {
        console.log(`${target} is instantly disintegrated`);
    }
}
> scepterOfDestruction.destruct('apple');
// => apple is instantly disintegrated
```

ES6 makes easier to create object literals with syntactic sugar for functions also known as *method shorthand*:

```
> var scepterOfDestruction = {
    description: 'Scepter of Destruction',
    toString() {
        return this.description;
    },
    destruct(target) {
        console.log(`${target} is instantly disintegrated`);
    }
}
```

And for creating properties from existing variables also known as *property shorthand*:

```
> var damage = 10000;
> var scepterOfDestruction = {
    description: 'Scepter of Destruction',
    damage, // as opposed to damage: damage
    toString() {
        return this.description;
    },
    destruct(target) {
        console.log(`${target} is instantly disintegrated`);
    }
}
> scepterOfDestruction.damage;
// => 10000
```

This works great for one-off objects. When you want to reuse the same type of object more than once you can either use a vanilla factory method or a constructor function with the `new` keyword:

```
// by convention constructor functions are capitalized
> function Scepter(name, damage, spell){
    this.description = `Scepter of ${name}`,
    this.damage = damage;
    this.castSpell = spell;
    this.toString = () => this.description;
}
> var scepterOfFire = new Scepter('Fire', 100,
    (target) => console.log(`${target} is burnt to cinders`));
> scepterOfFire.castSpell('grunt');
// => grunt is burnt to cinders
```

Prototypical Inheritance

Yet another big diffence between C# and JavaScript are their inheritance models. JavaScript exhibits what is known as prototypical inheritance. That means that objects inherit from other objects which therefore are called prototypes. These objects create what is known as a *prototypical chain* that is traversed when the JavaScript runtime tries to determine where in the chain a given method is defined.

Let's say that you have an object that represents an abstraction for any item that can exist in your inventory:

```
> var item = {
   durability: 100,
   sizeInSlots: 1,
   toString(){ return 'an undescriptive item';}
}
> item.toString();
// => an undescriptive item
```

And a two handed iron sword that in addition to being an item (and an awesome item at that) has its own specific set of traits:

```
> var ironTwoHandedSword = {
    damage: 60,
    sizeInSlots: 2,
    wield() {
        console.log('you wield your iron sword crazily over your head');
    },
    material: 'iron',
    toString() {return 'A rusty two handed iron sword';}
};
```

You can take advantage of JavaScript prototypical inheritance to reuse the item properties across many items, by setting the `item` object as the prototype[21] of the `ironTwoHandedSword` (and any other specific items that you create afterwards).

```
> Object.setPrototypeOf(ironTwoHandedSword, item);
```

This will establish a prototypical chain, so that, if we attempt to retrieve the sword `durability`, the JavaScript runtime will traverse the chain and retrieve the property from the `item` prototype:

```
> ironTwoHandedSword.durability;
// => 100
```

If, on the other hand, you attempt to access a property that exists in both the prototype and the object itself, the nearest property in the chain will win:

```
> ironTwoHandedSword.toString();
// => A rusty two handed iron sword
```

ES6 exposes the `__proto__` property that lets you directly assign a prototype through an object literal:

```
> var ironTwoHandedSword = {
    __proto__: item,
    damage: 60,
    // etc...
};
> ironTwoHandedSword.prototype = item;
```

There's a lot more to prototypical inheritance and the many different OOP paradigms supported by JavaScript. But we'll look into them further later in the book.

[21] One does not simply change the prototype of an object willy-nilly at runtime since it can affect performance a lot. Instead, you would use Object.create or a function constructor, and if needed, you would augment the prototype as needed.

ES6 Classes

A new addition to JavaScript you might have heard about and celebrated are ES6 classes. The existence of ES6 classes doesn't mean that JavaScript gets classes just like C# and we're not going to worry about constructor functions and prototypical inheritance anymore. **ES6 classes are "just" syntactic sugar over the existing inheritance model and the way we craft objects in JavaScript**. That being said, it is a great declarative way to represent constructor/prototype pairs.

A JavaScript class looks very similar to a C# class:

```
class Item {
  constructor(durability = 100, sizeInSlots = 1){
    this.durability = durability;
    this.sizeInSlots = sizeInSlots;
  }
  toString(){
    return 'an undescriptive item';
  }
}
var item = new Item();
item.toString();
// => an undescriptive item
```

And so does inheritance:

```
class Sword extends Item {
  constructor(durability = 500, sizeInSlots = 2,
              damage = 50, material = 'iron'){
    super(durability, sizeInSlots);
    this.damage = damage;
    this.material = material;
  }
  wield() {
    console.log(`you wield your ${this.material} sword
crazily over your head`);
  }
  toString() {
    return `A ${this.material} sword`;
  }
};
var sword = new Sword();
sword.wield();
// => you wield your iron sword crazily over your head
```

```
## Arrays, Maps and Sets in JavaScript

Up until recently JavaScript had only one single data structure, albeit very verstat\
ile, to handle collections of items: the array. You can create an array using using \
square brackets `[]`:

{lang="javascript"}
```

$$[1, 2, 3, 4 ,5] \text{ // } \Rightarrow [1,2,3,4,5]$$

You can mix and match the different elements of an array. There's no type restrictions so you can have numbers, strings, objects, functions, arrays, etc... in much the same way that you can find the most strange items in a kender's [22] pouch:

```
> var aKendersPouch = [
    'jewel',
    '3 stones',
     1,
    {name: 'Orb of Power'},
    function() { return 'trap!';}
    ];
```

You can access the items of an array via their indexes:

```
> aKendersPouch[0]
// => jewel
> aKendersPouch[4]()
// => trap!
```

You can also traverse the indexes of an array using the `for/in` loop:

```
> for (var idx in aKendersPouch) console.log(aKendersPouch[idx]);
// => jewel
// => 3 stones
// => ...etc
// => function() { return 'trap!';}
```

And even better the items of an array using ES6 `for/of` loop:

[22] like a hobbit but with shoes. If curious look up the joyful, corageous and beloved Tasslehoff Burrfoot.

```
> for (var item of aKendersPouch) console.log(item);
// => jewel
// => 3 stones
// => ...etc
// => function() { return 'trap!';}
```

Arrays have a lot of cool and useful methods that you can use to add/remove or otherwise operate on the items within the array:

```
> aKendersPouch.length
// => 5

// add item at the end of the array
> aKendersPouch.push('silver coin');
// => 6 (returns the current length of the array)
> aKendersPouch.push('6 copper coins', 'dental floss');
// => 8

// pop item at the end of the array
> aKendersPouch.pop();
// => dental floss

// insert item at the beginning
> aKendersPouch.unshift('The three Musketeers');
// => 8

// extract item from the beginning of the array
> aKendersPouch.shift();
// => 'The three musketeers'
```

And even LINQ-like methods to perform functional style transformations within an array:

```
> const isValuable = item => parseInt(item) > 5;
> const toGoldCoins = item => parseInt(item) || 0;
> const sumCoins = (sum, price) => sum + price;

> var goldCoins = aKendersPouch
                    .filter(isValuable) // ES6 analogous to LINQ Where
                    .map(toGoldCoins) // analogous to LINQ Select
                    .reduce(sumCoins, 0); // analogous to LINQ Aggregate
> console.log(goldCoins);
// => 6
```

You will learn a ton more about arrays and JavaScript versions of LINQ later in the book.

ES6 Spread Operator and Arrays

The **ES6 spread operator** can also be used to merge or flatten an array within another array:

```
> var newRecruits = ['Sam', 'John', 'Connor'];
> var merryBandits = ['Veteran Joe', 'Brave Krom', ...newRecruits];
> merryBandits;
// => ["Veteran Joe", "Brave Krom", "Sam", "John", "Connor"]
```

ES6 Maps and Sets

ES6 gives us magicians two new data structures to work with: maps, a true key/value pair data structure and sets to handle collections of unique items.

You can create a new map using the `Map` constructor:

```
> var libraryOfWisdom = new Map();
> libraryOfWisdom.set('A',
    ['A brief history of JavaScript-mancy', 'A Tale of Two Cities']);
> libraryOfWisdom.get('A')
// => ['A brief history of JavaScript-mancy', 'A Tale of Two Cities'];
```

You can even seed a map with existing information by sending an array of key/value pairs[23]:

```
> var libraryOfWisdom = new Map([
    ['A', ['A brief history of JavaScript-mancy', 'A Tale of Two Cities']],
    ['B', ['Better Dead Than Powerless: Tome I of Nigromantics']]
]);
> libraryOfWisdom.get('B');
// => ['Better Dead Than Powerless: Tome I of Nigromantics']
```

In a similar fashion, you create sets using the `Set` constructor:

```
> var powerElements = new Set(['earth', 'fire', 'water', 'wind']);
> powerElements
// => Set {"earth", "fire", "water", "wind"}
```

Sets will ensure that you don't have duplicated data within a collection:

[23] in reality, it does not need to be an array, but an iterable that produces key/value pairs `[<key>:<value>]`

```
> powerElements.add('water').add('earth').add('iron');
> console.log(powerElements);
// => Set {"earth", "fire", "water", "wind", "iron"}
```

JavaScript Flow Control

JavaScript gives you the classic flow control structures that you are accustomed to: `if`, `for`, `while` loops behave much in the same way in JavaScript than in C# (but for the function scoped variables of course).

In addition to these, JavaScript has the `for/in` loop that lets you iterate over the properties of an object:

```
> var spellOfFarseeing =
    { name: 'Spell of Farseeing',
      manaCost: 10,
      description: 'The spell lets you see a limited' +
                   'portion of a far away location;'}

> for (var prop in spellOfFarseeing) {
    console.log(`${prop} : ${spellOfFarseeing[prop]}`);
}
// => name : Spell of Farseeing
// => manaCost : 10
// => description : The spell lets you see a limited portion of a far away location
```

And the ES6 `for/of` loop that lets you iterate over collections[24] (arrays, maps and sets):

```
> for (var element of powerElements) console.log(element);
// => earth
// => fire
// => water
// => etc...
```

Logical Operators in JavaScript

Abstract Equality and Strict Equality

JavaScript equality operators behave in a particularly special way. The operators that you are accustomed to use in C# == and != are called **abstract equality operators** and evaluate the equality

[24] for the sake of correctness, you can use the for/of loop not only on arrays, maps and sets but on anything that implements the iterable protocol. We will discuss iterability later within the book.

of expressions in a loose way. If the two expressions being evaluated by one of these operators don't match in type, they'll be converted to a matching type. For instance, in evaluating the abstract equality of `42` and `"42"`, the string will be converted to a number resulting in both values being equal:

```
> 42 == '42'
==> true
```

Fortunately JavaScript also provides operators that performs strict equality (`===` and `!==`) which is basically a comparison of two values without the implicit type conversion.

```
> 42 === '42'
==> false
```

Implicit Type Conversion Also Known As Type Coercion

This implicit conversion that takes place in JavaScript gives birth to the concept of *truthy* and *falsey*. Since any value can be evaluated as a boolean, we say that some values like an array `[]` or an object `{}` are truthy, and some other values like empty string `''` or *undefined* are falsey. In the examples below we use the `!!` to explicitly convert values to boolean for clarity purposes:

```
> !![]
// => true
> !!{}
// => true
> !!""
// => false
> !!undefined
// => false
> !!null
// => false
> !!0
// => false
```

This allows for a terser way to write `if` statements

```
> if (troll) // as opposed to (troll != null && troll != undefined)
```

Since `troll` is coerced to a boolean type, having the `troll` variable holding an object value will evaluate to *truthy* and having it holding `null` or `undefined` will be *falsey*. In either case the `if` statement will fulfill its purpose while being much nicer to read.

OR and AND

OR (`||`) and **AND** (`&&`) operations also behave in an interesting way. The *OR* operation will return the first truthy expression or the last falsey expression (if all expressions are falsey):

```
// 0 falsey
// 'cucumber' truthy
// 42 truthy
> 0 || 'cucumber' || 42
// => 'cucumber'
> 0 || false || undefined
// => undefined
```

The *AND* operator will return the last truthy expression or the first falsey expression (if any falsey expression is encountered):

```
// 0 falsey
// 'cucumber' truthy
// 42 truthy
> 0 && 'cucumber' && 42
// => 0
> true && 'cucumber' && 42
// => 42
```

Exception Handling

Exception handling works similar as in C#, you have your familiar `try/catch/finally` blocks:

```
> try { asdf; }
  catch (e) { console.log(e.message);}
  finally { console.log('done!');}
// => asdf is not defined
// => done!
```

And you can throw new exceptions with the `throw` keyword:

```
> throw new Error("We're all gonna die!");
// => Uncaught Error: We're all gonna die!
```

Additionally, you can improve your error semantics and create custom errors by inheriting the `Error` prototype.

Regular Expressions

JavaScript also supports regular expressions. You can create a regular expression in two ways, either by wrapping the expression between slash (`/`):

```
> var matchNumbers = /\d+/;
> matchNumbers.test('40 gold coins');
// => true
> matchNumbers.exec('40 gold coints');
// => ["40"]
```

Or by creating a `RegExp` object:

```
> var matchItems = new RegExp('\\D+');
> matchItems.test('40 gold coins');
// => true
> matchItems.exec('40 gold coints');
// => [" gold coins"]
```

Strings have built-in support for regular expressions as well with the `match` and `search` methods:

```
> var bagOfGold = '30 gold coins';
> bagOfGold.match(/\d+/);
// => ['30']
> bagOfGold.search(/\d+/);
// => 0 (index where first match is found)
```

But Beware, JavaScript Can Be Weird and Dangerous

So far you've seen the bests parts of JavaScript and nothing too weird or inconsistent. But sometimes you'll experience strange behaviors in some less visited corners of JavaScript like any of the following:

```
> x = 10;
// => added a variable to the global scope (window.x)
> NaN == NaN
// => false
> null == undefined
// => true
> typeof(null)
// => object
> [] + []
// => ''
> [] + {}
// => {}
> {} + []
// => 0
> {} + {}
// => NaN
```

Oftentimes you won't run into these issues when building real web applications and my advice is that you ignore them. Be aware that they exist but just don't use them, or use patterns or conventions to avoid them.

Concluding

And that was a summary of pretty much the whole JavaScript language. I really hope it has sparkled your interest for JavaScript and that you cannot wait to turn the next page and learn more. But first, let's make a quick review of what you've learned in this chapter.

We've seen that JavaScript is a very flexible dynamic language that supports many paradigms of programming and has a lot of great features.

You have learned the many things you can do with strings and ES6 string templates. How functions are very independent entities that can live their own lives completely separate from objects and how they are a fundamental building block of applications in JavaScript. You also discovered arrow functions that resemble lambdas in C# and let you write super terse and beautiful code.

We took a look at objects, object initializers, prototypical inheritance and ES6 classes. We saw the different data structures supported, the versatility of the array and an overview of the new ES6 Map and Set.

We also examined more general language features like the flow control structures and logical operators. We saw the difference between abstract comparison and strict comparison, highlighted the implicit type conversion inherent to JavaScript, the existence of the concepts of *truthy* and *falsey* and the way the *OR* and *AND* operators work.

Finally we reviewed exception handling and regular expressions, and we saw some of the weird and best-avoided behaviors in JavaScript.

```
/*

    The first rays of a new day like dubious trendils of light
    start approaching the clearing when Randalf notices that the stranger
    is looking weirdly at him...

*/

randalf.says("Yes, I know what you are thinking, it is a lot to take in...");
stranger.says('...err... no... Who the hell are you? and whaaaat is a kender?!');
```