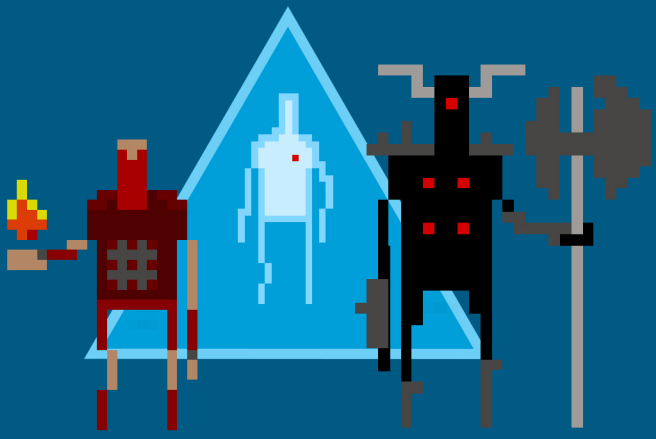
JavaScript-Mancy



Object-Oriented Programming

Mastering the Arcane Art Of Summoning Objects In JavaScript for C# Developers

Jaime González García

JavaScript-mancy: Object-Oriented Programming

Mastering the Arcane Art of Summoning Objects in JavaScript for C# Developers

Jaime González García

This book is for sale at http://leanpub.com/javascript-mancy-object-oriented-programming

This version was published on 2019-08-13

ISBN 978-1976459238

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Also By Jaime González García

JavaScript-mancy

JavaScript-mancy: Getting Started

Wizards Use Vim

Boost Your Coding Fu With VSCode and Vim

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 Note to the Illustrious Readers of JavaScript-mancy: Getting Started

If you are a reader of *JavaScript-mancy: Getting Started* then let me start this book by thanking you. When I started writing the JavaScript-mancy series little did I know about the humongous quest I was embarking in. Two years later, I have written more than a thousand pages, loads of code examples, hundreds of exercises, spent an insane amount of time reviewing the drafts, reviewing the reviews, etc... But all of this work is meaningless without you, the reader. Thank you for trusting in me and in this series, I hope you enjoy this book more than you enjoyed the first one. Go forth JavaScript-mancer!

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

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 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 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

[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

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?

This book is the second installment of the JavaScript-mancy series and its goal is to provide a great and smooth introduction to JavaScript Object-Oriented Programming to C# developers. Its goal is to teach you how you can bring and reuse all your C# knowledge into JavaScript and, at the same time, boost your OOP skills with new paradigms that take advantage of JavaScript dynamic nature.

What is the Goal of The JavaScript-mancy Series?

The goal of the JavaScript-mancy series 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 versions of JavaScript: ECMAScript 6 (also known ES6 and ES2015) , ES7 (ES2016), ES2017 and beyond.

> 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 series. 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 [4].

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**.

[4] Previously known as Angular 2 and later re-branded to just Angular. The former version of Angular 1.x is now known as Angular.js

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.

That being said, a lot[5] of the content of the book is useful beyond C# and regardless of your software development background.

[5] Really, A LOT :)

How is The Book Organized?

The goal of this book is to provide a smooth ride in learning OOP to C# developers that start developing in JavaScript. Since we humans like familiarity and analogy is super conductive to learning, the first part of the book is focused on helping you learn how to bring your OOP knowledge from C# into JavaScript.

We'll start examining the pillars of object oriented programming: encapsulation, inheritance and polymorphism and how they apply to JavaScript and its prototypical inheritance model.

We will continue with how to emulate classes in JavaScript prior to ES6 which will set the stage perfectly to demonstrate the value of ES6 classes.

After that we will focus on alternative object-oriented paradigms that take advantage of the dynamic nature of JavaScript to achieve great flexibility and composablity in a fraction of the code.

Later we'll move onto object internals and the obscure art of meta-programming in JavaScript with the new Reflect API, proxies and symbols.

Finally, we'll complete our view of object-oriented programming in JavaScript with a deep dive into TypeScript, a superset of JavaScript that enhances your developer experience with new features and type annotations.

How Are The JavaScript-mancy Series Organized? What is There in the Rest of the Books?

The rest of the books are organized in 3 parts focused in the language, the ecosystem and building your first app in JavaScript.

After this introductory book **Part I. Mastering the Art of JavaScriptmancy** continues by examining **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[6], jsFiddle[7], CodePen[8] or Plunker[9].

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

Any of these tools is excellent so you can pick your favorite.

If you don't feel like typing, all the examples are available in jsFiddle/jsBin JavaScriptmancy library: http://bit.ly/javascriptmancy-samples[10].

For testing ECMAScript 6 examples I recommend JsBin[11], jsFiddle[12] or the Babel REPL at https://babeljs.io/repl/[13]. 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[14] and run them locally in your computer using node.js[15].

Also keep an eye out for **javascriptmancy.com**[16] 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?

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

> 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`:

```
> 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('yippiiiiiii')
// => yippiiiiiii
// => 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 the Exercises

In order to encourage you to experiment with the different things that you will learn in each chapter I wrap every single one of them with exercises.

It is important that you understand that there is almost no wrong solution. I invite you to let your imagination free and try to experiment and be playful with your new found knowledge to your heart's content. I do offer a solution for each exercise but more as a guidance and example that as the one right solution.

In some of the exercises you may see the following pattern:

```
// mooleen.weaves('some code here');
mooleen.weaves('teleport("out of the forest", mooleen, randalf)');
```

This is completely equivalent to:

```
// some code here
teleport("out of the forest", mooleen, randalf);
```

I just use a helper function `weaves` to make it look like *Moolen, the mighty wizard* is casting a spell (in this case `teleport`).

A Note About ECMAScript 5 (ES5) and ES6, ES7, ES8 and ESnext 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 Regarding the Use of `var`, `let` and `const`

Since this book covers both ES5, ES6 and beyond the examples will intermingle the use of the `var`, `let` and `const` keywords to declare variables. If you aren't familiar with what these keywords do here is a quick recap:

- `var`: use it to declare variables with function scope. Variables declared with `var` are susceptible to hoisting which can result in subtle bugs in your code.
- `let`: use it to declare variables with block scope. Variables declared with `let` are not hoisted. Thanks to this, `let` allows you to declare variables nearer to where they are used.
- `const`: like `let`, but in addition, it declares a one-time binding. That is, a variable declared with `const` can't be bound to any other value. Attempting to assign the value of a `const` variable to something else will result in an error.

The examples for ES5 patterns like mimicking classes before the advent of ES6 (and the new `let` and `const`) will use `var`. The examples for post ES6 features like ES6 classes and onwards will use `let` and `const`. Of these two we will prefer the latter that offers a safer alternative to `let`, and we will use `let` in those cases where we need or want to allow assigning a variable multiple times. That being said there may be occasions where I won't follow these rules when a particular example escapes mine and my reviewer's watchful eye.

If you want to learn more about JavaScript scoping rules and the `var`, `let` and `const` keywords then I recommend you to take a look at JavaScript-mancy: Getting Started[17] the first book of this series.

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

[17] https://www.javascriptmancy.com/

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 or reach me on twitter @vintharas[18].

A Final Word From the Author

The goal for this series of books is to be holistic. 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, 2017

[18] https://twitter.com/Vintharas

Tome I. Mastering the Arcane Art of JavaScript-mancy

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[19] world where some people can wield JavaScript to affect the world around them,

[19]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

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?**

Tome II. JavaScriptmancy and OOP: The Path of The Summoner

Path of Summoning and Commanding Objects (Also Known as Object Oriented Programming)

Introduction to the Path of Summoning and Commanding Objects (aka OOP)

Many ways to build a Golem there are,

cast its blueprint in clay
then recite the instantiation chants,

or put together the parts
that'll made the whole alive,

or bring it forth at once
with no prior thought required.

Many ways to build a Golem there are,
in JavaScript.

- KeDo,
Master Artificer,
JavaScript-mancy poems

```
/*
Mooleen sits in a dark corner of a tavern sipping a jug of
the local brew.

She flinches. The local brew surely must have fire wyvern's
blood in it.

She silently observes the villagers around her.

They seem unhappy and nervous. As if they were expecting
something terrible was about to befall them any second.
*/

mooleen.says("A month has passed since we dispatched Great");
mooleen.says("You would think they would be happier");

rat.says("Well, people don't like change or surprises");
rat.says("They're expecting that someone worse will take control");
rat.says("Better the devil you know...");

/*
A maid stops by Mooleen's table confused
*/
maid.says("Are you feeling alright, sir? Speaking to yourself?");

rat.movesOutOfTheShadows();
maid.shrikes();

villager.shouts("A demon!!!");

rat.says("Great");
mooleen.says("That's just plain mean");

/*
The villagers quickly surround the dark corner with clubs, bottles
and whichever crude weapon they can muster.
*/
villager.shouts("Kill the demon!!");

mooleen.weaves("teleport('Caves of Infinity')");

/*
Mooleen and rat blink out of existence just as various pointy weapons
blink into existence precisely where they were sitting a second
earlier.
*/
```

```
randalf.says("There you are!");
mooleen.says("here I am!");
rat.says("A demon!?");

randalf.exclaims("A demon? Where!!");
bandalf.says("Yes where!")
zandalf.looksWorriedAllAround();

mooleen.says("There's no demon");
randalf.asks("Are you sure?");

randalf.says("We need to be on our toes");
mooleen.asks("You too?");

randalf.says("Yes, it's been a month, they must be about to attack");
mooleen.says("They? Who!");

randalf.says("Could be anyone really... The Dark Brootherhood, " +
 "The Clan, The Silver Guild, The Red Hand... " +
 "They'll want to control Asturi");
randalf.says("You need to summon an army");

mooleen.says("An army?");
randalf.says("An army indeed, n' bigger than the one you had before");

mooleen.says("Really? Cause that took a looooong time to summon");
randalf.says("Well, That's because you're a novice");

mooleen.says("That's encouraging");
randalf.says("Oh, don't you worry, " +
             "We'll take care of your ignorance");
mooleen.says("Ouch");

randalf.says("Let me tell you about OOP in JavaScript");
```

Let me Tell You About OOP in JavaScript

Welcome to *the Path of Summoning*[20] *and Commanding Objects*! In this part of this ancient manuscript you'll learn how you can work with objects in JavaScript, how to define them, create them and even how to interweave them. By the end of it you'll have mastered Object Oriented Programming in JavaScript and you'll be ready to command your vast armies of objects into eternal glory.

JavaScript OOP story is pretty special. When I started working seriously with JavaScript some years ago, one of my first concerns as a C# developer coming to JavaScript was to find out how to write a class. I had a lot of prowess in C# and I wanted to bring all my knowledge and abilities into the world of JavaScript, so my first approach was to try to map every C# concept into JavaScript. I saw classes, which are such a core construct in C# and which were such an important part of my programming style at the time, as my secret weapon to being proficient in JavaScript.

Well, for the life of me I couldn't find a good reference to this-is-how-you-write-a-class-in-JavaScript. It took me a long while to understand how to mimic classical inheritance. But it was time well spent because, along the way, I learnt a lot about JavaScript and about the many different ways in which it supports object-oriented programming. Moreover, this quest helped me look beyond classical inheritance into other OOP styles more akin to JavaScript where flexibility and expressiveness reign supreme over the strict and fixed taxonomies of classes.

In this part of the series I will attempt to bring you with me, hand in hand, through the same journey that I experienced. We will start with how to achieve classical inheritance in JavaScript, so you can

[20] In Fantasy, wizards of all sorts and kinds *summon* or *call forth* creatures to act as servants, or warriors, and follow the wizard's commands. As a JavaScript-mancer you'll be able to use Object Oriented Programming to summon your own objects into reality and do with them as you please.

get a basic level of proficiency by translating your C# skills into JavaScript. And then we will move beyond that into new patterns that truly leverage JavaScript as a language and which will blow your mind.

Experiment JavaScriptmancer!!

You can experiment with all examples in this chapter directly within this jsBin[21] or downloading the source code from GitHub[22].

Let's have a taste of what is in store for you by getting a high level overview [23] of object-oriented programming in JavaScript. Don't worry if you feel you can't follow the examples. In the upcoming chapters we will dive deeper into each of the concepts and techniques used, and we will discuss them separately at a much slower pace.

C# Classes in JavaScript

A C# *class* is more or less equivalent to a JavaScript *constructor function* and *prototype* pair:

[21] http://bit.ly/javascriptmancy-oop-introduction

[22] https://github.com/vintharas/javascriptmancy-code-samples

[23] In this section I am going to make a lot of generalizations and simplifications in order to give a simple and clear introduction to OOP in JavaScript. I'll dive into each concept in greater detail and with an appropriate level of correctness in the rest of the chapters ahead.

```
// Here we have a Minion constructor function
function Minion(name, hp){
  // The constructor function usually defines
  // the data within a "class", the properties
  // contained within a constructor function
  // will be part of each object created with it
  this.name = name;
  this.hp = hp;
}

// The prototype usually defines the methods
// within a "class". It is shared across all
// Minion instances
Minion.prototype.toString = function(){
  return this.name;
};
```

The *constructor function* represents how an object should be constructed (or created) while the *prototype* represents bits of reusable behavior. In practice, the *constructor function* usually defines the data members within a *"class"* while the *prototype* defines its methods.

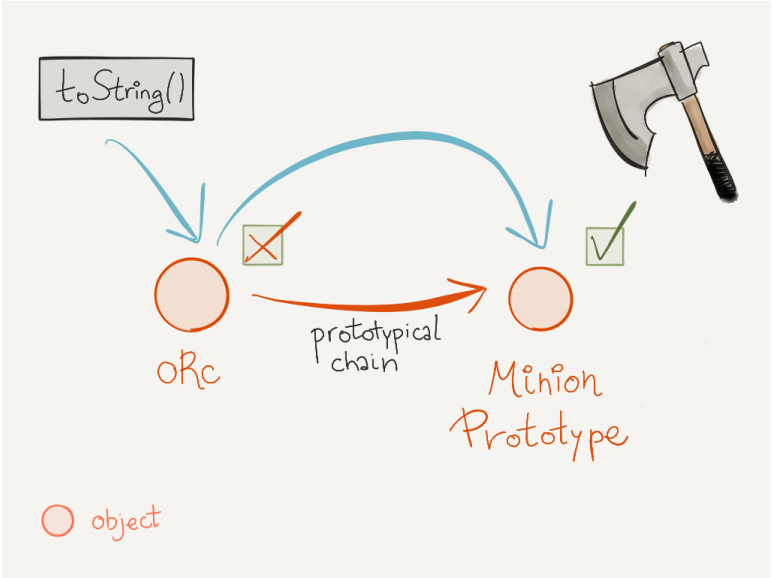
You can instantiate a new `Minion` object by using the `new` operator on the *constructor function*:

```
var orc = new Minion('orc', 100);
console.log(orc);
// => [object Object] {
//  hp: 100,
//  name: "orc",
//  toString: function () {
//    return this.name;
//  }
// }

console.log(orc.toString())
// => orc

console.log('orc is a Minion: ' + (orc instanceof Minion));
// => true
```

As a result of instantiating an `orc` we get a new `Minion` object with

two properties `hp` and `name`. The `Minion` object also has a hidden property called `[[prototype]]` that points to its `prototype` which is an object that has a method `toString`. This *prototype* and its `toString` method are shared across all instances of the `Minion` class.

When you call `orc.toString` the JavaScript runtime checks whether or not the `orc` object has a `toString` method and if it can't find it, *like in this case*, it goes down the *prototype chain* until it does. The *prototype chain* is established by the object itself, its prototype, its prototype's prototype and so on. In this case, the *prototype chain* leads to the `Minion.prototype` object that has a `toString` method. This method will then be called and evaluated as `this.name` (whose value is `orc` in this example).



The prototypical chain

We can mimic classical inheritance by defining a new *"class"* `Wizard` and making it inherit from `Minion`:

```
// Behold! A Wizard!
function Wizard(name, element, hp, mana){
  // the constructor function calls its parent constructor function
  // using [Function.prototype.call] (or apply)
  Minion.call(this, name, hp);
  this.element = element;
  this.mana = mana;
}

// the prototype of the Wizard is a Minion object
Wizard.prototype = Object.create(Minion.prototype);
Wizard.prototype.constructor = Wizard;
```

We achieve *classical inheritance* by:

1. Calling the `Minion` *constructor function* from the `Wizard` *constructor*.
2. Creating a new object that has Minion as its prototype (via `Object.create`) and assigning it to be the `Wizard` prototype. This is how you establish a prototypical chain between `Wizard` and `Minion`.

```
Wizard object => Wizard Prototype => Minion Prototype => Object Prototype
```

By following these two steps we achieve two things:

1. With the *constructor* delegation we ensure that a `Wizard` object has all the properties of a `Minion` object.
2. With the *prototype chain* we ensure that all the methods in the `Minion` prototype are available to a `Wizard` object.

We can also augment the `Wizard` *prototype* with new methods like this `castsSpell` method that allows the wizard to cast powerful spells:

```
// we can augment the prototype with a new method to
// cast mighty spells
Wizard.prototype.castsSpell = function(spell, target){
    console.log(this + ' casts ' + spell + ' on ' + target);
    this.mana -= spell.mana;
    spell(target);
};
```

Or even override or extend existing methods within its base *"class"* `Minion`:

```
// we can also override and extend methods
Wizard.prototype.toString = function(){
    return Minion.prototype.toString.apply(this, arguments) +
  ", the " + this.element +" Wizard";
};
```

Finally, we can verify that everything works as expected by instantiating our very own powerful wizard:

```
var gandalf = new Wizard(/* name */ "Gandalf",
                         /* element*/ "Grey",
                         /* hp */ 50,
                         /* mana */ 50);
```

The `gandalf` object is both an instance of `Wizard` and `Minion` which makes sense:

```
console.log('Gandalf is a Wizard: ' + (gandalf instanceof Wizard));
// => Gandalf is a Wizard: true
console.log('Gandalf is a Minion: ' + (gandalf instanceof Minion));
// => Gandalf is a Minion: true
```

The `toString` method works as defined in our overridden version:

```
console.log(gandalf.toString());
// => Gandalf, the Grey Wizard
```

And our great Grey wizard can cast potent spells:

```
// A lightning spell
var lightningSpell = function(target){
  console.log('A bolt of lightning electrifies ' + target + '(-10hp)');
  target.hp -= 10;
};
lightningSpell.mana = 5;
lightningSpell.toString = function(){ return 'lightning spell';};

gandalf.castsSpell(lightningSpell, orc);
// => Gandalf, the Grey Wizard casts lightning spell on orc
// => A bolt of lightning electrifies orc (-10hp)
```

As you can see from these previous examples, writing *"classes"* prior to *ES6* was no easy feat. It required a lot of moving parts and a lot of code. That's why *ES6* brings *classes* along which provide a much nicer syntax to what you've seen thus far. Instead of having to handle *constructor functions* and *prototypes* yourself, you get the new `class` keyword that nicely wraps both into a more coherent and developer friendly syntax:

```
// this is the equivalent of the Minion
class ClassyMinion{
  constructor(name, hp){
    this.name = name;
    this.hp = hp;
  }
  toString(){
    return this.name;
  }
}

const classyOrc = new ClassyMinion('classy orc', 50);
console.log(classyOrc);
// => [object Object] {
//  hp: 100,
//  name: "classy orc"
//}

console.log(classyOrc.toString());
// => classy orc

console.log('classy orc is a ClassyMinion: ' +
```

```
  (classyOrc instanceof ClassyMinion));
// => classy orc is a ClassyMinion: true
```

ES6 classes also provide the `extend` and `super` keywords which improve how classes can relate and interact with parent classes. `extend` lets you establish class inheritance in a readable, declarative fashion and `super` lets you access methods from parent classes:

```
// and this is the equivalent of the Wizard
class ClassyWizard extends ClassyMinion{
  constructor(name, element, hp, mana){
    // super lets you access the parent class methods
    // like the parent class constructor
    super(name, hp);
    this.element = element;
    this.mana = mana;
  }
  toString(){
    // or any other method
    return super.toString() + ", the " + this.element +" Wizard";
  }
  castsSpell(spell, target){
    console.log(this + ' casts ' + spell + ' on ' + target);
    this.mana -= spell.mana;
    spell(target);
  }
}
```

Again, we can verify that it works just like it did before by instantiating a *classy* wizard:

```
const classyGandalf = new Wizard(/* name */ "Classy Gandalf",
                                 /* element */ "Grey",
                                 /* hp */ 50,
                                 /* mana */ 50);
console.log('Classy Gandalf is a ClassyWizard: ' +
            (classyGandalf instanceof ClassyWizard));
// => Classy Gandalf is a ClassyWizard: true

console.log('Classy Gandalf is a ClassyMinion: ' +
            (classyGandalf instanceof ClassyMinion));
// => Classy Gandalf is a ClassyMinion: true

console.log(classyGandalf.toString());
// => Classy Gandalf, the Grey Wizard

classyGandalf.castsSpell(lightningSpell, classyOrc);
// => Classy Gandalf, the Grey Wizard casts lightning spell
//    on classy orc
// => A bolt of lightning electrifies classy orc(-10hp)
```

With ES6 classes we can achieve the same result than before with less code and better code at that. **It is important to highlight though that ES6 classes are just syntactic sugar**[24]. Under the hood, these ES6 classes that you have just seen are equivalent to *constructor function/prototype* pairs.

And that is how you mimic classical inheritance in JavaScript. Now let's look beyond.

OOP Beyond Classes

There are a lot of people in the JavaScript community who claim that the cause of JavaScript not having a nice way to mimic classical inheritance, not having classes, is that you were not meant to use them in the first place. You were meant to embrace *prototypical inheritance*, the natural way of working with inheritance in

[24] They are also safer to use: They aren't hoisted and JavaScript will alert you if you try to call a class constructor without the `new` operator.

JavaScript, instead of perverting it to make it behave sort of like *classical inheritance*.

In the world of *prototypical inheritance* you only have objects, and particularly objects that are based upon other objects which we call *prototypes*. Prototypes lend behaviors to other objects by means of delegation (via the *prototype chain*) or by the so called *concatenative inheritance* which consists in copying behaviors.

Let's illustrate the usefulness of this type of inheritance with an example. Imagine that, in addition to *wizards*, we also need to have some *thieves* for those occasions when we need to use a more gentle/shrew hand against our enemies.

A `ClassyThief` class could look something like this:

```
class ClassyThief extends ClassyMinion{
  constructor(name, hp){
    super(name, hp);
  }
  toString(){
    return super.toString() + ", the Thief";
  }
  steals(target, item){
    console.log(`${this} steals ${item} from ${target}`);
  }
}
```

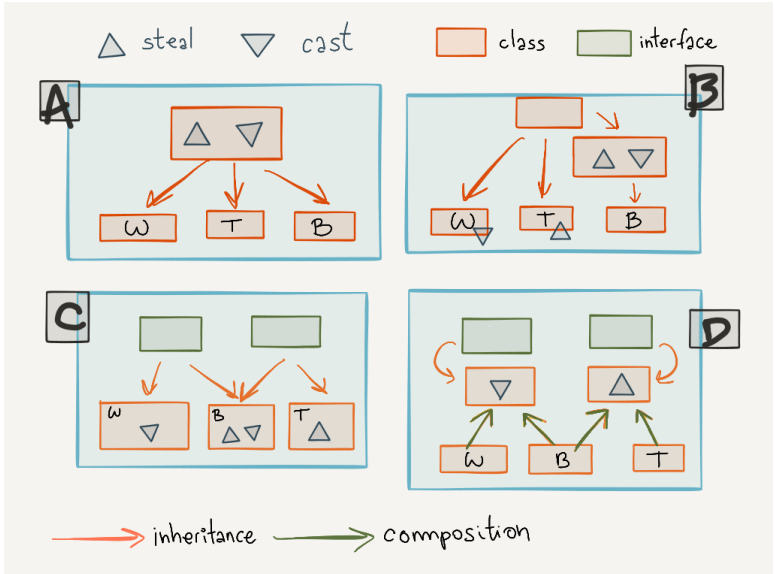
And let's say that a couple of weeks from now, we realize that it would be nice to have yet another type of minion, one that can both cast spells and steal, and why not? Play some music. Something like a *Bard*. In *pseudo-code* we would describe it as follows:

```
// class Bard
// should be able to:
// - cast powerful spells
// - steals many items
// - play beautiful music
```

Well, we've put ourselves in a pickle here. *Classical inheritance* tends to build rigid taxonomies of types where something is a `Wizard`, something is a `Thief` but it cannot be both. *How would we solve the issue of the `Bard` using classical inheritance in C#?* Well...

A. We could move both `castsSpell` and `steals` methods to a base class `SpellCastingAndStealingMinion` that all three types could inherit. The `ClassyThief` would throw an exception when casting spell and so would the `ClassyWizard` when stealing. Not a very good solution (goodbye Liskov principle [25])
B. We could create a `SpellCastingAndStealingMinion` that duplicates the functionality in `ClassyThief` and `ClassyWizard` and make the `Bard` inherit from it. This solution would imply code duplication and thus additional maintenance.
C. We could define interfaces for these behaviors `ICanSteal`, `ICanCastSpells` and make each class implement these interfaces. Nicer but we would need to provide an specific implementation in each separate class. No so much code reuse here.
D. We could do as in the previous solution, but delegate the implementation of stealing and casting to another class that could be reused by wizards, thieves and bards. This would achieve more code reuse but it'd require a lot of extra artificial plumbing to do the delegation.

[25] The Liskov substitution principle is one of the S.O.L.I.D. principles of object-oriented design. It states that derived classes must be substitutable for their base classes. This means that a derived class should behave as portrayed by its base class and not break the expectations created by its interface. In this particular example if you have a `castsSpell` and a `steals` method in the base class, and a derived class throws an exception when you call them you are violating this principle. That's because the derived class breaks the expectations established by the base class (i.e. that you should be able to use both methods).



So none of these solutions are very attractive: They involve bad design, code duplication or both. *Can JavaScript help us achieve a better solution to this problem?* **Yes! It can!**

Imagine that we broke down all these behaviors and encapsulated them inside separate objects (`canCastSpells`, `canSteal` and `canPlayMusic`):

```
const canCastSpells = {
  castsSpell(spell, target){
    console.log(this + ' casts ' + spell + ' on ' + target);
    this.mana -= spell.mana;
    spell(target);
  }
};

const canSteal = {
  steals(target, item){
    console.log(`${this} steals ${item} from ${target}`);
  }
};

```

```
const canPlayMusic = {
  playsMusic(){
    console.log(`${this} grabs his ${this.instrument} ` +
                `and starts playing music`);
  }
};

// Bonus behavior to identify a character by name!
const canBeIdentifiedByName = {
  toString(){
    return this.name;
  }
};
```

Now that we have encapsulated each behavior in a separate object we can compose them together to provide the necessary functionality to a `wizard`, a `thief` and a `bard`:

```
// And now we can create our objects by composing
// these behaviors together
function TheWizard(element, mana, name, hp){
  const wizard = {element,
                  mana,
                  name,
                  hp};
  Object.assign(wizard,
                canBeIdentifiedByName,
                canCastSpells);
  return wizard;
}

function TheThief(name, hp){
  const thief = {name,
                 hp};
  Object.assign(thief,
                canBeIdentifiedByName,
                canSteal);
  return thief;
}

function TheBard(instrument, mana, name, hp){
  const bard = {instrument,
                mana,
                name,
```

```
                hp};
  Object.assign(bard,
                canBeIdentifiedByName,
                canSteal,
                canCastSpells,
                canPlayMusic);
  return bard;
}
```

And in a very expressive way we can see how a `wizard` is someone than can cast spells, a `thief` is someone that can steal and a `bard` someone that not only can cast spells and steal but can also play music. By stepping out of the rigid limits of classical inheritance and static typing, we get to a place where we can easily reuse behaviors and compose new objects in a very flexible and extensible manner.

We can verify that indeed this approach works beautifully. The `Wizard` **casts powerful spells**:

```
const wizard = TheWizard('fire', 100, 'Randalf, the Red', 10);

wizard.castsSpell(lightningSpell, orc);
// => Randalf, the Red casts lightning spell on orc
// => A bolt of lightning electrifies orc(-10hp)
```

The `Thief` sneaks on you and **steals**:

```
const thief = TheThief('Locke Lamora', 100);

thief.steals('orc', /*item*/ 'gold coin');
// => Locke Lamora steals gold coin from orc
```

And the `Bard`, truly gifted `Bard`, **casts spells**, **steals** and **plays music**:

```
const bard = TheBard('lute', 100, 'Kvothe', 100);

bard.castsSpell(lightningSpell, orc);
// => Kvothe casts lightning spell on orc
// =>A bolt of lightning electrifies orc(-10hp)

bard.steals('orc', /*item*/ 'sandwich');
// => Kvothe steals sandwich from orc

bard.playsMusic();
// => Kvothe grabs his lute and starts playing music
```

The `Object.assign` in the examples is an *ES6* method that lets you extend an object with other objects. This is effectively the *concatenative prototypical inheritance* we mentioned previously.

> We usually call these objects *mixins*. A *mixin* in JavaScript is just an object that you compose with other objects to provide them with additional behavior or state. In the simplest example of *mixins* you just have a single object extending another object, but there're also functional *mixins*, where you use functions instead. We will cover all these *mixin* patterns in detail later in the book with a deep dive into Object.assign and possible alternatives in ES5.

This object composition technique constitutes a very interesting and flexible approach to object-oriented programming that isn't available in C#. But in JavaScript we can use it even with *ES6 classes*!

Combining Classes with Object Composition

Do you remember that *ES6 classes* are just syntactic sugar over the existing *prototypical inheritance model*? They may look like *classical inheritance* but they are not. This means that the following mix of *ES6 classes* and *object composition* would work:

```
class ClassyBard extends ClassyMinion{
  constructor(instrument, mana, name, hp){
    super(name, hp);
    this.instrument = instrument;
    this.mana = mana;
  }
}

Object.assign(ClassyBard.prototype,
       canSteal,
       canCastSpells,
       canPlayMusic);
```

In this example we extend the `ClassyBard` prototype with new functionality that will be shared by all future instances of `ClassyBard`. If we instantiate a new *bard* we can verify that it can **steal**, **cast spells** and **play music**:

```
const anotherBard = new ClassyBard('guitar', 100, 'Jimmy Hendrix', 100);

anotherBard.steals('orc', /*item*/ 'silver coin');
// => Jimmy Hendrix steals silver coin from orc

anotherBard.castsSpell(lightningSpell, orc);
// => Jimmy Hendrix casts lightning spell on orc
// => A bolt of lightning electrifies orc(-10hp)

anotherBard.playsMusic();
// => Jimmy Hendrix grabs his guitar and starts playing music
```

This is an example of *delegation-based prototypical inheritance* in which methods such as `steals`, `castsSpell` and `playsMusic` are

delegated to a single *prototype* object (instead of being appended to each object individually).

So far you've seen classical inheritance mimicked in JavaScript, *ES6 classes* and object composition via mixin objects, but there's much more to learn and in greater detail! Take a sneak peak at what you'll learn in each of the upcoming chapters and get excited!

The Path of the Object Summoner Step by Step

In **Summoning Fundamentals: an Introduction to Object Oriented Programming in JavaScript** you'll start by understanding the basic constructs needed to define and instantiate objects in JavaScript. In this chapter, *constructor functions* and the `new` operator will join what you've discovered thus far about *object initializers*. You'll review how to achieve **information hiding**, you'll learn the basics of JavaScript's **prototypical inheritance** model and how you can use it to reuse code/behaviors and improve your memory footprint. You'll complete the foundations of JavaScript OOP by understanding how JavaScript achieves **polymorphism**.

In **White Tower Summoning or Emulating Classical Inheritance in JavaScript** you'll use *constructor functions* in conjunction with *prototypes* to create the equivalent of C# classes in JavaScript. You'll then push the boundaries of JavaScript inheritance model further and emulate C# classical inheritance building inheritance chains with method extension and overriding just like in C#.

In **White Tower Summoning Enhanced: the Marvels of ES6 Classes** you'll learn about the new *ES6 Class* syntax and how it provides a much better *class* development experience over what it was possible prior to *ES6*.

In **Black Tower Summoning: Objects Interweaving Objects with Mixins** we'll go beyond classical inheritance into the arcane realm

of *object composition* with mixins. You'll learn about the extreme extensibility of object-oriented programming based on object composition. How you can define small pieces of reusable behavior and properties that combined together can create powerful objects (effectively achieving multiple inheritance).

In **Black Tower Summoning: Safer Object Composition with Traits** you'll learn about an object composition alternative to mixins called traits. Traits are as reusable and composable as mixins but are even more flexible and safe as they let you define required properties and resolve conflicts.

In **Black Tower Summoning Enhanced: Next Level Object Composition With Stamps** you'll find out about a new way to work with objects in JavaScript called *Stamps* that brings object composability to the next level.

You'll then dive into the depths of **Object Internals and metaprogramming** in JavaScript. You'll discover the mysteries of the low level JavaScript `Object` APIs, the new ESnext decorators, ES6 proxies, ES6 `Reflection` APIs and symbols.

Finally, we will complete the path of the Summoner by taking a look at **TypeScript**. TypeScript offers the nearest experience to C# that you can find on the web. It is a superset of JavaScript that enhances your developer experience with new features and type annotations. These type annotations bring static typing to JavaScript but they are flexible enough not to sacrifice JavaScript's dynamic nature.

Concluding

JavaScript is a very versatile language that supports a lot of programming paradigms and different styles of Object-Oriented Programming. In the next chapters you'll see how you can combine a small number of primitive constructs and techniques to achieve a variety of OOP styles.

JavaScript, like in any other part of the language, gives you a lot of freedom when working with objects, and sometimes you'll feel like there are so many options and things you can do that you won't know what's the right path. Because of that, I'll try to provide you with as much guidance as I can and highlight the strengths and weaknesses of each of the options available.

Get ready to learn some JavaScript OOP!

```
randalf.says("See? There's a lot of stuff for you to learn");
mooleen.says("Is any of that going to help me get home?");

randalf.says("Most definitely.");
randalf.says("I have scourged our library and found nothing " +
    "about this 'earth' you speak of. And now that I think about " +
    "it, what a weird name for a kingdom...");
randalf.says("Anyway, the only other option is the golden " +
             "library of Orrile...");

mooleen.says("Awesome! Then just show me the way");

randalf.says("... in Tates, guarded by The Deadly Seven... ");
mooleen.says("I can take care of them");

randalf.says("... and the vast host of armies " +
             "of the most powerful sorcerer alive");
mooleen.says("I see");
rat.says("downer");

mooleen.says("You were saying something about OOP techniques?...");
```