

The Little JavaScript Book

All you wanted to know about JavaScript but never dared asking!

Valentino Gagliardi

The Little JavaScript Book

All you wanted to know about JavaScript but never dared asking!

Valentino Gagliardi

This book is for sale at <http://leanpub.com/little-javascript>

This version was published on 2019-07-11



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.

© 2019 Valentino Gagliardi

Contents

| | |
|---|----------|
| Intro. Few words about the book | 1 |
| A brief history of the book | 1 |
| Structure of the book | 1 |
| Who this book is for? | 1 |
| What should I know before reading this book? | 1 |
| Typographic conventions | 1 |
| What's the best way for following the examples? | 2 |
| Errata and other requests | 2 |
| About the author | 2 |
| Acknowledgments | 3 |

Part 1 - JavaScript, the “hard” parts **4**

| | |
|--|-----------|
| Chapter 1. Getting started with JavaScript | 5 |
| What is JavaScript? | 5 |
| What can I build with JavaScript? | 5 |
| Why should I study JavaScript? | 6 |
| ECMAScript, ES5, ES6 and other technical terms | 6 |
| A special note on ES2015 | 8 |
| Conclusions | 8 |
| Chapter 2. Prologue: JavaScript fundamentals | 9 |
| Strings, numbers, variables and all in between | 9 |
| Standing on the shoulders of a giant Object | 12 |
| 50 shades of JavaScript functions | 14 |
| Conclusions | 19 |
| Chapter 3. A whirlwind tour of JavaScript engines | 21 |
| JavaScript engines and Global Memory | 22 |
| Global Execution Context and Call Stack | 24 |
| Asynchronous JavaScript, Callback Queue and the Event Loop | 26 |
| Callback hell and ES6 Promises | 30 |
| ES6 Promises and error handling | 32 |

CONTENTS

| | |
|---|-----------|
| ES6 Promises combinators: Promise.all, Promise.allSettled, Promise.any, and friends | 34 |
| ES6 Promises and Microtask Queue | 35 |
| Asynchronous evolution: from Promises to async/await | 35 |
| Conclusions | 38 |
| Chapter 4. Closures and modules in JavaScript | 40 |
| Global madness | 40 |
| Demystifying closures | 41 |
| The need for closures | 45 |
| Conclusions | 48 |
| Chapter 5. The secret life of JavaScript objects | 50 |
| Everything is an object! | 50 |
| Creating and linking objects | 53 |
| Constructing JavaScript objects | 58 |
| Checking the “linkage” | 61 |
| Protecting objects from manipulation | 65 |
| Class illusion | 68 |
| Conclusions | 69 |
| Chapter 6. This in JavaScript | 70 |
| Demystifying “this” | 70 |
| Rule number 1: default binding | 72 |
| Rule number 2: implicit binding | 73 |
| Rule number 3: explicit binding | 74 |
| Rule number 4: “new” binding | 82 |
| Conclusions | 82 |
| Chapter 7. Types, conversion, and comparison in JavaScript | 83 |
| Primitive types in JavaScript | 83 |
| When a number becomes a string | 85 |
| I’m not a number! | 87 |
| Equal equal or not? | 89 |
| Primitives and objects | 90 |
| Conclusions | 93 |
| Part 2 - Working with JavaScript | 95 |
| Chapter 8. Manipulating HTML elements with JavaScript | 96 |
| The Document Object Model | 96 |
| Nodes, elements, and DOM manipulation | 98 |
| Generating HTML tables with JavaScript | 101 |
| How to dynamically add DOM elements with JavaScript | 109 |
| Conclusions | 117 |

CONTENTS

| | |
|---|------------|
| Chapter 10. Working with asynchronous JavaScript | 119 |
| REST API and XMLHttpRequest | 119 |
| XMLHttpRequest in action: generating lists with HTML and JavaScript | 120 |
| Asynchronous evolution: from XMLHttpRequest to Fetch | 124 |
| Error handling with Fetch | 124 |
| Fetch with async/await | 124 |
| Rebuilding the Fetch API from scratch (for fun and profit) | 124 |
| Conclusions | 141 |

Intro. Few words about the book

A brief history of the book

Why another book on JavaScript? As much as I love “You don’t know JS” and “Eloquent JavaScript” I feel there is the need for a book which takes the reader by hand. Everyone has a unique viewpoint on technical topics and my readers love my style of teaching. This book was originally released in italian but a lot of fellow devs asked for a translation. And here it is. The “Little JavaScript Book” aims to be a reference for the hard parts of JavaScript while being beginner friendly. I hope you’ll appreciate. Enjoy the reading!

Structure of the book

The “Little JavaScript Book” is organized in three parts. The first part covers the inner working of JavaScript engines and the “hard parts” of the language: closures, the prototype system, `this` and `new`. Every chapter ends with a self-assessment test which is useful for making the concepts stick. The second part of the book has a lot of practical exercises on the DOM with a lot of pages converning code organization and best practices. The third and last part contains solutions for the exercises and future additions to the book.

Who this book is for?

While writing the book I had in mind web developers who always worked with jQuery or JavaScript without digging deeper into the language. The book is not a complete intro to programming. But with a little work you should be able to follow along even if you never programmed before. I highly suggest reading the book even if you’re an experienced JavaScript programmer or you’re coming from another language. You may be surprised how much you forgot about JavaScript.

What should I know before reading this book?

The “Little JavaScript Book” is not a complete guide to ES6, the 2015 JavaScript release. I assume the reader has some familiarity with ES6 but I’ll introduce it a bit in chapter 2. No worries though, I will explain ES6 syntax as we encounter it during the chapters.

Typographic conventions

This book uses JavaScript and you will find examples written inside a block:

```
1 function generateTableHead(table, data) {  
2   var thead = table.createTHead();  
3   var row = thead.insertRow();  
4   for (var i = 0; i < data.length; i++) {  
5     var th = document.createElement("th");  
6     var text = document.createTextNode(data[i]);  
7     th.appendChild(text);  
8     row.appendChild(th);  
9   }  
10 }
```

or inline: `th.appendChild(text)`. You will find both ES5 and ES6 code in the book, the former used for not overwhelming the reader at first, the latter used for refactoring later during the exercises.

What's the best way for following the examples?

I suggest experimenting hands-on with the code and not just copy-pasting. You can use whatever editor you prefer for writing HTML and JavaScript. Then you can run the pages in a browser. You can also use an online tool like [Js Bin](https://jsbin.com)¹ or [CodeSandbox](https://codesandbox.io/)².

Errata and other requests

This book is not a definitive guide to JavaScript and I might have missed something you'd like to read. The nice thing is that I can update the book whenever I want on Leanpub. If you want me to cover some topics in more depth feel free to drop me an email at valentino@valentinog.com.

About the author

I've been in love with computers since my mother gave me the first PC when I was 6 years old. In 1999 I was creating the first HTML pages and from there the web became my life. Today I help busy companies to embrace this crazy modern JavaScript stuff. I do training and consulting on JavaScript, React, Redux. Besides JavaScript I'm also active in the Python community. I serve as a coach for [Django Girls](https://djangogirls.org/)³ and I've spoke at Pycon Italia. Check out [my talks here](https://www.valentinog.com/talks)⁴. In my spare time you can find me in the countryside near Siena. If you want to get in touch for a beer feel free to ping me up at valentino@valentinog.com. Fancy reading some stuff about JavaScript or Python? Head over [my blog](https://www.valentinog.com/blog/)⁵!

¹<https://jsbin.com>

²<https://codesandbox.io/>

³<https://djangogirls.org/>

⁴<https://www.valentinog.com/talks>

⁵<https://www.valentinog.com/blog/>

Acknowledgments

This book would not have been possible without the clever questions I've got during the years from dozens of developers. A huge thanks goes also to all the people who helped shape the beta version of this book: Caterina Vicenti, Alessandro Muraro, Cristiano Bianchi, Irene Tomaini, Roberto Mossetto, Ricardo Antonio Piana, Alessio Forti, Valentino Pellegrino, Mauro De Falco, Sandro Cantagallo, Maurizio Zannoni.

Part 1 - JavaScript, the “hard” parts

Chapter 1. Getting started with JavaScript

What is JavaScript?

JavaScript is a scripting language for the web. Born in 1995 from Brendan Eich's hands, JavaScript was created for adding interactivity to web pages. At that time the web was still in its infancy and most of the fancy web pages we can see today were still a dream. Chased by his project manager, Brendan had only 10 days for creating a dynamic, flexible language that could run in a browser. He came up with JavaScript, a somewhat weird programming language which borrowed from Java (the cool kid at the time), C, and Scheme. JavaScript always had a bad reputation because it was full of quirks from the very beginning. But despite that JavaScript conquered a seat in the hall of fame and it's here to stay.

What can I build with JavaScript?

JavaScript is everywhere and it's used for almost everything. Born as a scripting language for the web is mostly used for adding interactivity to web pages: e.g. you can "listen" for clicks on HTML elements and react to the event with a JavaScript function. But besides these simple interactions JavaScript is also used for creating entire applications called SPA (single page applications). With the rise in usage the JavaScript ecosystem also saw a cambrian explosion of tools and libraries. Most of the frontend tools we use today are written in JavaScript: think of webpack, rollup, grunt, gulp, and friends. JavaScript is also used outside its natural habit. With Node.js we can create server-side and IoT applications, industrial appliances, and much more. But on top of that single page applications are one of the most prominent usage of JavaScript. What's a single page application? Take a look at the following HTML:

```
1 <!DOCTYPE html>
2 <html lang="en">
3
4 <head>
5     <meta charset="utf-8">
6     <title>My application</title>
7 <link href="main.css" rel="stylesheet"></head>
8
9 <body>
```

```
10     <div id="app">
11     </div>
12 <script type="text/javascript"
13     src="main.4a4c.js"
14     async="">
15 </script>
16 </html>
```

As you can see there is a script tag for loading a JavaScript file. The entire app is there. Single page applications are web apps built entirely with JavaScript. JavaScript takes over the application when the user visits the page and there will be no page refresh on subsequent interactions. From a user's perspective it's a huge improvement over classic applications. But with single page apps comes great responsibility: JavaScript is an heavy burden for most mobile devices and great care should be taken when designing and building JavaScript based web apps.

Why should I study JavaScript?

JavaScript has always been dismissed as a toy language, but the opposite is true. JavaScript has grown mature with the release of ECMAScript 2015 which brought a lot of nice improvements. The language is getting better and better year after year. Almost every frontend developer job opening requires JavaScript knowledge nowadays. And it's not a "basic jQuery knowledge" that hiring managers are looking for. Speaking of which, jQuery is slowly dying and that's a good thing for the web. It makes almost no sense to learn jQuery in 2019. Instead, frontend developers should know JavaScript quirks and be able to write idiomatic, well structured code, even with pure vanilla JavaScript. JavaScript is spreading fast and even if you don't like the language ignoring it at this point could even be detrimental to your career. Studying JavaScript today does not mean a shallow understanding of variables and functions: there is a lot more. An expert JavaScript developer knows closures, `this`, `new`, the prototype system, and code organization to name some. In this book you'll learn all the "hard parts" of JavaScript without the gimmicky packaging that comes with most of online tutorials and most important without arid academic lucubration like other JavaScript books. I hope this book will be an handy reference even for more experience developers looking forward to solidify their skills.

ECMAScript, ES5, ES6 and other technical terms

You'll find a lot of technical terms in the book. Expert developers know them all, but I don't want beginners to feel bad because they're afraid to ask what "API" means. Here's a brief review of the most important web-related terms.

API: stands for Advanced Programming Interface, but don't bother this strict definition: an API is like an USB socket, really. You, the developer, can interact with the socket and exchange data with

it. An API in programming is also a set of tools, a toolbox which is built by other developers and ready for use.

Native API: a native API is a built-in function available by default in a programming environment. Speaking of browsers for example we say that `document.querySelector()` is a native API for selecting HTML elements.

Browser API or Web API: like native APIs a Web API is a specific functionality available in a web browser. Developers can use these methods out of the box. Examples of Web API are `setTimeout`, `setInterval`, `console.log`. For a complete list check out [Web APIs](#)⁶.

ECMAScript: it is the official name for JavaScript. In 1996 JavaScript was donated to ECMA international, a third-party entity which takes care of defining standards for a lot of technology related things.

ES5: acronym for ECMAScript 2009, the fifth version of JavaScript. To avoid confusion it's more correct to say ECMAScript + year for denoting a specific JavaScript version.

ES6: stands for ECMAScript 2015, the sixth version of JavaScript. Since 2015 the JavaScript committee decided to release new features yearly. From there we had ECMAScript 2016, ECMAScript 2017, ECMAScript 2018, and so on.

JavaScript engine: is part of the browser and is able to compile and interpret JavaScript code. Browser vendors build JavaScript engines by following (sometimes not so strictly) a document called JavaScript specification.

JavaScript specification: is a formal, written document which outlines how the JavaScript language should behave. Browser vendors read the spec and implement JavaScript engines in a way that JavaScript code is executed as the spec prescribes.

HTTP request: is the act of “talking” to a remote web server for fetching or saving data. Most of the times the data is exchanged in JSON format.

HTTP error: sometimes things don't go well when talking to web services and the server may respond with an error. Errors are denoted with a numeric code: some common errors are 500 (server error), 404 (not found), 403 (forbidden), and so on.

JSON: JSON stands for JavaScript Object Notation, a format for exchanging data between web services and JavaScript applications.

REST API: is a web service (local or remote) which exposes data. A REST API makes possible to interact with an underlying database for saving or retrieving entities.

Transpiler: older browsers do not support modern JavaScript syntax from ECMAScript 2015 and above. A transpiler is a tool which takes modern JavaScript syntax and spits out a more compatible version (ECMAScript 2009).

Vanilla JavaScript: vanilla JavaScript is a term for denoting “pure” JavaScript applications, i.e. those written without the help of a frontend library like React, Vue, or Angular.

⁶<https://developer.mozilla.org/en-US/docs/Web/API>

AJAX: a set of technologies for fetching data in the browser without causing a page refresh. The acronym stands for “Asynchronous JavaScript and XML”, coined in 1999.

A special note on ES2015

ECMAScript 2015 (ES6) is one of the most important releases ever for JavaScript. It introduced a lot of improvements into the language and in this book you’ll find the most important ECMAScript 2015 features like Promise, arrow functions, class, and more. I wish I could go through every tiny bit of ECMAScript 2015 here, but another book would not be enough to cover them all. Instead I took another path and I decided to include just the most relevant features strategically, here and there during the chapters. If you want to learn ECMAScript 2015 in depth after reading this book I highly suggest one of my favourite essay on the subject: [Exploring ES6](https://exploringjs.com/es6/)⁷ by Axel Rauschmayer.

Conclusions

JavaScript was born in 1995 as a scripting language for the web. Since then it gained massive traction and today we can say with certainty that JavaScript has become the lingua franca of the web. JavaScript is used in many environments and developers well versed in this language are highly paid nowadays. But jQuery, one of the most used JavaScript library back in the days is not enough anymore as a skill. JavaScript developers should know the intricacies of JavaScript and being able to structure the code well, other than being able to understand legacy JavaScript applications.



Sharpen up your JavaScript skills

- What is an API?
- What is JavaScript used for?
- How many versions of JavaScript have been released since 1995?

⁷<https://exploringjs.com/es6/>

Chapter 2. Prologue: JavaScript fundamentals

Strings, numbers, variables and all in between

The first draft of this book was aiming at experienced JavaScript developers who wanted to refresh their skills. I utterly skipped JavaScript fundamentals (my bad) and later on based on some precious feedback I decided to add a quick re-introduction to the language. So here we are! JavaScript builds on seven lego bricks called “types”. The complete list is:

- String
- Number
- Boolean
- Null
- Undefined
- Object
- Symbol (part of ECMAScript 2015)

Except Object which is a type, everything else is also called “primitive” of the language. Every JavaScript type has a corresponding representation which can be used in our code, like strings:

```
1 var string = "Hello John";
```

Numbers:

```
1 var age = 33;
```

Speaking of which, JavaScript has arithmetic operations too:

- | | |
|----|--|
| + | sum |
| ++ | increment |
| * | multiplication |
| ** | exponent (part of ECMAScript 2015) |
| - | subtraction |
| -- | decrement |
| / | division |
| % | modulo (returns the remainder of a division) |

Then there are booleans:

```
1 var not = false;
```

and objects:

```
1 var obj = {  
2   name: "John",  
3   age: 33  
4 };
```

Values can be stored in a variable with the `var` keyword, the most compatible way for declaring variables:

```
1 var greet = "Hello";  
2 var year = 89;  
3 var not = false;
```

I said compatible because with ECMAScript 2015 we have two other options: `let` and `const`. Older browsers may not support these new keywords and unless using a “transpiler” (see chapter 1 for terminology) you may run into errors. Besides stand-alone variables it is also possible to declare list of entities, the array:

```
1 var array = ["Hello", 89, false, true];
```

Array elements are accessible by an index, starting at 0:

```
1 var array = ["Hello", 89, false, true];  
2 var first = array[0]; // Holds "Hello"
```

Almost all JavaScript entities have some functions attached, called methods. To name two, array has a lot of methods for manipulating itself:

```
1 var array = ["Hello", 89, false, true];  
2  
3 array.push(99);  
4 array.shift();  
5  
6 console.log(array); // [ 89, false, true, 99 ];
```

And the same is true for strings:

```
1 var string = "John";
2 console.log(string.toUpperCase()); // JOHN
```

Later in chapter 5 you'll see where these methods come from, but here's a spoiler: they're defined on `Array.prototype` and `String.prototype` respectively.



The more you know

JavaScript strings are immutable. String's methods always return a new string leaving the original unaltered.

Alongside with methods there are also properties which are useful for extracting info about the length of a string for example:

```
1 var string = "John";
2 console.log(string.length); // 4
```

or an array's length:

```
1 var array = ["Hello", 89, false, true];
2
3 array.push(99);
4 array.shift();
5
6 console.log(array.length); // 4
```

These properties are somewhat special because they're called "getters"/"setters". Most JavaScript types are objects under the hood (more on this later in the book). You can imagine a given string like an object with a bunch of methods and properties attached. When accessing the length of an array you're just calling the corresponding getter. The setter function kicks in for the set operation:

```
1 var array = {
2   value: ["Hello", 89, false, true],
3   push: function(element) {
4     //
5   },
6   shift: function() {
7     //
8   },
9   get length() {
10    // gets the length
11  },
```



```
12     set length(newLen) {
13         // sets the length
14     }
15 };
16
17 // Getter call
18 var len = array.length
19
20 // Setter call
21 array.length = 50;
```

And now having laid down the fundamentals let's take a closer look at Object, one of the most important JavaScript type.

Standing on the shoulders of a giant Object

Object is the most important type in JavaScript so that almost every other entity derives from it. Functions and array for example are specialized objects. Objects in JavaScript are container for key/value pairs, like the following example (literal form):

```
1  var obj = {
2      name: "John",
3      age: 33
4  };
```

There is also another way for creating objects but it's rare and slow, avoid it:

```
1  var obj = new Object({
2      name: "John",
3      age: 33
4  });
```

As you can see objects are a convenient way for holding values which can be retrieved later by accessing the corresponding property:

```
1  var obj = {
2    name: "John",
3    age: 33
4  };
5
6  console.log(obj.name); // "John"
```

We can also add new properties, delete, or change them:

```
1  var obj = {
2    name: "John",
3    age: 33
4  };
5
6  obj.address = "Some address";
7  delete obj.name;
8  obj.age = 89;
```

Object's keys can also be strings and in this case we access the property with bracket notation:

```
1  var obj = {
2    name: "John",
3    age: 33,
4    "complex key": "stuff"
5  };
6
7  console.log(obj["complex key"]); // "stuff"
```

The dot notation however is more common and unless the key is a complex string you should prefer the traditional property access:

```
1  var obj = {
2    name: "John",
3    age: 33
4  };
5
6  // Dot notation, more common
7  console.log(obj.name); // "John"
```

over the bracket notation:

```
1  var obj = {  
2    name: "John",  
3    age: 33  
4  };  
5  
6  // Bracket notation  
7  // Not so common if the key is a one word string  
8  console.log(obj["name"]); // "John"
```

That's all the basic you need to know, but in chapter 5 we'll see that JavaScript objects are really powerful and can do a lot more. Now let's take a look at JavaScript functions.

50 shades of JavaScript functions

Almost every programming language has functions and JavaScript makes no exception. Functions are reusable piece of code. Consider the following examples:

```
1  function hello(message) {  
2    console.log(message);  
3  }  
4  
5  hello("Hello");
```

and

```
1  function sum(a, b) {  
2    return a + b;  
3  }  
4  
5  var sum = sum(2, 6);
```

The first function prints a string while the second returns a value to the external world. As you can see functions can accept parameters, listed in the function “signature”:

```
1  // a and b are parameters in the function's signature  
2  function sum(a, b) {  
3    return a + b;  
4  }
```

We can pass values when invoking the function, and in that context they take the name of arguments:

```
1 // a and b are parameters in the function's signature
2 function sum(a, b) {
3     return a + b;
4 }
5
6 // 2 and 6 are arguments for the function
7 var sum = sum(2, 6);
```

A JavaScript function declared with the `function` keyword is a regular function, opposed to arrow functions which don't have a traditional body (more on arrow functions later). Regular functions can assume many shapes:

- named function
- anonymous function
- object method
- object method shorthand (ECMAScript 2015)
- IIFE (immediately invoked function expression)

Named functions are the most traditional type of function:

```
1 // A named function
2 function sum(a, b) {
3     return a + b;
4 }
```

Anonymous function on the other hand don't have a name and can be assigned to a variable for later use:

```
1 var sum = function(a, b) {
2     return a + b;
3 };
```

or used as callbacks (in the next chapter you'll see callbacks in depth) inside other functions:

```
1 var button = document.createElement("button");
2
3 button.addEventListener("click", function(event) {
4     // do stuff
5 });
```

Functions can also live inside JavaScript objects. If so we call the function (for good or bad) a method of that object:

```
1 var widget = {
2   showModal: function() {
3     // do stuff
4   }
5 };
6
7 widget.showModal();
```

Regular functions also get by default a `this` keyword which can assume different meanings depending on how the function is called. In chapter 6 we'll explore the topic in detail. For now take for granted a simple rule: a function running inside an object has `this` pointing to the host object:

```
1 var widget = {
2   html: "<div></div>",
3   showModal: function() {
4     console.log(this.html);
5   }
6 };
7
8 widget.showModal(); // "<div></div>"
```

In ECMAScript 2015 you can also use object method shorthand and write methods like so:

```
1 var widget = {
2   showModal() {
3     // object method shorthand
4   }
5 };
6
7 widget.showModal();
```

Last but not least there are IIFE (immediately invoked function expression). An IIFE is a function which immediately calls itself:

```
1 var IIFE = (function() {
2   // what happens in an IIFE stays in the IIFE
3 })();
```

The syntax may look a bit weird, but IIFE are really powerful as you'll see them in chapter 4. Alongside with regular functions there are also arrow function, added in ECMAScript 2015. Arrow functions don't use the `function` keyword, but they come in a similar variety of shapes:

- named arrow function
- anonymous arrow function
- object method (not so useful)
- IIFE arrow function (immediately invoked function expression)

Arrow functions are convenient, but I suggest not overusing them. Here's a named arrow function to start with. We can omit the `return` statement and use parenthesis if there are no parameters:

```
1  const arrow = () => console.log("Silly me");
```

If you need to compute something into the arrow function or maybe call other functions you can open a body with curly braces:

```
1  const arrow = () => {  
2    const a = callMe();  
3    const b = callYou();  
4    return a + b;  
5  };
```

Curly braces are also the literal form for defining a JavaScript object and that does not mean you can do something like:

```
1  const arrow = () => {  
2    a : "hello",  
3    b : "world"  
4  };
```

It's invalid syntax and won't ever compile. For returning an object from an arrow function you can use parenthesis:

```
1  const arrow = () => ({  
2    a: "hello",  
3    b: "world"  
4  });  
5  
6  console.log(arrow());  
7  // { a: 'hello', b: 'world' }
```

Or much better the `return` statement:

```
1  const arrow = () => {  
2    return {  
3      a: "hello",  
4      b: "world"  
5    };  
6  };
```



"I suggest replacing all regular function with arrow functions"

No, you don't. If you hear this suggestion from some colleague of yours, please think twice. Arrow functions are handy and more concise, but that doesn't mean they're more readable in the codebase, especially for junior developers.

Like regular anonymous functions there are also anonymous arrow functions. Here's one passed as a callback to another function:

```
1  const arr = [1, 2, 3];  
2  
3  const res = arr.map(element => element + 1);  
4  
5  console.log(res); // [ 2, 3, 4 ]
```

It takes `element` as a parameter and returns `element + 1` for each array element. As you see if there is a single parameter for an arrow function there is no need to put parenthesis around it:

```
1  const fun = singleParameter => singleParameter + 1;
```

But if you need more parameters the parenthesis are required:

```
1  const fun = (a, b) => a + b + 1;
```

Arrow functions can also appear as objects method, but they behave differently from a regular function. I introduced the `this` keyword some paragraph ago, a reference to the object in which a function is running. When called as an object method, regular function point `this` to the "host" object:

```
1 var widget = {  
2   html: "<div></div>",  
3   showModal: function() {  
4     console.log(this.html);  
5   }  
6 };  
7  
8 widget.showModal(); // "<div></div>"
```

Arrow functions instead have this pointing to something totally different:

```
1 var widget = {  
2   html: "<div></div>",  
3   showModal: () => console.log(this.html)  
4 };  
5  
6 widget.showModal(); // undefined
```

Surprise! For that reason arrow function are not well suited as objects method, but there are interesting use cases for them and during the book we'll see why and when to use them effectively. To conclude let's see IIFE arrow functions:

```
1 (() => {  
2   console.log("aa");  
3 })();
```

Perfectly, valid, confusing syntax don't you think? And now let's move on to the next chapter!

Conclusions

JavaScript has seven fundamental building blocks called “types”, of which six are known also as “primitives”. Object is a type on it's own, being also the most important entity of the language. Objects are containers for pairs of keys/values and can contain almost every other JavaScript primitive, including functions. Like most other programming languages JavaScript has strings, numbers, functions, booleans and a couple of special types called Null and Undefined. There are two species of functions in JavaScript: arrow functions (added in ECMAScript 2015) and regular functions. Both have their use cases and you'll learn with the experience when and how to use one instead of the other.



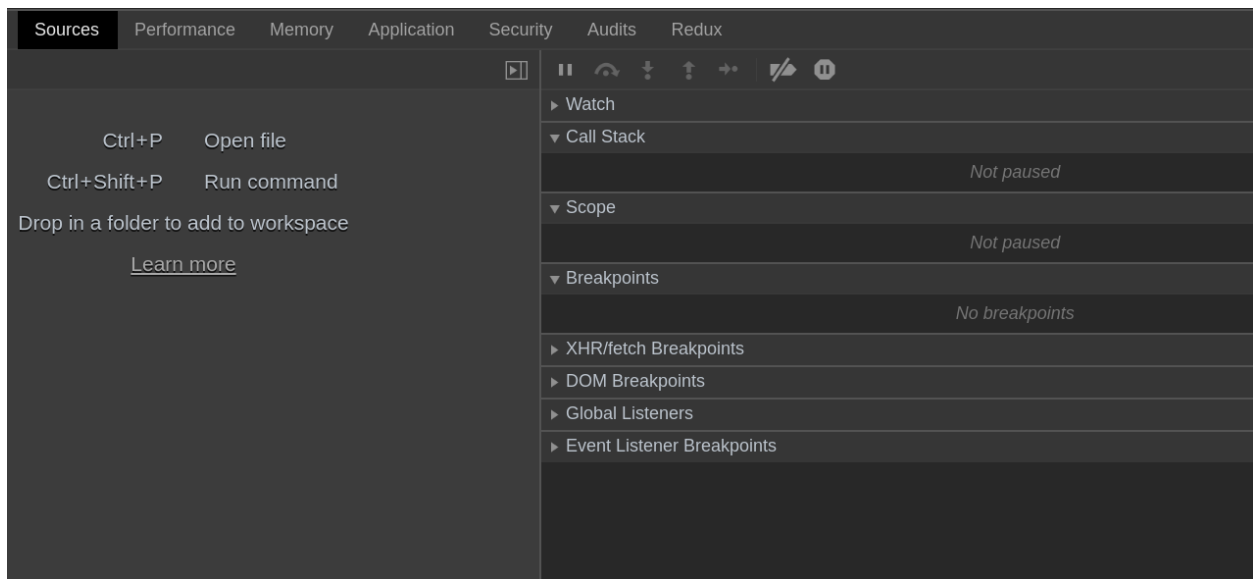
Sharpen up your JavaScript skills

- What's the difference between arguments and parameters?
- How many primitives there are in JavaScript?
- What's a regular arrow function?
- What special keyword a function has access to when it runs as an object method?
- How many ways there are for declaring variables in JavaScript?

Chapter 3. A whirlwind tour of JavaScript engines

Let's begin our immersion in the language by introducing the wonderful world of JavaScript engines.

Ever wondered how browsers read and run JavaScript code? It seems magic but you can get an hint of what's happening under the hood. Open up a browser console in Chrome and take a look at the Sources tab. You'll see some boxes, one of the more interesting named Call Stack (in Firefox you can see the Call Stack after inserting a breakpoint into the code):



Call Stack

What's a Call Stack? Looks like there is a lot of stuff going on, even for running a couple line of code. JavaScript in fact does not come out of the box with every web browser. There is a big component which compiles and interprets our JavaScript code: it's the JavaScript engine. The most popular JavaScript engines are V8, used by Google Chrome and Node.js, SpiderMonkey for Firefox, and JavaScriptCore, used by Safari/WebKit. JavaScript engines today are brilliant pieces of engineering and it would be impossible to cover every single facet of them. But there are some smaller pieces in every engine doing the hard work for us. One of these component is the Call Stack and alongside together with Global Memory and Execution Context make possible to run our code. Ready to meet them?

JavaScript engines and Global Memory

I said that JavaScript is both a compiled and an interpreted language at the same time. Believe it or not JavaScript engines actually compile your code mere microseconds before executing it. That sounds magic right? The magic is called JIT (Just in time compilation). It's a big topic on its own, another book would not be enough to describe how JIT works. But for now we can just skip the theory behind compilation and focus on the execution phase, which is nonetheless interesting.

To start off consider the following code:

```
1  var num = 2;
2
3  function pow(num) {
4      return num * num;
5  }
```

If I were to ask you how does the above code is processed in the browser? What will you say? You might say “the browser reads the code” or “the browser executes the code”. The reality is more nuanced than that. First, it's not the browser that reads that snippet of code. It's the engine. A JavaScript engine reads the code and as soon as it encounters the first line it puts a couple of references into the Global Memory.

The Global Memory (also called Heap) is an area where the JavaScript engine saves variables and function declarations. Pay attention, the difference might sound trivial but function declarations in JavaScript are not the same as function invocations. A function declaration is just a description of what the function should accept and how it's going to be invoked. A function invocation on the other hand is the actual execution of a previously declared function.

So, back to our example, when the engine reads the above code the Global Memory is populated with two bindings:



Global Memory in a JavaScript engine

At this point nothing is executed but what if we try to run our function like so:

```
1 var num = 2;  
2  
3 function pow(num) {  
4     return num * num;  
5 }  
6  
7 pow(num);
```

What will happen? Now things get interesting. When a function is called the JavaScript engine makes room for two more boxes:

- a Global Execution Context
- a Call Stack

Let's see what they are in the next section.



The more you know ...

Even if the example has just a variable and a function consider that your JavaScript code runs in a more bigger environment: in the browser or in Node.js. In these environment there are a lot of pre-defined functions and variables, called globals. The Global Memory will hold a lot more than `num` and `pow`.

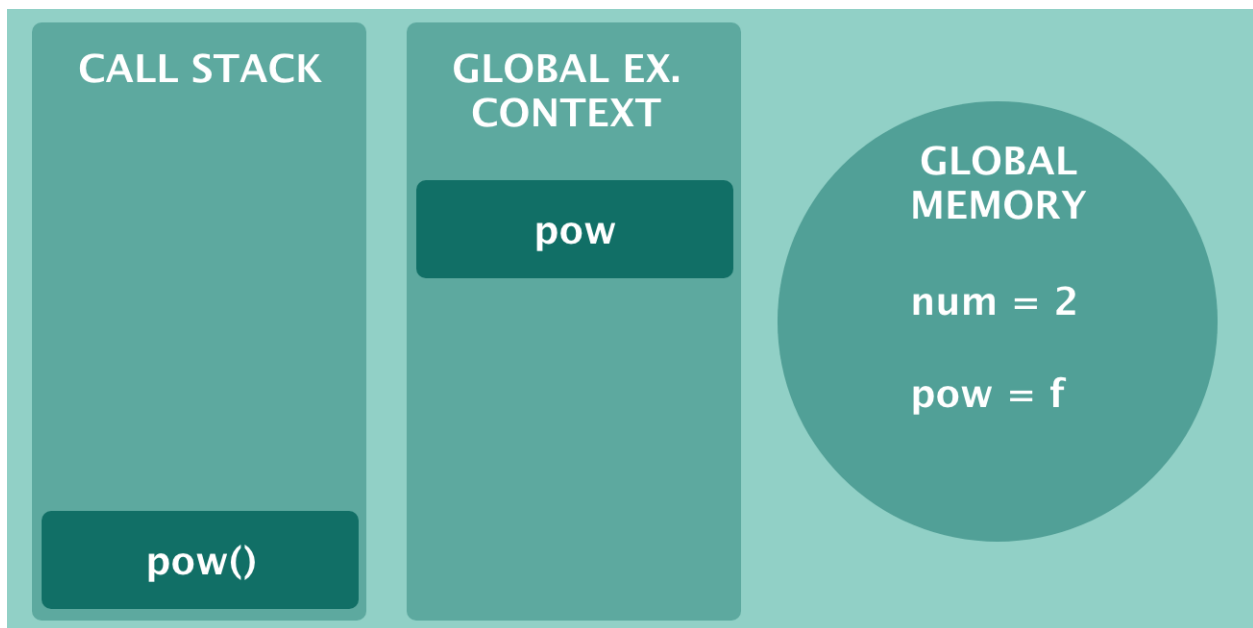
Global Execution Context and Call Stack

You learned how the JavaScript engine reads variables and function declarations. They end up in a Global Memory (the Heap). But now we executed a JavaScript function and the engine has to take care of it. How? There is a fundamental component in every JavaScript engine, called Call Stack. The Call Stack is a stack data structure: that means elements can enter from the top but they can't leave if there's some element above them. JavaScript functions are exactly like that. Once executing they can't leave the Call Stack if some other function remains stuck. Pay attention because this concept is helpful for wrapping your head around the sentence "JavaScript is single-threaded". But for now let's get back to our example. When the function is called the engine pushes that function inside the call stack:



Call Stack and Global Memory in a JavaScript engine

I like thinking of the Call Stack as a pile of Pringles. We cannot eat a pringle at the bottom of the pile without first eating all the pringles at the top! Luckily our function is synchronous: it's a simple multiplication and it's calculated quickly. At the very same time the engine allocates also a Global Execution Context, which is the global environment where our JavaScript code runs. Here's how it looks like:



Call Stack, Global Memory, and Global Execution Context in a JavaScript engine

Imagine the Global Execution Context as a sea where JavaScript global functions swim like fishes. How nice! But that's just the half of the story. What if our function has some nested variables or one or more inner functions? Even in a simple variation like the following, the JavaScript engine creates a Local Execution Context:

```

1  var num = 2;
2
3  function pow(num) {
4      var fixed = 89;
5      return num * num;
6  }
7
8  pow(num);

```

Notice that I added a variable named `fixed` inside the function `pow`. In that case the Local Execution Context will contain a box for holding `fixed`. I am not so good at drawing little tiny boxes inside other tiny boxes! Use your imagination for now. The Local Execution Context will appear near `pow`, inside the greener box contained in the Global Execution Context. You can also imagine that for every nested function of a nested function the engine creates more Local Execution Contexts. These boxes can go to far so quickly! Like a matrioska! Now how about getting back to that single-threaded story? What does it means?

We say that JavaScript is single-threaded because there is a single Call Stack handling our functions. That is, functions can't leave the Call Stack if there are other functions waiting for execution. That's not a problem when dealing with synchronous code. For example, a sum between two numbers is

synchronous and runs in microseconds. But how about network calls and other interactions with the outside world? Luckily JavaScript engines are designed to be asynchronous by default. Even if they can execute one function at a time there is a way for slower function to be executed by an external entity: the browser in our case. We'll explore this topic later.

In the meantime you learned that when a browser loads some JavaScript code an engine reads line by line and performs the following steps:

- populates the Global Memory (Heap) with variables and function declarations
- pushes every function invocation into a Call Stack
- creates a Global Execution Context in which global functions are executed
- creates lot of tiny Local Execution Contexts (if there are inner variables or nested functions)

You should have by now a big picture of the synchronous mechanic at the base of every JavaScript engine. In the next sections you'll see how asynchronous code works in JavaScript and why it works that way.



The more you know ...

JavaScript engines are single-threaded but modern browser are able to launch more than one background thread for running multiple operations in parallel. However that does not change the behavior of JavaScript, which is and will remain single-threaded.

Asynchronous JavaScript, Callback Queue and the Event Loop

Global Memory, Execution Context and Call Stack explain how synchronous JavaScript code runs in our browser. Yet we're missing something. What happens when there is some asynchronous function to run? By asynchronous function I mean every interaction with the outside world that could take some time to complete. Calling a REST API or invoking a timer are asynchronous because they can take seconds to run. With the elements we have so far in the engine there is no way to handle that kind of functions without blocking the Call Stack, and so the browser.



What is a REST API?

REST APIs are services exposing data and information over the internet. Developers can get these data through the wire with an HTTP request. An example of REST API is jsonplaceholder.typicode.com⁸.

Remember, the Call Stack can execute one function at a time and even one blocking function can literally freeze the browser. Luckily JavaScript engines are smart and with a bit of help from the

⁸<https://jsonplaceholder.typicode.com/posts>

browser can sort things out. When we run an asynchronous function like `setTimeout` the browser takes that function and runs it for us. Consider a timer like the following:

```
1 setTimeout(callback, 10000);
2
3 function callback(){
4     console.log('hello timer!');
5 }
```

I'm sure you saw `setTimeout` hundred of times yet you might not know that it's not a built-in JavaScript function. That is, when JavaScript was born there were no `setTimeout` built into the language. `setTimeout` in fact is part of the so called Browser APIs, a collection of handy tool that the browser gives us for free. How kind! What does it mean in practice? Since `setTimeout` is a Browser APIs that function is run straight by the browser (it appears for a moment into the Call Stack but is removed instantly). Then after 10 seconds the browsers takes the callback function we passed in and moves it into a Callback Queue.



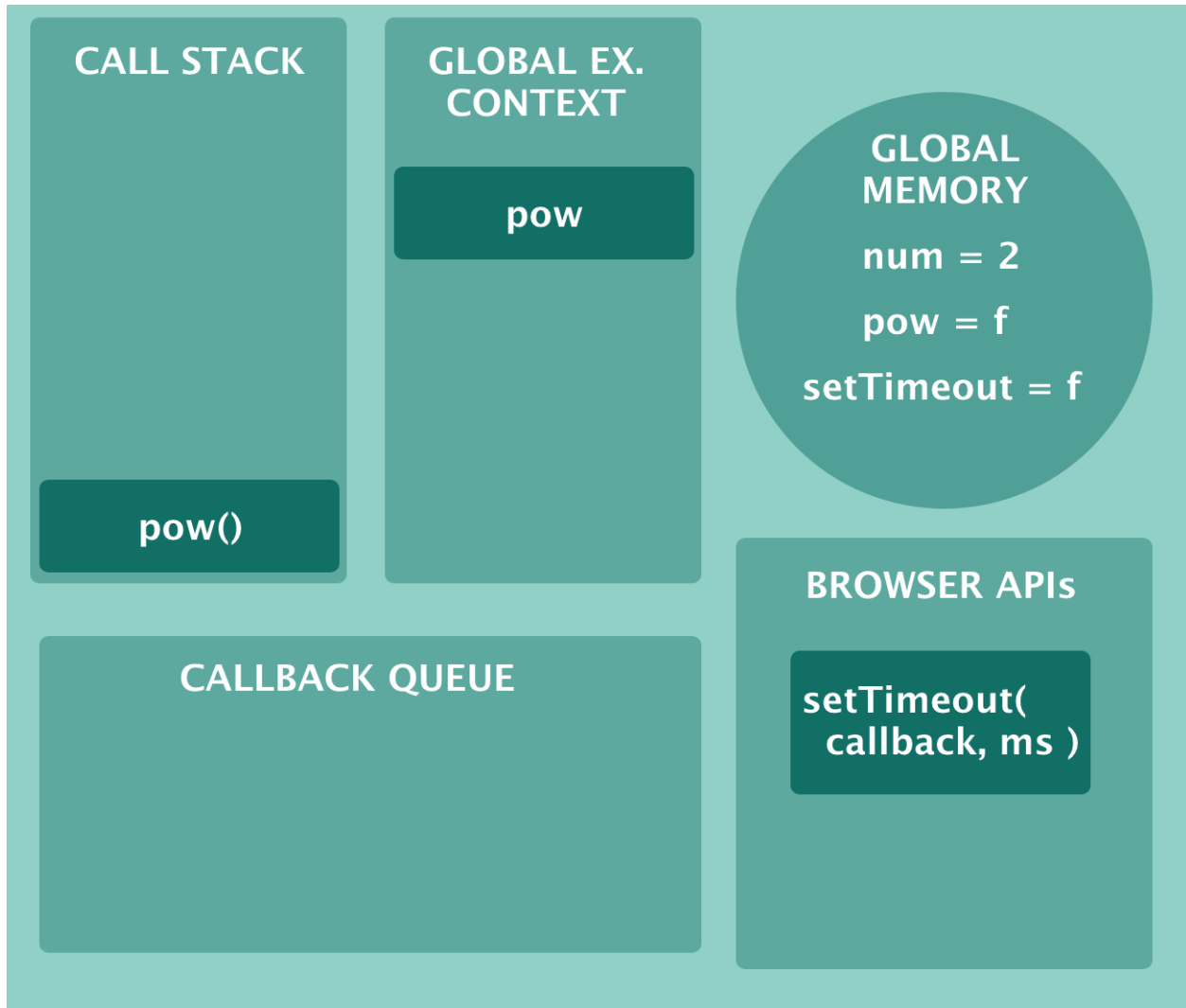
What is a callback function?

Callback functions in JavaScript are functions passed as arguments to other functions. Since functions in JavaScript are objects we can pass them as values inside other functions. When a function takes another function as input we call that an High order function. The input function is the callback. Callback functions are also used for flagging asynchronous code in JavaScript. It means literally: call me back later when you're done.

At this point we have two more boxes inside our JavaScript engine. If you consider the following code:

```
1 var num = 2;
2
3 function pow(num) {
4     return num * num;
5 }
6
7 pow(num);
8
9 setTimeout(callback, 10000);
10
11 function callback(){
12     console.log('hello timer!');
13 }
```

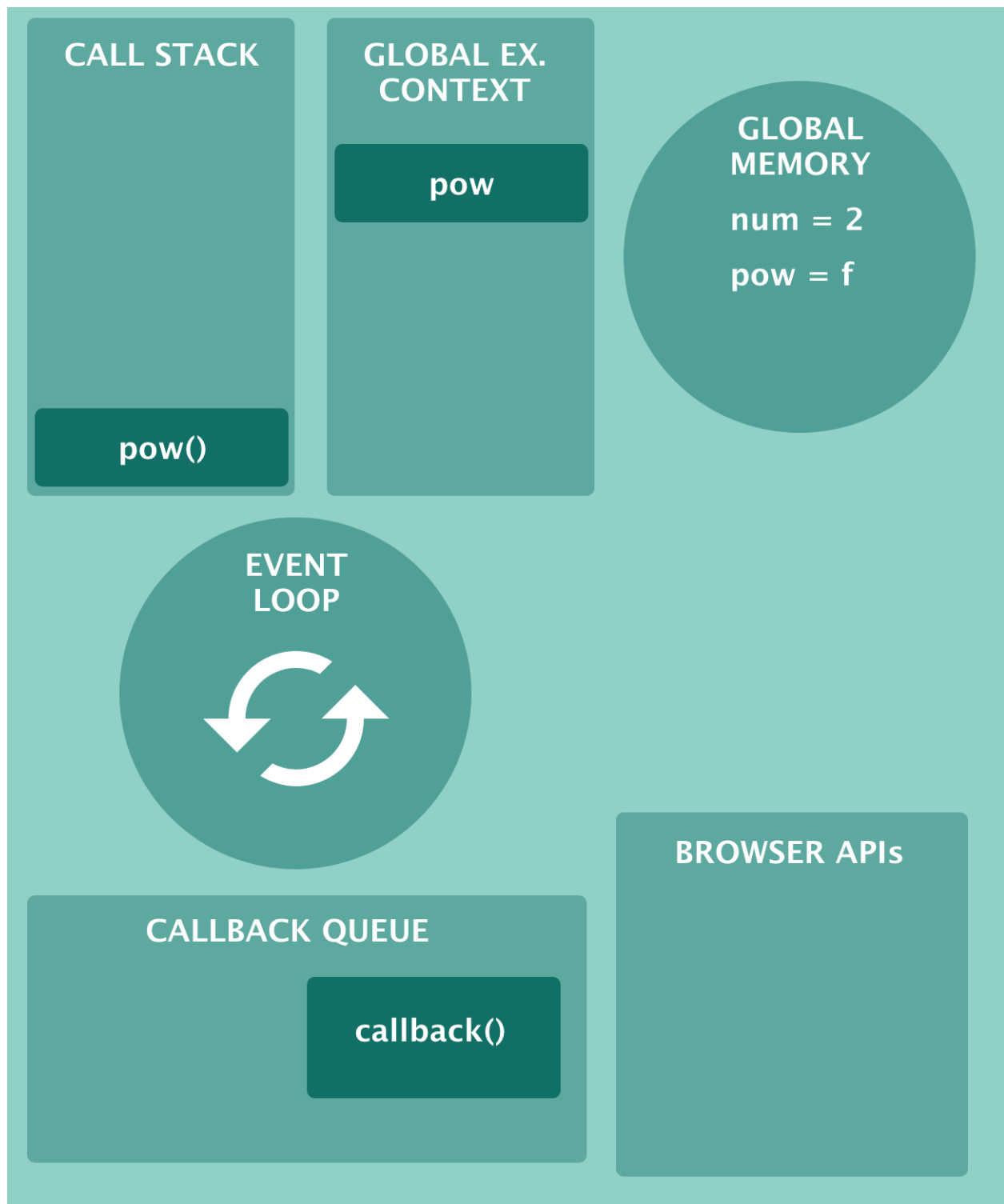

We can complete our illustration like so:



More boxes: Callback Queue and Browser APIs

As you can see `setTimeout` runs inside the browser context. After 10 seconds the timer is triggered and the callback function is ready to run. But first it has to go through the Callback Queue. The Callback Queue is a queue data structure and as its name suggest is an ordered queue of functions. Every asynchronous function must pass through the Callback Queue before is pushed into the Call Stack. But who pushes that function forward? There is another component named Event Loop.

The Event Loop has just one job for now: it should check whether the Call Stack is empty. If there is some function into the Callback Queue and if the Call Stack is free, then it's time to push the callback into the Call Stack. Once done the function is executed. This is the big picture of a JavaScript engine for handling asynchronous and synchronous code:



More boxes: Callback Queue, Browser APIs, Event Loop

Imagine that `callback()` is ready to be executed. When `pow()` finishes the Call Stack is empty and the Event Loop pushes `callback()` in. That's it! If you understand the illustration above then you are

ready to understand all the JavaScript. Remember: Browser APIs, Callback Queue, and Event Loop are the pillars of asynchronous JavaScript. And if you fancy videos I suggest watching [What the heck is the event loop anyway](#)⁹ by Philip Roberts. It's one of the best explanation ever of the Event Loop. Hold on though because we're not done with asynchronous JavaScript. In the next sections we'll take a closer look at ES6 Promises.

Callback hell and ES6 Promises

Callback functions are everywhere in JavaScript. They are used both for synchronous and asynchronous code. Consider the `map` method for example:

```
1 function mapper(element){
2     return element * 2;
3 }
4
5 [1, 2, 3, 4, 5].map(mapper);
```

`mapper` is a callback function passed inside `map`. The above code is synchronous. But consider instead an interval:

```
1 function runMeEvery(){
2     console.log('Ran!');
3 }
4
5 setInterval(runMeEvery, 5000);
```

That code is asynchronous, yet as you can see we pass the callback `runMeEvery` inside `setInterval`. Callbacks are pervasive in JavaScript so that during the years a problem emerged: callback hell. Callback hell in JavaScript refers to a “style” of programming where callbacks are nested inside callbacks which are nested ... inside other callbacks. Due to the asynchronous nature of JavaScript programmers fell into this trap over the years. To be honest I never run into extreme callback pyramids, maybe because I value readable code and I always try sticking to that principle. If you end up in a callback hell it's a sign that your function is doing too much.

I won't cover callback hell here, if you're curious there is a website, [callbackhell.com](#)¹⁰ which explores the problem in more detail and offers some solutions. What we want to focus on now are ES6 Promises. ES6 Promises are an addition to the JavaScript language aiming to solve the dreaded callback hell. But what is a Promise anyway?

A Promise in JavaScript is the representation of a future event. A Promise can end with success: in jargon we say it's resolved (fulfilled). But if the Promise errors out we say it's in a rejected state.

⁹<https://www.youtube.com/watch?v=8aGhZQkoFbQ>

¹⁰<http://callbackhell.com/>

Promises have also a default state: every new Promise starts in pending state. It is possible to create your own Promise by calling the Promise constructor and passing a callback function into it. The callback function can take two parameters: resolve and reject. Let's create a new Promise which will resolve in 5 seconds (you can try the examples in a browser's console):

```
1  const myPromise = new Promise(function(resolve){
2      setTimeout(function(){
3          resolve()
4      }, 5000)
5  });
```

As you can see resolve is a function that we call for making the Promise succeed. Reject on the other hand makes a rejected Promise:

```
1  const myPromise = new Promise(function(resolve, reject){
2      setTimeout(function(){
3          reject()
4      }, 5000)
5  });
```

Note that in the first example you can omit reject because it's the second parameter. But if you intend to use reject you can't omit resolve. In other words the following code won't work and will end up in a resolved Promise:

```
1  // Can't omit resolve !
2
3  const myPromise = new Promise(function(reject){
4      setTimeout(function(){
5          reject()
6      }, 5000)
7  });
```

Now, Promises don't look so useful isn't it? Those example print nothing to the user. Let's add some data to the mix. Both resolved and rejected Promises can return data. Here's an example:

```
1  const myPromise = new Promise(function(resolve) {
2      resolve([ { name: "Chris" } ]);
3  });
```

But still we can't see any data. For extracting data from a Promise you need to chain a method called then. It takes a callback (the irony!) which receives the actual data:

```
1  const myPromise = new Promise(function(resolve, reject) {
2    resolve([ { name: "Chris" } ]);
3  });
4
5  myPromise.then(function(data) {
6    console.log(data);
7  });
```

As a JavaScript developer and consumer of other's people code you will mostly interact with Promises from the outside. Library creators instead are more likely to wrap legacy code inside a Promise constructor like so:

```
1  const shinyNewUtil = new Promise(function(resolve, reject) {
2    // do stuff and resolve
3    // or reject
4  });
```

When in need we can also create and resolve a Promise in place by calling `Promise.resolve()`:

```
1  Promise.resolve({ msg: 'Resolve!' })
2  .then(msg => console.log(msg));
```

So to recap a JavaScript Promise is a bookmark for an event happening in the future. The event starts in a pending state and can either succeed (resolved, fulfilled) or fail (rejected). A Promise can return data and that data can be extracted by attaching `then` to the Promise. In the next section we'll see how to deal with errors coming from a Promise.

ES6 Promises and error handling

Error handling in JavaScript has been always straightforward, at least for synchronous code. Consider the following example:

```
1 function makeAnError() {
2   throw Error("Sorry mate!");
3 }
4
5 try {
6   makeAnError();
7 } catch (error) {
8   console.log("Catching the error! " + error);
9 }
```

The output will be:

```
1 Catching the error! Error: Sorry mate!
```

The error got in the catch block as expected. Now let's try with an asynchronous function:

```
1 function makeAnError() {
2   throw Error("Sorry mate!");
3 }
4
5 try {
6   setTimeout(makeAnError, 5000);
7 } catch (error) {
8   console.log("Catching the error! " + error);
9 }
```

The above code is asynchronous because of `setTimeout`. What happens if we run it?

```
1   throw Error("Sorry mate!");
2   ^
3
4 Error: Sorry mate!
5     at Timeout.makeAnError [as _onTimeout] (/home/valentino/Code/piccolo-javascript/\
6 async.js:2:9)
```

This time the output is different. The error didn't go through the catch block. It was free to propagate up in the stack. That's because try/catch only works with synchronous code. If you're curious the problem is explained in great detail in [Error Handling in Node.js](https://www.joyent.com/node-js/production/design/errors)¹¹. Luckily with Promises there is a way to handle asynchronous errors like they were synchronous. If you recall from the previous section a call to `reject` is what makes a rejected Promise:

¹¹<https://www.joyent.com/node-js/production/design/errors>

```
1 const myPromise = new Promise(function(resolve, reject) {  
2   reject('Errored, sorry!');  
3 });
```

In the above case we can handle the error with the catch handler, taking (again) a callback:

```
1 const myPromise = new Promise(function(resolve, reject) {  
2   reject('Errored, sorry!');  
3 });  
4  
5 myPromise.catch(err => console.log(err));
```

We can also call `Promise.reject()` for creating and rejecting a Promise in place:

```
1 Promise.reject({msg: 'Rejected!'}).catch(err => console.log(err));
```

To recap: the then handler runs when a Promise is fulfilled while the catch handler runs for rejected Promises. But that's not the end of the story. Later we will see how `async/await` works nicely with `try/catch`.

ES6 Promises combinators: `Promise.all`, `Promise.allSettled`, `Promise.any`, and friends

Promises are not meant to go alone. The Promise API offers a bunch of methods for combining Promises together. One of the most useful is `Promise.all` which takes an array of Promises and returns a single Promise. The problem is that `Promise.all` rejects if any Promise in the array is rejected.

`Promise.race` resolves or rejects as soon as one of the Promise in the array is settled. It still rejects if one of the Promise rejects.

Newer versions of V8 are also going to implement two new combinators: `Promise.allSettled` and `Promise.any`. `Promise.any` is still in the early stages of the proposal: at the time of this writing there is still no support for it. But the theory is that `Promise.any` can signal whether any of the Promise is fulfilled. The difference from `Promise.race` is that `Promise.any` does not reject even if one of the Promise is rejected.

Anyway the most interesting of the two is `Promise.allSettled`. It still takes an array of Promises but it does not short-circuit if one of the Promise rejects. It is useful for when you want to check if an array of Promises is all settled, regardless of an eventual rejection. Think of it as a counterpart of `Promise.all`.

ES6 Promises and Microtask Queue

If you remember from the previous sections every asynchronous callback functions ends up in the Callback Queue before being pushed into the Call Stack. But now try to execute the following code:

```
1 console.log("Start");
2
3 setTimeout(() => {
4   console.log("Log me!");
5 }, 0);
6
7 const myPromise = new Promise(function(resolve, reject) {
8   resolve([ { name: "Chris" } ]);
9 });
10
11 myPromise.then(data => console.log(data));
12
13 console.log("End");
```

What the output could be? Our Promise has no delay so it should return alongside with Log me!. But we'll get:

```
1 Start
2 End
3 [ { name: 'Chris' } ]
4 Log me!
```

That's because callbacks function passed in a Promise have a different fate: they are handled by the Microtask Queue. And there's an interesting quirk you should be aware of: Microtask Queue has precedence over the Callback Queue. When the Event Loop checks if there is any new callback ready to be pushed into the Call Stack callbacks from the Microtask Queue have priority. The Microtask Queue is an addition to ECMAScript 2015.

Asynchronous evolution: from Promises to async/await

JavaScript is moving fast and every year we get constant improvements to the language. Promises seemed the arrival point but with ECMAScript 2017 (ES8) a new syntax was born: async/await. async/await is just a stylistic improvement, what we call syntactic sugar. async/await does not alter JavaScript by any means (remember, JavaScript must be backward compatible with older browser and should not break existing code). It is just a new way for writing asynchronous code based on Promises. Let's make an example. Earlier we save a Promise with the corresponding then:


```
1  const myPromise = new Promise(function(resolve, reject) {
2    resolve([ { name: "Chris" } ]);
3  });
4
5  myPromise.then((data) => console.log(data))
```

Now with `async/await` we can handle asynchronous code in a way that looks synchronous from the reader's point of view. Instead of using `then` we can wrap the Promise inside a function marked `async` and then `await` on the result:

```
1  const myPromise = new Promise(function(resolve, reject) {
2    resolve([ { name: "Chris" } ]);
3  });
4
5  async function getData() {
6    const data = await myPromise;
7    console.log(data);
8  }
9
10 getData();
```

Makes sense right? Now, the funny thing is that an `async` function will always return a Promise and nobody prevents you from doing that:

```
1  async function getData() {
2    const data = await myPromise;
3    return data;
4  }
5
6  getData().then(data => console.log(data));
```

And how about errors? One of the boon offered by `async/await` is the chance to use `try/catch`. Let's look again at a Promise where for handling errors we use the `catch` handler:

```
1  const myPromise = new Promise(function(resolve, reject) {
2    reject('Errored, sorry!');
3  });
4
5  myPromise.catch(err => console.log(err));
```

With `async` functions we can refactor to the following code:

```
1  async function getData() {
2    try {
3      const data = await myPromise;
4      console.log(data);
5      // or return the data with return data
6    } catch (error) {
7      console.log(error);
8    }
9  }
10
11  getData();
```

Not everybody is still sold to this style though. try/catch can make your code noisy. And while using try/catch there is another quirk to point out. Consider the following code, raising an error inside a try block:

```
1  async function getData() {
2    try {
3      if (true) {
4        throw Error("Catch me if you can");
5      }
6    } catch (err) {
7      console.log(err.message);
8    }
9  }
10
11  getData()
12    .then(() => console.log("I will run no matter what!"))
13    .catch(() => console.log("Catching err"));
```

What of the two strings is printed to the console? Remember that try/catch is a synchronous construct but our asynchronous function produces a Promise. They travel on two different tracks, like two trains. But they will never meet! That is, an error raised by throw will never trigger the catch handler of getData(). Running the above code will result in “Catch me if you can” followed by “I will run no matter what!”. In the real world we don’t want throw to trigger a then handler. One possible solution is returning Promise.reject() from the function:

```
1  async function getData() {  
2    try {  
3      if (true) {  
4        return Promise.reject("Catch me if you can");  
5      }  
6    } catch (err) {  
7      console.log(err.message);  
8    }  
9  }
```

Now the error will be handled as expected:

```
1  getData()  
2    .then(() => console.log("I will NOT run no matter what!"))  
3    .catch(() => console.log("Catching err"));  
4  
5  "Catching err" // output
```

Besides that `async/await` seems the best way for structuring asynchronous code in JavaScript. We have better control over error handling and the code looks more cleaner. Anyway, I don't advise refactoring all your JavaScript code to `async/await`. These are choices that must be discussed with the team. But if you work alone whether you use plain Promises or `async/await` it's a matter of personal preference.

Conclusions

JavaScript is a scripting language for the web and has the peculiarity of being compiled first and then interpreted by an engine. Among the most popular JavaScript engines there are V8, used by Google Chrome and Node.js, SpiderMonkey, built for the web browser Firefox, and JavaScriptCore, used by Safari.

JavaScript engines have a lot of moving parts: Call Stack, Global Memory, Event Loop, Callback Queue. All these parts work together in perfect tuning for handling synchronous and asynchronous code in JavaScript. JavaScript engines are single-threaded, that means there is a single Call Stack for running functions. This restriction is at the base of JavaScript's asynchronous nature: all the operations that require time must be taken in charge by an external entity (the browser for example) or by a callback function.

Since ES6, JavaScript engines also implement a Microtask Queue. The Microtask Queue is a queue data structure much like the Callback Queue. Only difference is that the Microtask Queue takes all the callbacks triggered by ES6 Promises. These callbacks, also called executors, have precedence

over callbacks handled by the Callback Queue. The mechanic is exposed in greater detail by Jake Archibald in [Tasks, microtasks, queues and schedules](https://jakearchibald.com/2015/tasks-microtasks-queues-and-schedules/)¹².

For simplifying asynchronous code flow ECMAScript 2015 brought us Promises. A Promise is an asynchronous object and is used to represent either the failure or the success of any asynchronous operation. But the improvements did not stop there. In 2017 `async/await` was born: it's a stylistic make up for Promises that makes possible to write asynchronous code as if it was synchronous. And more important makes possible to use `try/catch` for handling errors in asynchronous code.



Sharpen up your JavaScript skills

- How does the browser is able to understand JavaScript code?
- What's the main task of the Call Stack?
- Can you describe the main components of a JavaScript engine and how they work together?
- What's the Microtask Queue made for?
- What is a Promise?
- It is possible to handle errors in asynchronous code? If yes, how? x> - Can you name a couple of methods of the browser API?

¹²<https://jakearchibald.com/2015/tasks-microtasks-queues-and-schedules/>

Chapter 4. Closures and modules in JavaScript

Global madness

Global variables are a plague in software engineering. We've been taught to avoid them at all costs although globals are useful in some situations. For example when working with JavaScript in the browser we have access to the global window object. window has a lot of useful methods like:

```
1 window.alert('Hello world'); // Shows an alert
2 window.setTimeout(callback, 3000); // Delay execution
3 window.fetch(someUrl); // make XHR requests
4 window.open(); // Opens a new tab
```

These methods are also available globally like so:

```
1 alert('Hello world'); // Shows an alert
2 setTimeout(callback, 3000); // Delay execution
3 fetch(someUrl); // make XHR requests
4 open(); // Opens a new tab
```

That's handy. Redux is another example of “good” globals: the entire application's state is stored in a single JavaScript object which is accessible from the entire app (through Redux). But when writing JavaScript code in a team where maybe 50 developers are supposed to ship features ... how do you cope with code like this:

```
1 var arr = [];
2
3 function addToArr(element) {
4   arr.push(element);
5   return element + " added!";
6 }
```

What are the odds of a colleague creating a new global array named arr in another file? Very high if you ask me! Another reason why globals in JavaScript are really bad is that the engine is kind enough to create global variables for us. What I mean is that if you forgot to put var before a variable's name, like so:

```
1 name = "Valentino";
```

the engine creates a global variable for you! Shudder ... Even worst when the “unintended” variable is created inside a function:

```
1 function doStuff() {  
2   name = "Valentino";  
3 }  
4  
5 doStuff();  
6  
7 console.log(name); // "Valentino"
```

An innocent function ended up polluting the global environment. Luckily there is a way for neutralizing this behavior with “strict mode”. The statement “use strict” at the very top of every JavaScript file is enough to avoid silly mistakes:

```
1 "use strict";  
2  
3 function doStuff() {  
4   name = "Valentino";  
5 }  
6  
7 doStuff();  
8  
9 console.log(name); // ReferenceError: name is not defined
```

Always use strict! Errors like that are a solved problem these days. Developers are becoming more and more self-disciplined (I hope so) but back in the old days global variables and procedural code was the norm in JavaScript. So we had to find a way for solving the “global” problem and luckily JavaScript has always had a built-in mechanism for that.

Demystifying closures

So how do we protect our global variables from prying eyes? Let’s start with a naive solution, moving `arr` inside a function:

```
1 function addToArr(element) {  
2   var arr = [];  
3   arr.push(element);  
4   return element + " added to " + arr;  
5 }
```

Seems reasonable but the result isn't what I was expecting:

```
1 var firstPass = addToArr("a");  
2 var secondPass = addToArr("b");  
3 console.log(firstPass); // a added to a  
4 console.log(secondPass); // a added to a
```

Ouch. `arr` keeps resetting at every function invocation. It is a local variable now, while in the first example it was declared as a global. Turns out global variables are “live” and they’re not destroyed by the JavaScript engine. Local variables on the other hand are wiped out by the engine as part of a process called garbage collection (I will spare you the nitty gritty but here’s an interesting article on the subject: [A tour of V8: Garbage Collection¹³](http://jayconrod.com/posts/55/a-tour-of-v8-garbage-collection)). Seems there is no way to keep local variables from being destroyed? Or there is? Would a closure help? But what’s a closure anyway?

It should be no surprise to you that JavaScript functions can contain other functions. What follows is perfectly valid code:

```
1 function addToArr(element) {  
2   var arr = [];  
3  
4   function push() {  
5     arr.push(element);  
6   }  
7  
8   return element + " added to " + arr;  
9 }
```

The code does nothing but ... ten years ago I was trying to solve a similar problem. I wanted to distribute a simple, self-contained functionality to my colleagues. No way I could handle them a JavaScript file with a bunch of global variables. I had to find a way for protecting those variables while keeping some local state inside the function. I tried literally everything until out of desperation I did some research and come up with something like this:

¹³<http://jayconrod.com/posts/55/a-tour-of-v8-garbage-collection>

```
1  function addToArr(element) {  
2    var arr = [];  
3  
4    return function push() {  
5      arr.push(element);  
6      console.log(arr);  
7    };  
8  
9    //return element + " added to " + arr;  
10 }
```

The outer function becomes a mere container, returning another function. The second return statement is commented because that code will never be reached. At this point we know that the result of a function invocation can be saved inside a variable like so:

```
1  var result = addToArr();
```

Now `result` becomes an executable JavaScript function:

```
1  var result = addToArr();  
2  result("a");  
3  result("b");
```

There's just a fix to make, moving the parameter "element" from the outer to the inner function:

```
1  function addToArr() {  
2    var arr = [];  
3  
4    return function push(element) {  
5      arr.push(element);  
6      console.log(arr);  
7    };  
8  
9    //return element + " added to " + arr;  
10 }
```

And that's where the magic happens. Here's the complete code:


```

1  function addToArr() {
2      var arr = [];
3
4      return function push(element) {
5          arr.push(element);
6          console.log(arr);
7      };
8
9      //return element + " added to " + arr;
10 }
11
12 var result = addToArr();
13 result("a"); // [ 'a' ]
14 result("b"); // [ 'a', 'b' ]

```

Bingo! What's this sorcery? It's called JavaScript closure: a function which is able to remember its environment. For doing so the inner function must be the return value of an enclosing (outer) function. This programming technique is also called factory function in nerd circles. The code can be tweaked a bit, maybe a better naming for the variables, and the inner function can be anonymous if you want:

```

1  function addToArr() {
2      var arr = [];
3
4      return function(element) {
5          arr.push(element);
6          return element + " added to " + arr;
7      };
8  }
9
10 var closure = addToArr();
11 console.log(closure("a")); // a added to a
12 console.log(closure("b")); // a added to a,b

```

Should be clear now that “closure” is the inner function. But one question needs to be addressed: why are we doing this? What's the real purpose of a JavaScript closure?



The more you know

Closures exist not only in JavaScript: Python for example has also closures in the form of a function returning another function.

The need for closures

Besides mere “academic” knowledge, JavaScript closures are useful for a number of reasons. They help in:

- giving privacy to otherwise global variables
- preserving variables (state) between function calls

One of the most interesting application for closures in JavaScript is the [module pattern](#)¹⁴. Before ECMAScript 2015 which introduced ES Modules there was no way for modularizing JavaScript code and give “privacy” to variables and methods other than wrapping them inside functions. Closures, paired with immediately invoked function expressions were and still are the most natural solution for structuring JavaScript code. A JavaScript module in it’s simplest form is a variable capturing the result of an IIFE (an immediately invoked function expression):

```
1 var Person = (function() {  
2     // do stuff  
3 })();
```

Inside the module you can have “private” variables and methods:

```
1 var Person = (function() {  
2     var person = {  
3         name: "",  
4         age: 0  
5     };  
6  
7     function setName(personName) {  
8         person.name = personName;  
9     }  
10  
11    function setAge(personAge) {  
12        person.age = personAge;  
13    }  
14 })();
```

From the outside we can’t access `person.name` or `person.age` nor we can call `setName` or `setAge`. Everything inside the module is “private”. If we want to expose our methods to the public we can return an object containing references to private methods:

¹⁴<https://addyosmani.com/resources/essentialjsdesignpatterns/book/#revealingmodulepatternjavascript>

```
1  var Person = (function() {
2    var person = {
3      name: "",
4      age: 0
5    };
6
7    function setName(personName) {
8      person.name = personName;
9    }
10
11   function setAge(personAge) {
12     person.age = personAge;
13   }
14
15   return {
16     setName: setName,
17     setAge: setAge
18   };
19 })();
```

The person object is still hidden away but it's easy to add a method for returning the new person:

```
1  var Person = (function() {
2    var person = {
3      name: "",
4      age: 0
5    };
6
7    function setName(personName) {
8      person.name = personName;
9    }
10
11   function setAge(personAge) {
12     person.age = personAge;
13   }
14
15   function getPerson() {
16     return person.name + " " + person.age;
17   }
18
19   return {
20     setName: setName,
21     setAge: setAge,
```

```

22     getPerson: getPerson
23   };
24   })();
25
26   Person.setName("Tom");
27   Person.setAge(44);
28   var person = Person.getPerson();
29   console.log(person); // Tom 44

```

This way person is still not directly accessible from the outside:

```

1 console.log(Person.person); // undefined

```

The module pattern is not the only way for structuring JavaScript code. With objects we could achieve the same outcome:

```

1 var Person = {
2   name: "",
3   age: 0,
4   setName: function(personName) {
5     this.name = personName;
6   }
7   // other methods here
8 };

```

But now internal properties are leaking as you would expect from a JavaScript object:

```

1 var Person = {
2   name: "",
3   age: 0,
4   setName: function(personName) {
5     this.name = personName;
6   }
7   // other methods here
8 };
9
10 Person.setName("Tom");
11
12 console.log(Person.name); // Tom

```

That's one of the main selling point for modules. Another boon is that modules help in organizing JavaScript code in a way that it's both idiomatic and readable. In other words almost every JavaScript developer can look at the following code and guess its purpose:

```
1  "use strict";
2
3  var Person = (function() {
4      var person = {
5          name: "",
6          age: 0
7      };
8
9      function setName(personName) {
10         person.name = personName;
11     }
12
13     function setAge(personAge) {
14         person.age = personAge;
15     }
16
17     function getPerson() {
18         return person.name + " " + person.age;
19     }
20
21     return {
22         setName: setName,
23         setAge: setAge,
24         getPerson: getPerson
25     };
26 })();
```



ES2015 TIP

If you're writing ES2015 the returning object can be simplified with object shorthand property names. You can return { setName, setAge, getPerson } instead of { setName: setName, setAge: setAge, getPerson: getPerson }.

Conclusions

Global variables are bad and you should avoid them whenever you can. Sometimes are useful but in JavaScript we must take extra care because the engine takes the liberty to create global variables out of thin air. A number of patterns emerged during the years for taming global variables, the module pattern being one of them. The module pattern builds on closures, an innate feature of JavaScript. A closure in JavaScript is a function able to “remember” its variable’s environment, even between

subsequent function calls. A closure is created when we return a function from another function, a pattern also known as “factory function”.



Sharpen up your JavaScript skills

- What’s the meaning of “closure”?
- Why global variables are bad?
- What’s a JavaScript module and why you would use it?

Chapter 5. The secret life of JavaScript objects

Everything is an object!

How many times have you heard “everything is an object in JavaScript”? Have you ever asked yourself what’s the true meaning of that? “Everything is an object” is common to other languages too, like Python. But objects in Python are not just containers for values like a JavaScript object. Object in Python is a class. There is something similar in JavaScript but an “object” in JS is just a container for keys and values:

```
1 var obj = { name: "Tom", age: 34 }
```

Really, an object in JavaScript is a “dumb” type yet a lot of other entities seem to derive from objects. Even arrays. Create an array in JavaScript like so:

```
1 var arr = [1,2,3,4,5]
```

and then check it out with the `typeof` operator. You’ll see a surprising result:

```
1 typeof arr
2 "object"
```

It appears that arrays are a special kind of object! Even functions in JavaScript are objects. And there’s more if you dig deeper. Create a function and you will have methods ready for use, attached to that function:

```
1 var a = function(){ return false; }
2 a.toString()
```

Output:

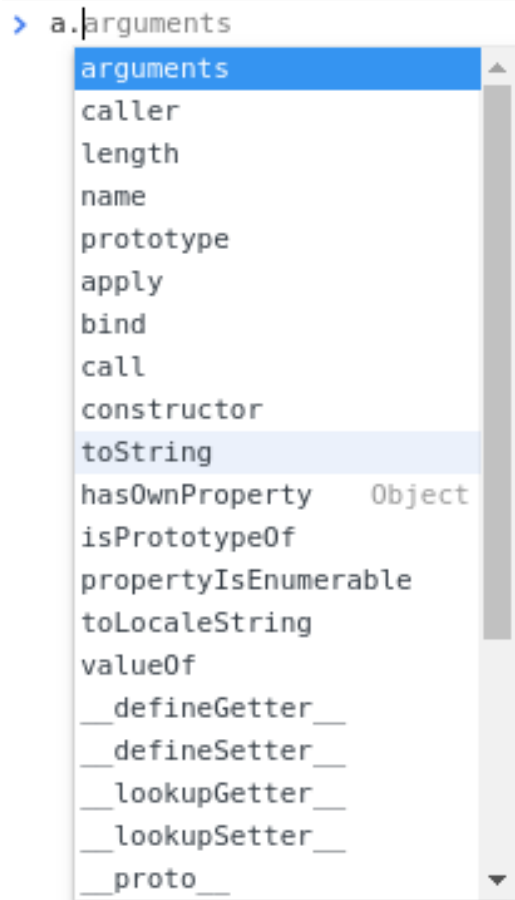
```
1 "function(){ return false; }"
```

There must be something more under the hood because I didn’t attach `toString` to my function. Did you? Where does it come from? Turns out `Object` has a method named `.toString`. Seems like our function has the same methods of `Object`!

```
1 Object.toString()
```

The mystery thickens even more if we look at all the methods and properties “attached” to a simple JavaScript function:

```
> var a = function(){ return false; }  
< undefined
```



Our function has a lot of methods!

Who put those methods there? I said that functions in JavaScript are a special kind of object. Could it be an hint? Maybe! But for now let’s hold our joy. Take a look again at the picture above: there is a strangely named property on our function called `prototype`. What’s that?

`prototype` in JavaScript is an object. It is like a backpack, attached to most of the JavaScript built-in objects. For example `Object`, `Function`, `Array`, `Date`, `Error`, all have a “`prototype`”:


```
1 typeof Object.prototype; // 'object'
2 typeof Date.prototype; // 'object'
3 typeof String.prototype; // 'object'
4 typeof Number.prototype; // 'object'
5 typeof Array.prototype; // 'object'
6 typeof Error.prototype; // 'object'
```



Don't touch the “native” prototype

As a rule of thumb do not never ever touch the prototype object of built-in JavaScript objects. In other words don't modify or add new methods on `Object.prototype`, `String.prototype`, `Array.prototype` or whatever built-in object you see. You're asking for trouble. JavaScript is changing constantly, new feature are being added into the language year after year. And there is an high chance that some `String.prototype.shinyMethod` will being overwritten by a native method with the same name in the future.

Pay attention that built-in objects have a capital letter and resemble those lego bricks we saw earlier:

- String
- Number
- Boolean
- Object
- Symbol
- Null
- Undefined

These lego bricks are JavaScript's types, and most of them are also primitives, except `Object` which is a type. Built-in objects on the other hand are like mirrors of JavaScript's types and are used as a function too. For example you can use `String` as a function for converting a number to a string:

```
1 String(34)
```

So what's the purpose of prototype? Prototype is an home for all the common method and properties that should be available across “child” objects deriving from an ancestor. That is, given an original prototype, JavaScript developers can create new objects that will use a single prototype as the source of truth for common functions. Let's see how (spoiler: `Object.create`).



TIP

The method `.toString` on our function is not the same exact version of `Object.toString`. Function replaces that method with a custom version which prints a stringified version of itself.

Suppose you're absolutely new to the language but your boss gives you a task with JavaScript. You should build a chat application and she said that Person is one of the “models” of this app. A person can connect to a channel, can send messages and maybe is welcomed with a nice greeting when she logs in. Being new to JavaScript you don't even know where to start. But you're smart and from the documentation you've been reading lately it appears that JavaScript objects are really powerful. In particular [this page on MDN¹⁵](#) lists all the methods available on JavaScript objects and `create()` looks interesting. `Object.create()` “creates a new object with the specified prototype object and properties”.

Maybe there is a way to create a “primitive” Person from which other objects can copy methods and properties? We saw from chapter 2 that objects in JavaScript may contain not only simple values but also functions:

```
1 var firstObj = {  
2   name: "Alex",  
3   age: 33,  
4   hello: function() {  
5     console.log("Hello " + this.name);  
6   }  
7 };
```

Now that's a game changer! Let's create a Person object then. While we're there let's use ES6 template literals for concatenating “Hello” with the name:

```
1 var Person = {  
2   name: "noname",  
3   age: 0,  
4   greet: function() {  
5     console.log(`Hello ${this.name}`);  
6   }  
7 };
```

You may wonder why I'm using a capital letter for Person. We'll see later in detail but for now think of it as of a convention developers use for signalling “models”. Person is a model in your application. Given this Person blueprint we need to create more people starting from that single Person. And `Object.create()` is your friend here.

Creating and linking objects

You have a feeling that objects in JavaScript are somehow linked together. And `Object.create()` confirms the theory. This method creates new objects starting from an original object. Let's create a new person then:

¹⁵https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object

```
1  var Person = {
2    name: "noname",
3    age: 0,
4    greet: function() {
5      console.log(`Hello ${this.name}`);
6    }
7  };
8
9  var Tom = Object.create(Person);
```

Now, Tom is a new person but I didn't specify any new method or property on it. Yet it still has access to the original name and age:

```
1  var Person = {
2    name: "noname",
3    age: 0,
4    greet: function() {
5      console.log(`Hello ${this.name}`);
6    }
7  };
8
9  var Tom = Object.create(Person);
10
11  var tomAge = Tom.age;
12  var tomName = Tom.name;
13
14  console.log(`${tomAge} ${tomName}`);
15
16  // Output: 0 noname
```

Bingo! Now you can create new people starting from a single, common ancestor. But the weird thing is that new objects remains connected to the original object. Not a big problem since “children” objects can customize properties and methods:

```
1  var Person = {
2    name: "noname",
3    age: 0,
4    greet: function() {
5      console.log(`Hello ${this.name}`);
6    }
7  };
8
9  var Tom = Object.create(Person);
10
11  Tom.age = 34;
12  Tom.name = "Tom";
13  var tomAge = Tom.age;
14  var tomName = Tom.name;
15
16  console.log(`${tomAge} ${tomName}`);
17
18  // Output: 34 Tom
```

And that’s called “shadowing” the original properties. There is also another way for passing properties to our new object. `Object.create` takes another object as the second argument where you can specify keys and values for the new object:

```
1  var Tom = Object.create(Person, {
2    age: {
3      value: 34
4    },
5    name: {
6      value: "Tom"
7    }
8  });
```

Properties configured this way are by default not writable, not enumerable, not configurable. Not writable means you cannot change that property later, the alteration is simply ignored:

```
1  var Tom = Object.create(Person, {
2    age: {
3      value: 34
4    },
5    name: {
6      value: "Tom"
7    }
8  });
9
10 Tom.age = 80;
11 Tom.name = "evilchange";
12
13 var tomAge = Tom.age;
14 var tomName = Tom.name;
15
16 Tom.greet();
17
18 console.log(`${tomAge} ${tomName}`);
19
20 // Hello Tom
21 // 34 Tom
```

Not enumerable means that properties will not show in a for...in loop for example:

```
1  for (const key in Tom) {
2    console.log(key);
3  }
4
5  // Output: greet
```

But as you can see, linked properties shows up because the JavaScript engine goes up the link chain and finds greet up on the “parent” object. Finally not configurable means the property neither can be modified, nor deleted:

```
1 Tom.age = 80;
2 Tom.name = "evilchange";
3 delete Tom.name;
4 var tomAge = Tom.age;
5 var tomName = Tom.name;
6
7 console.log(`${tomAge} ${tomName}`);
8
9 // 34 Tom
```

If you want to change the behaviour of properties just specify whether you want them writable, configurable and maybe enumerable:

```
1 var Tom = Object.create(Person, {
2   age: {
3     value: 34,
4     enumerable: true,
5     writable: true,
6     configurable: true
7   },
8   name: {
9     value: "Tom",
10    enumerable: true,
11    writable: true,
12    configurable: true
13  }
14 });
```

Now, back to our person, Tom has also access to greet():

```
1 var Person = {
2   name: "noname",
3   age: 0,
4   greet: function() {
5     console.log(`Hello ${this.name}`);
6   }
7 };
8
9 var Tom = Object.create(Person);
10
11 Tom.age = 34;
12 Tom.name = "Tom";
```

```
13 var tomAge = Tom.age;
14 var tomName = Tom.name;
15 Tom.greet();
16
17 console.log(`${tomAge} ${tomName}`);
18
19 // Hello Tom
20 // 34 Tom
```

Don't worry too much about "this" for now. In the next chapter you'll learn more about it. For now keep in mind that "this" is a reference to some object in which the function is executing. In our case `greet()` runs in the context of `Tom` and for that reason has access to `this.name`.

And now that you know a bit more about JavaScript objects let's explore another pattern for creating entities: the constructor function.

Constructing JavaScript objects

Up until now I gave you just an hint of the "prototype" but other than playing with `Object.create()` we didn't use it directly. A lot of developers coming from more traditional languages like Java stumbled upon `Object.create()` and "prototypal" over the years. They had to find a way for making sense of this weird JavaScript language. So with the time a new pattern emerged: the constructor function. Using a function for creating new objects sounded reasonable. Suppose you want to transform your `Person` object to a function, you could come up with the following code:

```
1 function Person(name, age) {
2   var newPerson = {};
3   newPerson.age = age;
4   newPerson.name = name;
5   newPerson.greet = function() {
6     console.log("Hello " + newPerson.name);
7   };
8   return newPerson;
9 }
```

So instead of calling `Object.create()` all over the place you can simply call `Person` as a function and get back an object:

```
1 var me = Person("Valentino");
```

The constructor pattern helps encapsulating the creation and the configuration of a series of JavaScript objects. And here again we have `Person` with the capital letter. It's a convention borrowed

from object oriented languages where classes have a capital letter. In JavaScript there are no classes but this style eventually made developers more comfortable. Now the example above has a serious problem: every time we create a new object we duplicate `greet()` over and over. That's sub-optimal. Luckily you learned that `Object.create()` creates links between objects and that leads to a nicer solution. First things first let's move the `greet()` method outside, to an object on its own. You can then link new objects to that common object with `Object.create()`:

```
1  var personMethods = {
2    greet: function() {
3      console.log("Hello " + this.name);
4    }
5  };
6
7  function Person(name, age) {
8    // greet lives outside now
9    var newPerson = Object.create(personMethods);
10   newPerson.age = age;
11   newPerson.name = name;
12   return newPerson;
13 }
14
15 var me = Person("Valentino");
16 me.greet();
17
18 // Output: "Valentino"
```

How nice! But still we can do better. How about the good 'ol "prototype" you saw earlier? Could help here? Turns out, functions (and all the built-in objects) in JavaScript have a backpack called `prototype`. `prototype` is just another object and can contain everything we can think of: properties, methods. For adding a method to our `Person` prototype you can do:

```
1  Person.prototype.greet = function() {
2    console.log("Hello " + this.name);
3  };
```

and get rid of `personMethods`. The only problem now is that new objects won't be linked automatically to a common ancestor unless we adjust the argument for `Object.create`:


```
1  function Person(name, age) {
2    // greet lives outside now
3    var newPerson = Object.create(Person.prototype);
4    newPerson.age = age;
5    newPerson.name = name;
6    return newPerson;
7  }
8
9  Person.prototype.greet = function() {
10   console.log("Hello " + this.name);
11 };
12
13 var me = Person("Valentino");
14 me.greet();
15
16 // Output: "Hello Valentino"
```

Now the source for all common methods is `Person.prototype`. At this point we're ready to do the last tweak to our `Person` because there is a more cleaner way for arranging that function. With the [new operator in JavaScript¹⁶](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new) we can get rid of all the noise inside `Person` and just take care of assigning the arguments to `this`. That means, the following code:

```
1  function Person(name, age) {
2    // greet lives outside now
3    var newPerson = Object.create(Person.prototype);
4    newPerson.age = age;
5    newPerson.name = name;
6    return newPerson;
7  }
```

becomes:

```
1  function Person(name, age) {
2    this.name = name;
3    this.age = age;
4  }
```

And here's the complete code:

¹⁶<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new>

```
1  function Person(name, age) {
2      this.name = name;
3      this.age = age;
4  }
5
6  Person.prototype.greet = function() {
7      console.log("Hello " + this.name);
8  };
9
10 var me = new Person("Valentino");
11 me.greet();
12
13 // Output: "Hello Valentino"
```

Note that now you need to prefix the function call with `new`. That’s called “constructor call” and it’s the “standard” syntax for creating new shape of objects in JavaScript. In other words, `new` does all the hard work for us:

- creates a new blank objects
- links the object to a prototype
- points `this` to the newly created object
- returns the new object

Neat!

Checking the “linkage”

There are many ways for checking the link between our JavaScript objects. `Object.getPrototypeOf` for example is a method which returns the prototype of any given object. Consider the following code where Tom is created from a “Person” blueprint:

```
1  var Person = {
2      name: "noname",
3      age: 0,
4      greet: function() {
5          console.log(`Hello ${this.name}`);
6      }
7  };
8
9  var Tom = Object.create(Person);
```

You can check if Person appears to be the prototype of Tom:

```
1 var tomPrototype = Object.getPrototypeOf(Tom);
2
3 console.log(tomPrototype === Person);
4
5 // Output: true
```

`Object.getPrototypeOf` works of course also if you constructed the object with a constructor call. But you should check against the prototype object, not on the constructor function itself:

```
1 function Person(name, age) {
2   this.name = name;
3   this.age = age;
4 }
5
6 Person.prototype.greet = function() {
7   console.log("Hello " + this.name);
8 };
9
10 var me = new Person("Valentino");
11
12 var mePrototype = Object.getPrototypeOf(me);
13
14 console.log(mePrototype === Person.prototype);
15
16 // Output: true
```

Besides `Object.getPrototypeOf` there is another method, `isPrototypeOf`. It checks whether the given prototype appears in the prototype chain of the object we're checking against. In the example above "me" is an object linked to `Person.prototype`. So the following check returns "Yes I am!":

```
1 Person.prototype.isPrototypeOf(me) && console.log("Yes I am!");
```

There is also a third way for checking the link between JavaScript objects: it's the `instanceof` operator. In all honesty the name is a bit misleading because there are no "instances" in JavaScript. In a true object oriented language an instance is a new object created from a class. Consider the following example in Python. We have a class named `Person` and from that class we create a new instance called "tom":

```

1  class Person():
2      def __init__(self, age, name):
3          self.age = age;
4          self.name = name;
5
6      def __str__(self):
7          return f'{self.name}'
8
9
10 tom = Person(34, 'Tom')

```

Notice that in Python there is no `new` keyword. Now we can check whether `tom` is an instance of `Person` with the `isinstance` method:

```

1  isinstance(tom, Person)
2
3  // Output: True

```

`Tom` is also an instance of “object” in Python and the following code returns true as well:

```

1  isinstance(tom, object)
2
3  // Output: True

```

According to the documentation `isinstance` “Return true if the object argument is an instance of the class argument, or of a (direct, indirect or virtual) subclass thereof”. So we’re talking about classes here. Now let’s see what `instanceof` does instead. We’re going to create `tom` starting from a `Person` function in JavaScript (because there are no real classes):

```

1  function Person(name, age) {
2      this.name = name;
3      this.age = age;
4  }
5
6  Person.prototype.greet = function() {
7      console.log(`Hello ${this.name}`);
8  };
9
10 var tom = new Person(34, "Tom");

```

And now let’s use `instanceof` in a creative way:

```
1  if (tom instanceof Object) {  
2    console.log("Yes I am!");  
3  }  
4  
5  if (tom instanceof Person) {  
6    console.log("Yes I am!");  
7  }
```

Take a moment and try to guess the output. `instanceof` simply checks whether `Person.prototype` or `Object.prototype` appear in the prototype chain of `tom`. And indeed they are! The output is “Yes I am!” “Yes I am!”.

So here’s the key takeaway: the prototype of a JavaScript object is (almost) always connected to both the direct “parent” and to `Object.prototype`. There are no classes like Python or Java. JavaScript is just made out of objects. But since I mentioned it, what’s a prototype chain by the way? If you paid attention I mentioned “prototype chain” a couple of times. It may look like magic that a JavaScript object has access to a method defined elsewhere in the code. Consider again the following example:

```
1  var Person = {  
2    name: "noname",  
3    age: 0,  
4    greet: function() {  
5      console.log(`Hello ${this.name}`);  
6    }  
7  };  
8  
9  var Tom = Object.create(Person);  
10  
11 Tom.greet();
```

Tom has access to `greet()` even though the method does not exist directly on the “Tom” object. How is that possible? Unlike OOP languages like Python or Java where every child object gets a copy of all the parent’s methods, in JavaScript the relation works kind of backward. Children remain connected to the parent and have access to properties and methods from it. That’s an intrinsic trait of JavaScript which borrowed the prototype system from another language called Self. When I access `greet()` the JavaScript engine checks if that method is available directly on Tom. If not the search continues up the chain until the method is found. The “chain” is the hierarchy of prototypes objects to which Tom is connected. In our case Tom is an object of type Person so Tom’s prototype is connected to `Person.prototype`. And `Person.prototype` is an object of type `Object` so shares the same prototype of `Object.prototype`. If `greet()` would not have been available on `Person.prototype` then the search would have continued up the chain until reaching `Object.prototype`. That’s what we call a “prototype chain”.

You might have read the term “prototypal inheritance” in dozens of tutorials. I see a lot of JavaScript developers confused by “prototypal inheritance”. It’s simpler than you think: there is no “prototypal inheritance” in JavaScript. Just forget that sentence and remember that JavaScript objects are most of the times linked to other objects. It’s a link. There is no inheritance.

Protecting objects from manipulation

Most of the times it’s fine to keep JavaScript objects “extensible”, that is, making possible for other developers to add new properties to an object. But there are situations when you want to protect an object from further manipulation. Consider a simple object:

```
1 var superImportantObject = {  
2   property1: "some string",  
3   property2: "some other string"  
4 };
```

By default everybody can add new properties to that object:

```
1 var superImportantObject = {  
2   property1: "some string",  
3   property2: "some other string"  
4 };  
5  
6 superImportantObject.anotherProperty = "Hei!";  
7  
8 console.log(superImportantObject.anotherProperty); // Hei!
```

Now suppose you want to protect the object from new additions: in jargon you want to prevent “extensions” to the object. You can call `Object.preventExtensions` on it:

```
1 var superImportantObject = {  
2   property1: "some string",  
3   property2: "some other string"  
4 };  
5  
6 Object.preventExtensions(superImportantObject);  
7  
8 superImportantObject.anotherProperty = "Hei!";  
9  
10 console.log(superImportantObject.anotherProperty); // undefined
```

This technique is handy for “protecting” critical objects in your code. There are also a lot of pre-made objects in JavaScript that are closed for extension, thus preventing developers from adding new properties on them. That’s the case of “important” objects like the response of an XMLHttpRequest. Browser vendors forbid the addition of new properties on the response object:

```
1 var request = new XMLHttpRequest();
2 request.open("GET", "https://jsonplaceholder.typicode.com/posts");
3 request.send();
4 request.onload = function() {
5     this.response.arbitraryProp = "messing with youuu";
6     console.log(this.response.arbitraryProp); // undefined
7 };
```

That’s done by calling `Object.preventExtensions` internally on the “response” object. You can also check whether a JavaScript object is protected with the `Object.isExtensible` method. It will return `true` if the object is extensible:

```
1 var superImportantObject = {
2     property1: "some string",
3     property2: "some other string"
4 };
5
6 Object.isExtensible(superImportantObject) && console.log("Open for extension!");
```

or false if it is not:

```
1 var superImportantObject = {
2     property1: "some string",
3     property2: "some other string"
4 };
5
6 Object.preventExtensions(superImportantObject);
7
8 Object.isExtensible(superImportantObject) ||
9     console.log("Closed for extension!");
```

Of course existing properties of the object can be changed or even deleted:

```
1  var superImportantObject = {
2    property1: "some string",
3    property2: "some other string"
4  };
5
6  Object.preventExtensions(superImportantObject);
7
8  delete superImportantObject.property1;
9
10 superImportantObject.property2 = "yees";
11
12 console.log(superImportantObject); // { property2: 'yees' }
```

Now, to prevent this sort of manipulation you could always define each property as not writable and not configurable. And for that there is a method called `Object.defineProperties`:

```
1  var superImportantObject = {};
2
3  Object.defineProperties(superImportantObject, {
4    property1: {
5      configurable: false,
6      writable: false,
7      enumerable: true,
8      value: "some string"
9    },
10   property2: {
11     configurable: false,
12     writable: false,
13     enumerable: true,
14     value: "some other string"
15   }
16 });
```

Or, more conveniently you can use `Object.freeze` on the original object:

```
1  var superImportantObject = {
2    property1: "some string",
3    property2: "some other string"
4  };
5
6  Object.freeze(superImportantObject);
```


`Object.freeze` does the same job of `Object.preventExtensions`, plus it makes all the object's properties not writable and not configurable. The only drawback is that “freezing” works only on the first level of the object: nested object won't be affected by the operation.

And now having shed some light on JavaScript objects, let's talk a bit about ES6 classes. What are they, really?

Class illusion

There is a huge literature on ES6 classes so here I'll give you just a taste of what they are. Is JavaScript a real object oriented language? It seems so if you look at this code:

```
1  class Person {
2    constructor(name) {
3      this.name = name;
4    }
5
6    greet() {
7      console.log(`Hello ${this.name}`);
8    }
9  }
```

The syntax is quite similar to classes in other programming languages like Python:

```
1  class Person:
2      def __init__(self, name):
3          self.name = name
4
5      def greet(self):
6          return 'Hello' + self.name
```

or PHP (forgive me father):

```
1 class Person {  
2     public $name;  
3  
4     public function __construct($name){  
5         $this->name = $name;  
6     }  
7  
8     public function greet(){  
9         echo 'Hello ' . $this->name;  
10    }  
11 }
```

Classes have been introduced with ES6 in 2015. But at this point it should be clear to you that there are no “real” classes in JavaScript. Everything is just an object and despite the keyword `class` the “prototype system” is still there. Remember, new JavaScript versions are backward-compatible. That means new features are added on top of existing ones and most of those new features are syntactic sugar over legacy code. So what’s a class? There is a tool called Babel which helps developers to write modern JavaScript code and then run that code even in older browsers. In other words, new syntax like ES6 `class` is transformed to a legacy version. In fact, if you copy the `Person` class in the [Try it out section of the Babel website](https://babeljs.io/repl)¹⁷ you’ll get a `Person` function and a bunch of other methods. What a surprise! In the end, everything narrows down to prototypes, objects, and functions!

Conclusions

Almost everything in JavaScript is an object. Literally. JavaScript objects are containers for keys and values and may also contain functions. Object is the basic building block in JavaScript: so that it’s possible to create other custom objects starting from a common ancestor. Then we can link objects together by the means of an intrinsic trait of the language: the prototype system. Starting from a common object, we can create other objects sharing the same properties and methods of the original “parent”. But the way this works is not by copying methods and properties to every child, like OOP languages do. In JavaScript, every derived object remains connected to the parent. New custom objects are created either with `Object.create` or with a so-called constructor function. Paired with the new keyword, constructor functions sort of mimic traditional OOP classes.



Sharpen up your JavaScript skills

- How would you create an immutable JavaScript object?
- What is a constructor call?
- What is a constructor function?
- What’s the meaning of “prototype”?
- Can you describe step by step what `new` does under the hood?

¹⁷<https://babeljs.io/repl>

Chapter 6. This in JavaScript

Demystifying “this”

The `this` keyword in JavaScript is a thick mystery for beginners and a constant crux for more experience developers. `this` in fact is a moving target, something that could change during code execution, without any apparent reason. But let's see for a moment how the `this` keyword looks like in other programming languages. First, here's a JavaScript class:

```
1 class Person {
2   constructor(name) {
3     this.name = name;
4   }
5
6   greet() {
7     console.log("Hello " + this.name);
8   }
9 }
```

We know from chapter 5 that there are no classes in JavaScript yet `this` appears like a reference to the class itself. Also, there seems to be a parallel in Python classes called `self`:

```
1 class Person:
2     def __init__(self, name):
3         self.name = name
4
5     def greet(self):
6         return 'Hello' + self.name
```

In a Python class `self` represent the class's instance: i.e. the new object that gets created starting from the class:

```
1 me = Person('Valentino')
```

There is something similar in PHP too:

```

1  class Person {
2      public $name;
3
4      public function __construct($name){
5          $this->name = $name;
6      }
7
8      public function greet(){
9          echo 'Hello ' . $this->name;
10     }
11 }

```

Here `$this` is the actual class instance. If we take again our JavaScript class for creating two new objects from it we can see that whenever I call `object.name` I get the correct string back:

```

1  class Person {
2      constructor(name) {
3          this.name = name;
4      }
5
6      greet() {
7          console.log("Hello " + this.name);
8      }
9  }
10
11 const me = new Person("Valentino");
12 console.log(me.name); // 'Valentino'
13
14 const you = new Person("Tom");
15 console.log(you.name); // 'Tom'

```

Seems to make sense. JavaScript is like Python, Java, PHP and `this` appears to point to the actual class instance? Not really. Let's not forget that JavaScript is not an object oriented language, plus it's really permissive and dynamic, and there are no real classes. `this` has nothing to do with classes and I can prove the point with a simple JavaScript function (try it a browser):

```

1  function whoIsThis() {
2      console.log(this);
3  }
4
5  whoIsThis();

```

What's the output? Spoiler: when a JavaScript function runs in the so called global context `this` will be a reference to said global. And when running in a browser the global points to `window`. As you will see JavaScript functions have always a link to some context object: it is an object in which the function is running. The link is not fixed: it can change by accident or could be altered by developers on purpose.

Rule number 1: default binding

If you run the following code in a browser:

```
1 function whoIsThis() {  
2   console.log(this);  
3 }  
4  
5 whoIsThis();
```

you'll see the output:

```
1 Window {postMessage: f, blur: f, focus: f, close: f, parent: Window, ...}
```

Surprising? We're not inside any class and `this` is still there. That's because `this` is always present in regular functions and could assume many forms depending on how the function is called. When a function is called in the global environment it will always point its `this` to the global object, `window` in our case. That's the rule number 1 of `this` in JavaScript and it's called **default binding**. Default binding is like a fallback and most of the times it's undesirable. Any function running in the global environment can "pollute" global variables and wreck havoc to your code. Consider the following example:

```
1 function firstDev() {  
2   window.globalSum = function(a, b) {  
3     return a + b;  
4   };  
5 }  
6  
7 function nastyDev() {  
8   window.globalSum = null;  
9 }  
10  
11 firstDev();  
12 nastyDev();  
13 var result = firstDev();
```

```
14 console.log(result);
15
16 // Output: undefined
```

The first developer creates a global variable named `globalSum` and assigns a function to it. Later on another developer assigns `null` to the same variable causing malfunction in the code. Messing with global variables is always risky and for this reason JavaScript gained a “safe mode” years ago: strict mode. Strict mode is enabled by putting `"use strict"`; at the top of every JavaScript file. Strict mode has many positive effects and one of them is the neutralization of the default binding. In strict mode `this` is undefined when trying to access it from the global context:

```
1 "use strict";
2
3 function whoIsThis() {
4   console.log(this);
5 }
6
7 whoIsThis();
8
9 // Output: undefined
```

Strict mode makes your JavaScript code more secure and (almost) free from silly bugs like that. So to recap, default binding is the first rule of `this` in JavaScript: when the engine can’t figure out what `this` is it falls back to the global object. But that’s only half of the story because there are 3 other rules for `this`. Let’s see them together.

Rule number 2: implicit binding

“Implicit binding” is an intimidating term, but the theory behind it is not so complicated. It all narrows down to objects. It’s no surprise that JavaScript objects can contain functions:

```
1 var widget = {
2   items: ["a", "b", "c"],
3   printItems: function() {
4     console.log(this.items);
5   }
6 };
```

When a function is assigned as an object’s property then that object becomes the “home” (context object) in which the function runs. In other words the `this` keyword inside the function will point automatically to that object. That’s rule number 2 of `this` in JavaScript and goes under the name of **implicit binding**. Implicit binding is in action even when we call a function in the global context:

```
1  function whoIsThis() {  
2    console.log(this);  
3  }  
4  
5  whoIsThis();
```

You can't tell from the code but the JavaScript engine assigns the function to a new property on the global object `window`. Under the hood it's like:

```
1  window.whoIsThis = function() {  
2    console.log(this);  
3  };
```

You can easily confirm the assumption. Run the following code in a browser:

```
1  function whoIsThis() {  
2    console.log(this);  
3  }  
4  
5  console.log(typeof window.whoIsThis)
```

and you'll get back "function". And this point you might be asking: what's the real rule for `this` inside a global function? It's called default binding but in reality is more like of an implicit binding. Confusing right? It's JavaScript after all! Just remember that the JavaScript engine always falls back to the global `this` when in doubt about the context (default binding). On the other hand when a function is defined inside a JavaScript object and is called as part of that object then there is no other way around: `this` refers to the host object (implicit binding). And now let's see what is the third rule for `this` in JavaScript.

Rule number 3: explicit binding

There is no JavaScript developer that at some point in her/his career didn't see code like this:

```
1  someObject.call(anotherObject);  
2  Someobject.prototype.someMethod.apply(someOtherObject);
```

That's **explicit binding** in action. Another example of binding: with the rise of React as the UI library of choice it is common to see class methods "bound" to the class itself (more on that later):

```

1  class Button extends React.Component {
2    constructor(props) {
3      super(props);
4      this.state = { text: "" };
5      // bounded method
6      this.handleClick = this.handleClick.bind(this);
7    }
8
9    handleClick() {
10     this.setState(() => {
11       return { text: "PROCEED TO CHECKOUT" };
12     });
13   }
14
15   render() {
16     return (
17       <button onClick={this.handleClick}>
18         {this.state.text || this.props.text}
19       </button>
20     );
21   }
22 }

```

Nowadays React hooks make classes almost unnecessary, yet there are a lot of “legacy” React components out there using ES2015 classes. One of the question most beginners ask is why we re-bind event handler methods in React with “bind”? Those three methods, `call`, `apply`, `bind` belong to `Function.prototype` and are used for the so called **explicit binding** (rule number 3). It’s in the name: explicit binding means taking a function and forcing its home, also called context object, to be one provided by you. But why would I explicitly bind or re-bind a function? Consider some legacy JavaScript code wrote by a fictional, less JavaScript-savvy colleague some years ago:

```

1  var legacyWidget = {
2    html: "",
3    init: function() {
4      this.html = document.createElement("div");
5    },
6    showModal: function(htmlElement) {
7      var newElement = document.createElement(htmlElement);
8      this.html.appendChild(newElement);
9      window.document.body.appendChild(this.html);
10   }
11 };

```


showModal is a “method” bound to the object legacyWidget. It is implicitly bound (rule number 2), almost useless because apparently there is no way to change the HTML element (`this.html`) on which the modal should append the new element. `this.html` is hardcoded inside the method. No worries, we can resort to explicit binding for altering the `this` object on which showModal runs. It is a job for `call` which takes the new context object and a list of optional arguments. Now we can create a new “shiny” widget and provide a different HTML element as the starting point:

```

1  var legacyWidget = {
2    html: "",
3    init: function() {
4      this.html = document.createElement("div");
5    },
6    showModal: function(htmlElement) {
7      var newElement = document.createElement(htmlElement);
8      this.html.appendChild(newElement);
9      window.document.body.appendChild(this.html);
10   }
11 };
12
13 var shinyNewWidget = {
14   html: "",
15   init: function() {
16     // A different HTML element
17     this.html = document.createElement("section");
18   }
19 };

```

At this point you can call `call` on the original method:

```

1  var legacyWidget = {
2    html: "",
3    init: function() {
4      this.html = document.createElement("div");
5    },
6    showModal: function(htmlElement) {
7      var newElement = document.createElement(htmlElement);
8      this.html.appendChild(newElement);
9      window.document.body.appendChild(this.html);
10   }
11 };
12
13 var shinyNewWidget = {
14   html: "",

```

```
15   init: function() {
16       this.html = document.createElement("section");
17   }
18 };
19
20 // Initialize the new widget with a different HTML element
21 shinyNewWidget.init();
22
23 // Run the original method with a new context object
24 legacyWidget.showModal.call(shinyNewWidget, "p");
```

If you're still confused about explicit binding think of it like a primitive style for reusing code. It looks hacky and buggy to me but that's the best you can do if you've got legacy JavaScript code to refactor. Also, the prototype system is a better candidate for structuring the above code but believe me, bad software like that still exists in production today. At this point you may wonder what `apply` and `bind` are. `apply` has the same effect of using `call` except that the former accepts an array of parameters while the latter expects the parameters to be given as a list. In other words `call` works with a list of parameters:

```
1  var obj = {
2      version: "0.0.1",
3      printParams: function(param1, param2, param3) {
4          console.log(this.version, param1, param2, param3);
5      }
6  };
7
8  var newObj = {
9      version: "0.0.2"
10 };
11
12 obj.printParams.call(newObj, "aa", "bb", "cc");
```

While `apply` expects an array of parameters:

```
1  var obj = {
2    version: "0.0.1",
3    printParams: function(param1, param2, param3) {
4      console.log(this.version, param1, param2, param3);
5    }
6  };
7
8  var newObj = {
9    version: "0.0.2"
10 };
11
12 obj.printParams.apply(newObj, ["aa", "bb", "cc"]);
```

And what about `bind`? That's the most powerful method for binding functions. `bind` still takes a new context object for a given function but it does not just call the function with the new context object. It returns a new function bound to that object permanently:

```
1  var obj = {
2    version: "0.0.1",
3    printParams: function(param1, param2, param3) {
4      console.log(this.version, param1, param2, param3);
5    }
6  };
7
8  var newObj = {
9    version: "0.0.2"
10 };
11
12 var newFunc = obj.printParams.bind(newObj);
13
14 newFunc("aa", "bb", "cc");
```

A common use case for `bind` is a permanent re-binding of an original function:

```
1  var obj = {
2    version: "0.0.1",
3    printParams: function(param1, param2, param3) {
4      console.log(this.version, param1, param2, param3);
5    }
6  };
7
8  var newObj = {
9    version: "0.0.2"
10 };
11
12 obj.printParams = obj.printParams.bind(newObj);
13
14 obj.printParams("aa", "bb", "cc");
```

From now on `obj.printParams` will always refer `newObj` as the object in which the function is running. At this point should be clear why we re-bind class methods in React with `bind`:

```
1  class Button extends React.Component {
2    constructor(props) {
3      super(props);
4      this.state = { text: "" };
5      this.handleClick = this.handleClick.bind(this);
6    }
7
8    handleClick() {
9      this.setState(() => {
10        return { text: "PROCEED TO CHECKOUT" };
11      });
12    }
13
14    render() {
15      return (
16        <button onClick={this.handleClick}>
17          {this.state.text || this.props.text}
18        </button>
19      );
20    }
21  }
```

But the reality is more nuanced and has something to do with “lost binding”. When we assign an event handler as a prop to a React element the method is passed as a reference, not as a function.

To speak in terms of “vanilla” JavaScript it’s like passing the event handler reference inside another callback:

```

1    // BINDING LOST!
2    const handleClick = this.handleClick;
3    //
4    element.addEventListener("click", function() {
5        handleClick();
6    });

```

The assignment operation breaks the binding. In the example component above the method `handleClick` (assigned to a button element) tries to update the component’s state by calling `this.setState()`. When called, the method has already lost its binding which is no longer the class itself: now its context object is the window global object. At that point you get the dreaded “TypeError: Cannot read property ‘setState’ of undefined”. React components are most of the times exported as ES2015 modules: `this` is undefined because ES modules use strict mode by default, thus disabling the default binding. Strict mode is enabled also for ES2015 classes and `this` is undefined even when not exporting the component as an ES module. You can test by yourself with a simple class which mimics a React component. There is a `setState` method, called by `handleClick` in response to a click event:

```

1    class ExampleComponent {
2        constructor() {
3            this.state = { text: "" };
4        }
5
6        handleClick() {
7            this.setState({ text: "New text" });
8            alert(`New state is ${this.state.text}`);
9        }
10
11       setState(newState) {
12           this.state = newState;
13       }
14
15       render() {
16           const element = document.createElement("button");
17           document.body.appendChild(element);
18           const text = document.createTextNode("Click me");
19           element.appendChild(text);
20
21           const handleClick = this.handleClick;
22

```

```
23     element.addEventListener("click", function() {
24         handleClick();
25     });
26 }
27 }
28
29 const component = new ExampleComponent();
30 component.render();
```

The offending line of code is:

```
1  const handleClick = this.handleClick;
```

Try to include the class in an HTML file:

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <title>Loosing the binding</title>
6  </head>
7  <body>
8
9  </body>
10 <script src="this.js"></script>
11 </html>
```

and click the button. Check the console and you'll see "TypeError: Cannot read property 'setState' of undefined". To solve the problem we can use `bind` for making the method stick to the right context, the class itself:

```
1  constructor() {
2      this.state = { text: "" };
3      this.handleClick = this.handleClick.bind(this);
4  }
```

Click the button again and you'll see an alert as expected. Explicit binding is stronger than both implicit binding and default binding. With `apply`, `call`, and `bind` we can bend `this` at our own will by providing a dynamic context object to our functions. If "context object" is still too abstract for you think of it as a box in which JavaScript functions run. The box is always there, but we can change its coordinates at any time with an explicit `bind`.

Rule number 4: “new” binding

TODO

Conclusions

TODO



Sharpen up your JavaScript skills

- Take `legacyWidget` and `shinyNewWidget` from the explicit binding section and try to refactor the code to use prototype.
- What’s stronger between default and explicit binding?
- TODO
- TODO
- TODO

Chapter 7. Types, conversion, and comparison in JavaScript

Primitive types in JavaScript

At the beginning of the book we saw some of the JavaScript's fundamental units like strings and numbers:

```
1 var greet = "Hello";  
2 var year = 89;
```

Strings and numbers are part of the so called “types” of the language, also known as “primitives” (except Object which is a type on its own). The complete list is:

- String
- Number
- Boolean
- Null
- Undefined
- Object
- Symbol (added in ES6)

Booleans represent values that could be either `true` or `false`. `null` on the other hand is the intentional absence of a value. `null` is usually assigned to a variable for signalling that the variable will be later populated with something meaningful.

```
1 var maybe = null;
```

And then there is `undefined` which means that a variable has still nothing attached to it:

```
1 var name;  
2 console.log(name)  
3 undefined
```

`null` and `undefined` look pretty similar yet they are two distinct entities so much that developers are still unsure which one to use.

If you want to find out the type of a JavaScript entity you can use the `typeof` operator. Let's try with a string:


```
1  typeof "alex"
2  "string"
```

with a number:

```
1  typeof 9
2  "number"
```

with a boolean:

```
1  typeof false
2  "boolean"
```

on undefined:

```
1  typeof undefined
2  "undefined"
```

and with null:

```
1  typeof null
2  "object"
```

which gives us a suprising result! `null` looks like an object but in reality it is an historic bug in JavaScript, lying there since the language was born. JavaScript has always had a bad reputation because of these things. And that's just the beginning. There are some strange rules for converting between one type and another. Let me give you a bit of context. Let's make an example in Python. The following instruction in Python:

```
1  'hello' + 89
```

gives you a clear error:

```
1  TypeError: can only concatenate str (not "int") to str
```

While in JavaScript only the sky is your limit:

```
1  'hello' + 89
```

in fact gives:

```
1 "hello89"
```

Things looks even more stranger if we try adding an array to a string:

```
1 'hello' + []
```

gives:

```
1 'hello'
```

and:

```
1 'hello' + [89]
```

gives a surprising:

```
1 "hello89"
```

Looks like there is some kind of logic behind this conversion. It works even with more crowded arrays:

```
1 'hello' + [89, 150.156, 'mike']
```

gives:

```
1 "hello89,150.156,mike"
```

These two lines of JavaScript are enough to make a Java developer run away. But this behaviour in JavaScript is 100% intentional. So it is worth exploring the most glaring cases of implicit conversion in JavaScript, also know as type coercion.

When a number becomes a string

Some programming languages have a concept called type casting which means more or less: if I want to convert an entity to another type then I have to make the conversion clear-cut. It is possible in JavaScript too. Consider the following example:

```
1 var greet = "Hello";
2 var year = 89;
```

If I want to convert explicitly I can signal the intention in my code, either with `toString()`:

```
1 var greet = "Hello";
2 var year = 89;
3
4 var yearString = year.toString()
```

or with `String`:

```
1 var greet = "Hello";
2 var year = 89;
3
4 var yearString = String(year)
```

`String` is part of the so called built-in JavaScript objects which mirror some of the primitive types: `String`, `Number`, `Boolean`, and `Object`. These built-ins can be used for converting between types. After the conversion I can concatenate the two variables:

```
1 greet + yearString;
```

But apart from this explicit conversion, in JavaScript there is a subtle mechanic called implicit conversion, kindly offered by JavaScript engines. The language does not prevent us from adding numbers and strings:

```
1 'hello' + 89
```

gives:

```
1 "hello89"
```

But what's the logic behind this conversion? You may be surprised to find out that the addition operator `+` in JavaScript automatically converts any of the two operands to a string if at least one of them is... a string!

And you may find even more surprising that this rule is set in stone in the ECMAScript spec. [Section 11.6.1](http://www.ecma-international.org/ecma-262/5.1/#sec-11.6.1)¹⁸ defines the behavior of the addition operator which I'll summarize here for your own sake:

The addition operator (+)

If `x` is `String` or `y` is `String` then return `ToString(x)` followed by `ToString(y)`

Does this trick work only on numbers? Not really. Array and objects are subject to the same fate:

¹⁸<http://www.ecma-international.org/ecma-262/5.1/#sec-11.6.1>

```
1 'hello' + [89, 150.156, 'mike']
```

gives:

```
1 "hello89,150.156,mike"
```

And how about:

```
1 'hello' + { name: "Jacopo" }
```

To find the solution you can do a quick test by converting the object to a string:

```
1 String({ name: "Jacopo" })
```

which gives:

```
1 "[object Object]"
```

So I've a feeling that:

```
1 'hello' + { name: "Jacopo" }
```

will give:

```
1 "hello[object Object]"
```

But another question remains: what happens with multiplication, division and subtraction?



What's the meaning of [object Object]?

Almost every JavaScript entity has a method called `toString()` which in some ways is a courtesy of `Object.toString()`. Some types like arrays implement a custom version of `toString()` so that the value gets converted to string when the method is called. For example `Array.prototype.toString` overwrites the original `Object.toString()` (this practice is called shadowing). But when you call `toString()` on a plain JavaScript object the engine gives “[object Object]” because the default behaviour of `Object.toString()` is to return the string object followed by the entity type (Object in this case).

I'm not a number!

We saw how the addition operator tries to convert the operands to a string when at least one of them is a string. But what happens with the other arithmetic operators? In JavaScript we have:

| | |
|----|--|
| + | sum |
| ++ | increment |
| * | multiplication |
| ** | exponent (added in ES6) |
| - | subtraction |
| -- | decrement |
| / | division |
| % | module (returns the remainder of a division) |

If we use one of these operators (except +) against a type that it's not a number then we get a special kind of JavaScript object called NaN:

```
1 89 ** "alex"
```

in fact gives:

```
1 NaN
```

NaN stands for not a number and represents any failed arithmetic operation, like in the following code:

```
1 var obj = { name: "Jacopo" } % 508897
```

which gives:

```
1 console.log(obj)
2 NaN
```

Pay attention to `typeof` in combination with NaN. This code looks ok:

```
1 typeof 9 / "alex"
2 NaN
```

And how about the following?

```
1 var strange = 9 / "alex"

1 typeof strange
2 "number"
```

NaN becomes `number` when it is assigned to a variable. This opens up a new question. How can I check whether some variable is NaN or not? There is a new method in ES6 called `isNaN()`:

```
1 var strange = 9 / "alex"
```

```
1 isNaN(strange)
```

```
2 true
```

But now let's point our attention to JavaScript comparison operators which are as weird as their arithmetic counterparts.

Equal equal or not?

There are two main families of comparison operators in JavaScript. First we have what I like to call “weak comparison”. It's the abstract comparison operator (double equal): `==`. Then there's a “strong comparison” which you can recognize from the triple equal: `===` also called strict comparison operator. Both of them behave in a different way from the other. Let's see some examples. First thing first if we compare two strings with both operators we get consistent results:

```
1 "hello" == "hello"
```

```
2 true
```

```
3
```

```
4 "hello" === "hello"
```

```
5 true
```

Everything looks fine. Now let's try to compare two different types, a number and a string. First with the “strong comparison”:

```
1 "1" === 1
```

```
2 false
```

Makes sense! The string “1” is different from the number 1. But what happens with a “weak comparison”?

```
1 "1" == 1
```

```
2 true
```

True! JavaScript is saying that the two values are equal. Now it doesn't make any sense. Unless this behaviour has something to do with the implicit conversion we saw earlier. What if the same rules apply? Bingo. The ECMAScript [spec](http://www.ecma-international.org/ecma-262/5.1/#sec-11.9.3)¹⁹ strikes again. Turns out the abstract comparison operator makes an automatic conversion between types, before comparing them. This is an abstract:

¹⁹<http://www.ecma-international.org/ecma-262/5.1/#sec-11.9.3>

Abstract Equality Comparison Algorithm

The comparison `x == y` is performed as follows: if `x` is `String` and `y` is `Number` return the result of the comparison `ToNumber(x) == y`

The spec says: if the first operand is a string and the second operand is a number then convert the first operand to a number. Interesting. The JavaScript spec is full of this crazy rules and I highly encourage digging deeper into it. In the meantime unless you have a strong reason to do otherwise avoid the abstract comparison operator in your JavaScript code. You will thank yourself later. And how about the “strong comparison”? The spec at [Strict Equality Comparison](#)²⁰ says that no automatic conversion is made before comparing values with the triple equal `===`. And using the Strict Equality Comparison in your code will save yourself from silly bugs.

Primitives and objects

We already saw JavaScript’s building blocks: `String`, `Number`, `Boolean`, `Null`, `Undefined`, `Object` and `Symbol`. They are all capitalized and this style appears even in the [ECMAScript spec](#)²¹. But besides these primitives there are a some twins mirroring the primitives: the so called **built-in objects**. The `String` primitive for example has an equivalent `String` which is used in two ways. If called like a function (by passing an argument) it converts any value into a string:

```
1 var someValue = 555;
2
3 String(someValue);
4
5 "555"
```

If we instead use `String` as a constructor with `new` the result is an object of type string:

```
1 var someValue = 555;
2
3 var newString = new String(someValue);
```

It is worth inspecting the object in the console for seeing how different from a “plain” string is:

²⁰<http://www.ecma-international.org/ecma-262/5.1/#sec-11.9.6>

²¹<http://www.ecma-international.org/ecma-262/5.1/#sec-8>

```
> var someValue = 555;

    var newString = new String(someValue);
< undefined

> newString
< ▼ String {"555"} ⓘ
  0: "5"
  1: "5"
  2: "5"
  length: 3
  ▶ __proto__: String
  [[PrimitiveValue]]: "555"

> |
```

JavaScript String Object

You can use `typeof` for confirming that it is indeed an object:

```
1 typeof newString
2 "object"
```



Do you remember “new” in JavaScript?

`new` in JavaScript stands for “constructor”. The syntax `new FunctionName` is said to be a constructor call. We saw `new` in chapter 5.

Other “mirrors” of the primitive types are `Number` which can convert (almost) any value to a number:

```
1 var someValue = "555";
2
3 Number(someValue);
4
5 555;
```

I said almost because you get `NaN` when trying to convert an invalid “number”:

```
1 var notValidNumber = "aa555";
2
3 Number(notValidNumber);
4
5 NaN;
```

When used in the constructor form `Number` returns a new object of type number:


```
1 var someValue = "555";
2
3 new Number(someValue);
4
5 Number {555}
```

The Boolean built-in behaves the same and can convert any value to a boolean when used as a function

```
1 var convertMe = 'alex';
2
3 Boolean(convertMe)
4
5 true
```

While the constructor returns an object (what a surprise!):

```
1 var convertMe = 'alex';
2
3 typeof new Boolean(convertMe)
4
5 "object"
```

Object makes no exceptions:

```
1 Object('hello'); // String { "hello" }
2
3 Object(1); // Number { 1 }
```

If called without arguments it returns an empty object:

```
1 Object()
2
3 {}
```

While if called as a constructor it returns a new object:

```
1 new Object({
2   name: "Alex",
3   age: 33
4 });
5
6 {name: "Alex", age: 33}
```

At this point you might ask yourself: when it is fine to use the built-in objects and when should I initialize values with a primitive?. As a rule of thumb avoid the constructor call when all you need is a simple primitive:

```
1 // AVOID THIS
2 var bool = new Boolean("alex");
3 var str = new String('hi');
4 var num = new Number(33);
5 var strObj = new Object('hello')
```

Beside being impractical this form imposes a performance penalty because we create a new object every single time. Last but not least it doesn't make sense to create an object when we want a simple string or a number. So the following form is preferred:

```
1 // OK
2 var bool = true
3 var str = 'hi';
4 var num = 33;
5 var obj = {
6     name: "Alex",
7     age: 33
8 };
```

You can use Boolean, String, and Number like a function every time there's the need to convert something:

```
1 // OK FOR CONVERTING
2 var boolFromString = Boolean("alex");
3 var numFromString = Number('22');
```

Conclusions

There are seven building blocks in JavaScript, namely String, Number, Boolean, Null, Undefined, Object, and Symbol. These types are called primitives. JavaScript developers can manipulate these

types with arithmetic and comparison operators. But we need to pay particular attention to both the addition operator `+` and the abstract comparison operator `==` which by nature tend to convert between types. This implicit conversion in JavaScript is called type coercion and is defined in the ECMAScript spec. Whenever possible in your code use always the strict comparison operator `===` instead of `==`. As a best practice always make clear when you intend to convert between one type and another. For this purpose JavaScript has a bunch of built-in objects which mirror the primitive types: `String`, `Number`, `Boolean`. These built-in can be used for converting explicitly between different types.



Sharpen up your JavaScript skills

- What's the output of `44 - "alex"`?
- What's the output of `"alex" + 44` and why?
- What's the output of `"alex" + { name: "Alex" }`?
- What's the output of `["alex"] == "alex"` and why?
- What's the result of `undefined == null` and why?

Part 2 - Working with JavaScript

Chapter 8. Manipulating HTML elements with JavaScript

The Document Object Model

JavaScript is not that bad. As a scripting language running in the browser it is really useful for manipulating web pages. In this chapter we'll see what function we have for interacting and modifying an HTML document and its elements. But first let's demystify the Document Object Model.

The Document Object Model is a fundamental concept at the base of everything we do inside the browser. But what exactly is that? When we visit a web page the browser figures out how to interpret every HTML element. So that it creates a virtual representation of the HTML document, saved in memory. The HTML page is converted to a tree-like structure and every HTML element becomes a leaf, connected to a parent branch. Consider this simple HTML page:

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <title>A super simple title!</title>
5 </head>
6 <body>
7 <h1>A super simple web page!</h1>
8 </body>
9 </html>
```

When the browser scan through the HTML above it creates a Document Object Model which is a mirror of our HTML structure.

At the top of this structure there is document also called root element, which contains another element: `html`. The `html` element contains an `head` which in turn has a `title`. Then we have `body` containing an `h1`. Every HTML element is represented by a specific type (also called interface) and may contains text or other nested elements:

```

1 document (HTMLDocument)
2   |
3   | --> html (HTMLHtmlElement)
4     |
5     | --> head (HtmlHeadElement)
6       |
7       | --> title (HtmlTitleElement)
8         | --> text: "A super simple title!"
9       |
10      | --> body (HtmlBodyElement)
11        |
12        | --> h1 (HTMLHeadingElement)
13          | --> text: "A super simple web page!"

```

Every HTML element descends from [Element](#)²² but a large part of them specializes further. You can inspect the prototype for finding out to what “species” an element belongs to. The h1 element for example is an HTMLHeadingElement:

```

1 document.querySelector('h1').__proto__
2
3 // Output: HTMLHeadingElement

```

HTMLHeadingElement in turn is a “descendant” of HTMLElement:

```

1 document.querySelector('h1').__proto__.__proto__
2
3 // Output: HTMLElement

```



__proto__ and prototype

In chapter 5 we saw two methods for inspecting the ancestor of a given object: `__proto__` and `getPrototypeOf`

At this point (especially for beginners) there could be some confusion between `document` and `window`. Well, `window` refers to the browser while `document` on the other hand is the HTML page you’re currently operating. `window` is a global object, directly accessible from any JavaScript code running in the browser. It is not a JavaScript “native” object but rather exposed by the browser itself. `window` has a lot of properties and methods, here are some:

²²<https://developer.mozilla.org/en-US/docs/Web/API/Element>

```
1 window.alert('Hello world'); // Shows an alert
2 window.setTimeout(callback, 3000); // Delay execution
3 window.fetch(someUrl); // make XHR requests
4 window.open(); // Opens a new tab
5 window.location; // Browser location
6 window.history; // Browser history
7 window.navigator; // The actual device
8 window.document; // The current page
```

Since these properties and methods are also global you can omit `window.` and access them directly:

```
1 alert('Hello world'); // Shows an alert
2 setTimeout(callback, 3000); // Delay execution
3 fetch(someUrl); // make XHR requests
4 open(); // Opens a new tab
5 location; // Browser location
6 history; // Browser history
7 navigator; // The actual device
8 document; // The current page
```

You should already be familiar with some of these methods like `setTimeout()` (chapter 3) or `window.navigator`, useful when you want to sniff the browser's language of your users:

```
1 if (window.navigator) {
2     var lang = window.navigator.language;
3     if (lang === "en-US") {
4         // show something
5     }
6
7     if (lang === "it-IT") {
8         // show something else
9     }
10 }
```

There are a lot more methods though and would be impossible to cover everything here. For learning more make sure to check out [MDN Docs](https://developer.mozilla.org/en-US/docs/Web/API#Interfaces)²³.

Nodes, elements, and DOM manipulation

The document interface has a number of utilities, like `querySelector()`, a method for selecting any HTML element inside a given page:

²³<https://developer.mozilla.org/en-US/docs/Web/API#Interfaces>

```
1 document.querySelector('h1');
```

window represents the current window's browser and the following instruction is the same as above:

```
1 window.document.querySelector('h1');
```

Anyhow the following syntax is more common and we'll use it a lot in the next sections:

```
1 document.methodName();
```

Other than `querySelector()` for selecting HTML elements we have a lot more useful methods:

```
1 // returns a single element
2 document.getElementById('testimonials');
3
4 // returns an HTMLCollection
5 document.getElementsByTagName('p');
6
7 // returns a NodeList
8 document.querySelectorAll('p');
```

Not only you can select HTML elements, one can also interact and modify their inner state. For example you may want to read or change the inner content of a given element:

```
1 // Read or write
2 document.querySelector('h1').innerHTML;
3 document.querySelector('h1').innerHTML = ''; // Ouch!
```

Every HTML element of the DOM is also a “Node”, in fact we can inspect the node type like so:

```
1 document.querySelector('h1').nodeType;
```

The above code returns 1 which is the identifier for nodes of type Element. You can also inspect the node name:

```
1 document.querySelector('h1').nodeName;
2
3 "H1"
```

The above example returns the node name in uppercase. The most important concept to understand is that we mainly work with four types of node in the DOM:

- document: the root node (nodeType 9)
- nodes of type Element: the actual HTML tags (nodeType 1)
- node of type attribute: the property (attribute) of every HTML element
- nodes of type Text: the actual text content of an element (nodeType 3)

Since elements are node and nodes can have properties (also called attributes) JavaScript developers can inspect and manipulate said properties:

```
1 // Returns true or false
2 document.querySelector('a').hasAttribute('href');
3
4 // Returns the attribute text content, or null
5 document.querySelector('a').getAttribute('href');
6
7 // Sets the given attribute
8 document.querySelector('a').setAttribute('href', 'someLink');
```

Earlier we said that the DOM is a tree-like structure. This trait reflects on HTML elements too. Every element may have parents and children and we can look at them by inspecting certain properties of the element:

```
1 // Returns an HTMLCollection
2 document.children;
3
4 // Returns a NodeList
5 document.childNodes;
6
7 // Returns a Node
8 document.querySelector('a').parentNode;
9
10 // Returns an HTML element
11 document.querySelector('a').parentElement;
```

So far we saw how to select and how to interrogate HTML elements. Now how about making some? For creating new nodes of type Element the native DOM API offers the `createElement` method which you'll call like so:

```
1 var heading = document.createElement('h1');
```

And for creating text we use `createTextNode`:

```
1 var text = document.createTextNode('Hello world');
```

The two nodes can be combined together by appending the text inside a new HTML element. Last but not least we can also append the heading element to the root document:

```
1 var heading = document.createElement('h1');
2 var text = document.createTextNode('Hello world');
3 heading.appendChild(text);
4 document.body.appendChild(heading);
```

The methods we saw so far are all you need to know for start working with JavaScript in the browser. In nerd circles we refer to these instructions as DOM manipulation. Let's see what the DOM can do for us.

Generating HTML tables with JavaScript

HTML is a static markup language. After defining some HTML elements there is no way to modify them (besides using CSS). JavaScript will become our friend here and you'll do a simple exercise for refreshing your JavaScript skills. You may wonder "why should I use pure JavaScript and not jQuery?". Well, be aware that jQuery is fading away. Bootstrap 5 will remove it from the dependencies and many more are dropping it as well. There is a valid reason behind this: the native DOM API is complete and mature enough to make [jQuery obsolete](http://youmightnotneedjquery.com/)²⁴.

Now let's begin coding. Given an array of objects we want to dynamically generate an HTML table. An HTML table is represented by a `<table>` element. Every table can have an header as well, defined by a `<thead>` element. The header can have one or more rows, `<tr>`, and every header's row has a cell, represented by a `<th>` element. Like so:

```
1 <table>
2   <thead>
3     <tr>
4       <th>name</th>
5       <th>height</th>
6       <th>place</th>
7     </tr>
8   </thead>
9   <!-- more stuff here! -->
10 </table>
```

But there's more. Every table has a body most of the time, defined by `<tbody>` which in turn contains a bunch of rows, `<tr>`. Every row can have cells containing the actual data. Table cells are defined by `<td>`. Here's how a complete table looks like:

²⁴<http://youmightnotneedjquery.com/>

```

1  <table>
2    <thead>
3    <tr>
4      <th>name</th>
5      <th>height</th>
6      <th>place</th>
7    </tr>
8  </thead>
9  <tbody>
10 <tr>
11   <td>Monte Falco</td>
12   <td>1658</td>
13   <td>Parco Foreste Casentinesi</td>
14 </tr>
15 <tr>
16   <td>Monte Falterona</td>
17   <td>1654</td>
18   <td>Parco Foreste Casentinesi</td>
19 </tr>
20 </tbody>
21 </table>

```

Our task now is to generate the table starting from an array of JavaScript objects. To start off create a new file named **build-table.html** and save it in a folder of choice:

```

1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4    <meta charset="UTF-8">
5    <title>Build a table</title>
6  </head>
7  <body>
8    <table>
9    <!-- here goes our data! -->
10 </table>
11 </body>
12 <script src="build-table.js"></script>
13 </html>

```



Practice makes perfect

I suggest not copy pasting the code you'll find in this book. Practicing is always better than copying.

Create another file named **build-table.js** in the same folder and start the code with the following array (remember, "use strict" should appear at the top of every new JavaScript file you write):

```
1  "use strict";
2
3  var mountains = [
4    { name: "Monte Falco", height: 1658, place: "Parco Foreste Casentinesi" },
5    { name: "Monte Falterona", height: 1654, place: "Parco Foreste Casentinesi" },
6    { name: "Poggio Scali", height: 1520, place: "Parco Foreste Casentinesi" },
7    { name: "Pratomagno", height: 1592, place: "Parco Foreste Casentinesi" },
8    { name: "Monte Amiata", height: 1738, place: "Siena" }
9  ];
```

Now let's think about the table. First thing first we need a `<thead>`. You might be tempted to use `createElement`:

```
1  document.createElement('thead')
```

It's not wrong but a closer look at the [MDN documentation for table²⁵](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/table) reveals an intriguing detail. `<table>` is an [HTMLTableElement²⁶](https://developer.mozilla.org/en-US/docs/Web/API/HTMLTableElement), an interface containing some interesting methods. One of the most useful is `HTMLTableElement.createTHead()` which can help in creating our `<thead>`.

So let's start with a first JavaScript function called `generateTableHead`:

```
1  function generateTableHead(table) {
2    var thead = table.createTHead();
3  }
```

Our function takes a selector and creates a `<thead>` on the given table:

```
1  function generateTableHead(table) {
2    var thead = table.createTHead();
3  }
4
5  var table = document.querySelector("table");
6
7  generateTableHead(table);
```

Open **build-table.html** in a browser: there's nothing! But if you open up the browser console you can see a new `<thead>` attached to the table. Time to populate the header! First we have to create a row inside it. There is another method that can help: `HTMLTableElement.insertRow()`. With this info we can extend our function:

²⁵<https://developer.mozilla.org/en-US/docs/Web/HTML/Element/table>

²⁶<https://developer.mozilla.org/en-US/docs/Web/API/HTMLTableElement>

```
1 function generateTableHead(table) {  
2   var thead = table.createTHead();  
3   var row = thead.insertRow();  
4 }
```

At this point we can generate our rows. By looking at the source array we can see that any object in it has the info we're looking for: name, height, place:

```
1 var mountains = [  
2   { name: "Monte Falco", height: 1658, place: "Parco Foreste Casentinesi" },  
3   { name: "Monte Falterona", height: 1654, place: "Parco Foreste Casentinesi" },  
4   { name: "Poggio Scali", height: 1520, place: "Parco Foreste Casentinesi" },  
5   { name: "Pratomagno", height: 1592, place: "Parco Foreste Casentinesi" },  
6   { name: "Monte Amiata", height: 1738, place: "Siena" }  
7   ];
```

That means we can pass another parameter to our function: an array to iterate over for generating header cells:

```
1 function generateTableHead(table, data) {  
2   var thead = table.createTHead();  
3   var row = thead.insertRow();  
4   for (var i = 0; i < data.length; i++) {  
5     var th = document.createElement("th");  
6     var text = document.createTextNode(data[i]);  
7     th.appendChild(text);  
8     row.appendChild(th);  
9   }  
10 }
```

Unfortunately there is no native method for creating header cells so I'll resort to `document.createElement("th")`. Also worth noting `document.createTextNode(data[i])` for creating text nodes and `appendChild()` for appending our new elements to every tag. When creating and manipulating elements this way we talk of “imperative” DOM manipulation. Modern front-end libraries are tackling this problem by favoring a “declarative” approach. Instead of commanding the browser step by step we can declare what HTML elements do we need and the library takes care of the rest.

Back to our code we can use our first function like so:

```
1 var table = document.querySelector("table");
2 var data = Object.keys(mountains[0]);
3 generateTableHead(table, data);
```

Now we can move further to generating the actual table's data. Our next function will implement a logic similar to `generateTableHead` but this time we need two nested for loops. In the innermost loop we'll exploit another native method for creating a series of `td`. The method is `HTMLTableRowElement.insertCell()`. Add another function named `generateTable` in the same file we created earlier:

```
1 function generateTable(table, data) {
2   for (var i = 0; i < data.length; i++) {
3     var row = table.insertRow();
4     for (var key in data[i]) {
5       var cell = row.insertCell();
6       var text = document.createTextNode(data[i][key]);
7       cell.appendChild(text);
8     }
9   }
10 }
```

The above function is called passing an HTML table and an array of objects as arguments:

```
1 generateTable(table, mountains);
```

Let's dig a bit into the logic for `generateTable`. The parameter `data` is an array corresponding to `mountains`. The outermost for loop cycles through the array and creates a row for every element:

```
1 function generateTable(table, data) {
2   for (var i = 0; i < data.length; i++) {
3     var row = table.insertRow();
4     // omitted for brevity
5   }
6 }
```

The innermost loop cycles through every key of any given object and for every object creates a text node containing the key's value:

```
1 function generateTable(table, data) {
2   for (var i = 0; i < data.length; i++) {
3     var row = table.insertRow();
4     for (var key in data[i]) {
5       // inner loop
6       var cell = row.insertCell();
7       var text = document.createTextNode(data[i][key]);
8       cell.appendChild(text);
9     }
10  }
11 }
```

If you followed every step you should end up with the following code:

```
1 var mountains = [
2   { name: "Monte Falco", height: 1658, place: "Parco Foreste Casentinesi" },
3   { name: "Monte Falterona", height: 1654, place: "Parco Foreste Casentinesi" },
4   { name: "Poggio Scali", height: 1520, place: "Parco Foreste Casentinesi" },
5   { name: "Pratomagno", height: 1592, place: "Parco Foreste Casentinesi" },
6   { name: "Monte Amiata", height: 1738, place: "Siena" }
7 ];
8
9 function generateTableHead(table, data) {
10  var thead = table.createTHead();
11  var row = thead.insertRow();
12  for (var i = 0; i < data.length; i++) {
13    var th = document.createElement("th");
14    var text = document.createTextNode(data[i]);
15    th.appendChild(text);
16    row.appendChild(th);
17  }
18 }
19
20 function generateTable(table, data) {
21  for (var i = 0; i < data.length; i++) {
22    var row = table.insertRow();
23    for (var key in data[i]) {
24      var cell = row.insertCell();
25      var text = document.createTextNode(data[i][key]);
26      cell.appendChild(text);
27    }
28  }
29 }
```

Which called with:

```
1 var table = document.querySelector("table");
2 var data = Object.keys(mountains[0]);
3 generateTable(table, mountains);
4 generateTableHead(table, data);
```

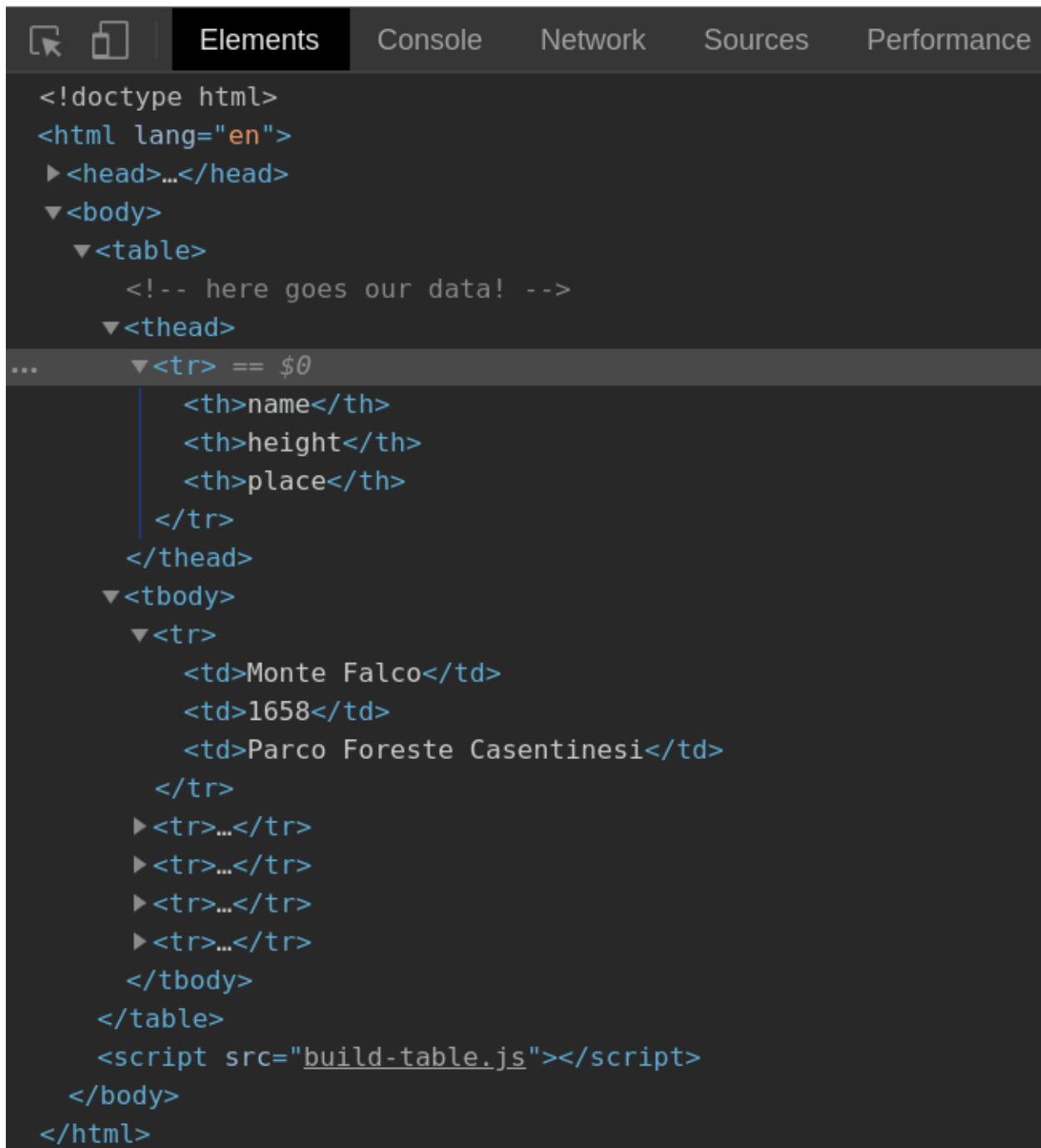
will produce our table!



TIP

Pay attention to the call order of the functions. Calling `generateTableHead` before `generateTable` will add every row to `<thead>` instead of `<tbody>`!

| name | height | place |
|-----------------|--------|---------------------------|
| Monte Falco | 1658 | Parco Foreste Casentinesi |
| Monte Falterona | 1654 | Parco Foreste Casentinesi |
| Poggio Scali | 1520 | Parco Foreste Casentinesi |
| Pratomagno | 1592 | Parco Foreste Casentinesi |
| Monte Amiata | 1738 | Siena |



```

<!doctype html>
<html lang="en">
  <head>...</head>
  <body>
    <table>
      <!-- here goes our data! -->
      <thead>
        <tr> == $0
          <th>name</th>
          <th>height</th>
          <th>place</th>
        </tr>
      </thead>
      <tbody>
        <tr>
          <td>Monte Falco</td>
          <td>1658</td>
          <td>Parco Foreste Casentinesi</td>
        </tr>
        <tr>...</tr>
        <tr>...</tr>
        <tr>...</tr>
        <tr>...</tr>
      </tbody>
    </table>
    <script src="build-table.js"></script>
  </body>
</html>

```

The resulting HTML table

I know what you're thinking. The code we wrote so far can be improved. Hold tight, in the next chapter we'll cover refactoring and code organization. But first time for another exercise!

How to dynamically add DOM elements with JavaScript

There was a time when jQuery dominated the frontend world (and still it is to some extents). Then came React, emerging as the library of choice for building UIs. jQuery, React, Vue, Svelte, Angular, are all “better” abstractions over the Document Object Model but that doesn't mean you should completely forget the native API. Most beginners don't know how to dynamically add DOM elements with “vanilla” JavaScript because they reach straight for complex libraries, completely ignoring the native options. In the following exercise you'll learn how to build your mini-library for dynamically generating DOM elements.

React became so popular because allows developers to “declare” interfaces. Declaring means describing how the UI should look instead of ordering the browser how it should build those elements. In React you won't call `document.createElement()` by hand unless you have valid reasons to do so. But libraries like React do not make vanilla JavaScript “unnecessary”, it's just that they take care of the hard work for us. Turns out you can too build your mini-library, kind of like a “factory” for DOM elements. And in the process you can learn how serious libraries deal with the DOM. To start off let's define how a declarative DOM would look like. In its simplest form could be a list of HTML tags:

```
1  const elements = ["div", "p", "h1"];
```

However the above approach doesn't seem to scale well. What if I need some text inside each element? What if an element takes some children? The array is fine because we can iterate over, but elements would be better suited in a JavaScript object. Every object can have a key named “tag” defining what element we'll build:

```
1  const elements = [  
2    {  
3      tag: "div"  
4    },  
5    {  
6      tag: "h1"  
7    },  
8    { tag: "p" }  
9  ];
```

Ooh, way better! Now you can also add another key named `text` for tags that should have a text node:

```
1  const elements = [  
2    {  
3      tag: "div"  
4    },  
5    {  
6      tag: "h1",  
7      text: "An educational JavaScript tutorial"  
8    },  
9    { tag: "p", text: "Building a silly library" }  
10 ];
```

And with this simple structure in place it's time to build a JavaScript function. With the knowledge you gained from the previous section where I talked about `createElement()` and `createTextNode()` it should be easy to build a simple page. For doing so we'll build a new shiny JavaScript library called "Dommy".

Dommy is an element factory. Nothing fancy though: it's just a JavaScript function! Dommy does a bunch of things:

- takes a target element
- builds elements
- appends elements to the given target

Target will be another DOM element and elements is an array of objects. So a first loop will iterate over each element and for each tag we'll create a new HTML element. Here's how I would write the function:

```
1  function dommy(target, elements) {  
2    for (const element of elements) {  
3      const newEl = document.createElement(element.tag);  
4      target.appendChild(newEl);  
5    }  
6  }
```

Easy? We're using two native APIs: `createElement()` and `appendChild()`. Only thing is that I'm not accounting for the case when element has a text. For that to work I need to add `createTextNode()` and a check for the text property (it is a good practice to avoid creating elements unless necessary):

```
1 function dommy(target, elements) {
2   for (const element of elements) {
3     const newEl = document.createElement(element.tag);
4     target.appendChild(newEl);
5     if (element.text) {
6       const text = document.createTextNode(element.text);
7       newEl.appendChild(text);
8     }
9   }
10 }
```

You can already test the code in an HTML file:

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <title>Dommy</title>
6 </head>
7 <body>
8
9 </body>
10 <script src="dynamically-create-elements.js"></script>
11 </html>
```

and here's the complete code:

```
1 const elements = [
2   {
3     tag: "div"
4   },
5   {
6     tag: "h1",
7     text: "An educational JavaScript tutorial"
8   },
9   { tag: "p", text: "Building a silly library" }
10 ];
11
12 function dommy(target, elements) {
13   for (const element of elements) {
14     const newEl = document.createElement(element.tag);
15     target.appendChild(newEl);
```

```
16     if (element.text) {
17         const text = document.createTextNode(element.text);
18         newEl.appendChild(text);
19     }
20 }
21 }
22
23 const target = document.body;
24
25 dommy(target, elements);
```

By opening the HTML in a browser you should see an header and a paragraph with the corresponding text. Good job! But still your “JavaScript frontend library” is quite useless because the goal is having children inside other elements. Let’s fix that. As a first approach we can add a property named “children” into our objects. Since every HTML element may have a certain amount of children it makes sense to provide them as an array. To test things out you can append some children to “div”. Take a look:

```
1  const elements = [
2    {
3      tag: "div",
4      children: [
5        {
6          tag: "h1",
7          text: "An educational JavaScript tutorial"
8        },
9        { tag: "p", text: "Building a silly library" }
10     ]
11   }
12 ];
```

Now h1 and p are children of div. You can also notice that children has the same shape of elements: it is an array of objects. “Valentino are you saying that ... recursion ... is ...” yes. You can do it! First let’s add a check for the children property inside our function:

```
1  function dommy(target, elements) {
2    for (const element of elements) {
3      const newEl = document.createElement(element.tag);
4      target.appendChild(newEl);
5      if (element.text) {
6        const text = document.createTextNode(element.text);
7        newEl.appendChild(text);
8      }
9      if (element.children) {
10         // if there are children ...
11      }
12    }
13 }
```

And now nothing stops you from adding a recursive call to “dommy” itself:

```
1  function dommy(target, elements) {
2    for (const element of elements) {
3      const newEl = document.createElement(element.tag);
4      target.appendChild(newEl);
5      if (element.text) {
6        const text = document.createTextNode(element.text);
7        newEl.appendChild(text);
8      }
9      if (element.children) {
10         dommy(newEl, element.children);
11      }
12    }
13 }
```

Now we’re saying: I’ll call myself when there is a property named children inside element. And while calling myself recursively I pass newEl as a target and element.children as the array of elements to build.

Believe it or not Dommy works even on children of a children. Test the code yourself:

```
1  const elements = [  
2    {  
3      tag: "div",  
4      children: [  
5        {  
6          tag: "h1",  
7          text: "An educational JavaScript tutorial"  
8        },  
9        { tag: "p", text: "Building a silly library" },  
10       { tag: "div", children: [{ tag: "p", text: "Children of a children" }] }  
11     ]  
12   }  
13 ];  
14  
15 function dommy(target, elements) {  
16   for (const element of elements) {  
17     const newEl = document.createElement(element.tag);  
18     target.appendChild(newEl);  
19     if (element.text) {  
20       const text = document.createTextNode(element.text);  
21       newEl.appendChild(text);  
22     }  
23     if (element.children) {  
24       dommy(newEl, element.children);  
25     }  
26   }  
27 }  
28  
29 const target = document.body;  
30  
31 dommy(target, elements);
```

Running the above code will produce the following HTML structure:

```
1  <body>  
2    <div>  
3      <h1>An educational JavaScript tutorial</h1>  
4      <p>Building a silly library</p>  
5      <div><p>Children of a children</p></div>  
6    </div>  
7  </body>
```

With a somehow basic function you can dynamically add DOM elements starting from a declaration.

Declare what elements you need and the “library” builds them for you. Fantastic. Another nice functionality I would add now are attributes. HTML elements may have attributes. The anchor element for example may have the “href” attribute. Adding attributes to an existing HTML element with JavaScript means using `setAttribute()`:

```
1 document.querySelector('a').setAttribute("href", "https://www.valentinog.com/blog/");
```

Our function can be extended to add attributes on a given element. To make things fancy I’m going to add another key named “props” to a couple of existing elements:

```
1 const elements = [  
2   {  
3     tag: "div",  
4     children: [  
5       {  
6         tag: "h1",  
7         text: "An educational JavaScript tutorial"  
8       },  
9       { tag: "p", text: "Building a silly library" },  
10      {  
11        tag: "div",  
12        props: { class: "flex-row" },  
13        children: [{ tag: "p", text: "Children of a children" }]  
14      }  
15    ]  
16  },  
17  {  
18    tag: "a",  
19    text: "A blog",  
20    props: {  
21      href: "https://www.valentinog.com/blog/",  
22      target: "_blank",  
23      rel: "noopener noreferrer"  
24    }  
25  }  
26 ];
```

Next up let’s add a check for the props key into Dommy:


```
1 function dommy(target, elements) {
2   for (const element of elements) {
3     const newEl = document.createElement(element.tag);
4     target.appendChild(newEl);
5     if (element.text) {
6       const text = document.createTextNode(element.text);
7       newEl.appendChild(text);
8     }
9     if (element.props) {
10      // if there are props ...
11    }
12    if (element.children) {
13      dommy(newEl, element.children);
14    }
15  }
16 }
```

Since props is an object I can iterate over its keys with a for...in loop. Inside the loop we're going to add all the attributes to the element:

```
1 function dommy(target, elements) {
2   for (const element of elements) {
3     const newEl = document.createElement(element.tag);
4     target.appendChild(newEl);
5     if (element.text) {
6       const text = document.createTextNode(element.text);
7       newEl.appendChild(text);
8     }
9     if (element.props) {
10      for (key in element.props) {
11        newEl.setAttribute(key, element.props[key]);
12      }
13    }
14    if (element.children) {
15      dommy(newEl, element.children);
16    }
17  }
18 }
```

The above code produces the following HTML if you did everything correctly:

```
1 <body>
2   <div>
3     <h1>An educational JavaScript tutorial</h1>
4     <p>Building a silly library</p>
5     <div class="flex-row"><p>Children of a children</p></div>
6   </div>
7   <a
8     href="https://www.valentinog.com/blog/"
9     target="_blank"
10    rel="noopener noreferrer"
11    >A blog</a>
12 >
13 </body>
```

Look at that. It works! Building a JavaScript frontend library is really that simple? Not quite but hopefully you got an hint of what happens when React or Vue build an HTML page for you. In the process you learned about recursion and DOM manipulation.

Conclusions

The Document Object Model is a virtual copy of the web page that the browser creates and keeps in memory. When creating, modifying, removing HTML elements we talk of “DOM Manipulation”. In the past we used to rely on jQuery even for simpler tasks but today the native API is compatible and mature enough to make jQuery obsolete.

jQuery won’t disappear soon but every JavaScript developer must know how to manipulate the DOM with the native API. There are many reasons to do so. Additional libraries increase load time and size of JavaScript applications. Not to mention that DOM manipulation come up a lot in technical interviews.

Every HTML element available in the DOM has an interface exposing a certain number of properties and methods. When in doubt about what method to use head over the excellent docs at [MDN](https://developer.mozilla.org/en-US/)²⁷. Most common methods for manipulating the DOM are `document.createElement()` for creating a new HTML element, `document.createTextNode()` for creating text nodes inside the DOM. Last but not least there is `.appendChild()` for appending your new HTML element or a text node to an existing element.

While it is nice to have a good knowledge of the native API these days modern front-end libraries offer unquestionable benefits. It is indeed feasible to build a large JavaScript application with “vanilla” JavaScript. But sometimes Angular, React, Vue can help a lot. Sticking with just JavaScript for simpler prototypes and small to medium apps can be a wise move too.

²⁷<https://developer.mozilla.org/en-US/>



Sharpen up your JavaScript skills

- What is the difference between window and DOM?
- Name at least 3 useful methods for manipulating tables with JavaScript
- Given a JavaScript object how can I extract every key from it?

Chapter 10. Working with asynchronous JavaScript

REST API and XMLHttpRequest

If you ask me, JavaScript is really useful. As a scripting language running in a browser it does all sort of things like creating elements dynamically, adding interactivity, and so on. In chapter 8 you built an HTML table starting from an array. An array is a synchronous data source, that is, it's available straight away in our code. No need to wait. But as JavaScript developers most of the time we want to fetch data from the outside: from a REST API for example. As opposed to synchronous data sources, network communication is always an asynchronous operation: we request the data and the server “might” respond back later. But JavaScript has no built-in asynchronicity: most of the times it's the browser that provides external helpers for dealing with time-consuming operations. In chapter 3 you also saw `setTimeout` and `setInterval`, two browser APIs for time based work. Browsers offers a lot of API and there's one called `XMLHttpRequest` aiming specifically at making network requests. The naming is really weird: `XMLHttpRequest` sounds so '00. In fact it comes from an era when most web services exposed data in XML format. Nowadays JSON is the most popular communication “protocol” for moving data between web services but the name `XMLHttpRequest` eventually stuck. `XMLHttpRequest` is also part of a set of technologies called AJAX which stands for “Asynchronous JavaScript and XML”. AJAX was born for making network requests in the browser as slick as possible. AJAX meant the ability to fetch data from a remote source without causing a page refresh. At that time the idea was almost revolutionary. With the introduction of `XMLHttpRequest` (almost 13 years ago) the platform gained a native method for making network requests. Given a url we can make an HTTP request like so:

```
1 var request = new XMLHttpRequest();
2
3 request.open("GET", "https://academy.valentinog.com/api/link/");
4
5 request.addEventListener("load", function() {
6     console.log(this.response);
7 });
8
9 request.send();
```

The act of making an XHR (`XMLHttpRequest`) goes like that:

- create a new `XMLHttpRequest` object

- open the request by providing a method and an url
- register event listeners
- send the request

XMLHttpRequest is based on DOM events (seen on chapter 8) thus we can use either `addEventListener` or `onload` for listening to the “load” event which is triggered whenever the request succeeds. Here’s the alternative syntax:

```
1  var request = new XMLHttpRequest();
2
3  request.open("GET", "https://academy.valentinog.com/api/link/");
4
5  request.onload = function() {
6    console.log(this.response);
7  };
8
9  request.send();
```

For failing requests (network errors) we can register a listener on the “error” event:

```
1  var request = new XMLHttpRequest();
2
3  request.open("GET", "https://academy.valentinog.com/api/link/");
4
5  request.onload = function() {
6    console.log(this.response);
7  };
8
9  request.onerror = function() {
10   // handle the error
11 };
12
13 request.send();
```

Now armed with this knowledge let’s put `XMLHttpRequest` to good use.

XMLHttpRequest in action: generating lists with HTML and JavaScript

In the following exercise you’ll build a simple HTML list but not before fetching the data from a REST API. Create a new file named **build-list.html** in a folder of choice:

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <title>XMLHttpRequest</title>
6 </head>
7 <body>
8
9 </body>
10 <script src="xhr.js"></script>
11 </html>
```

Next up create a file named `xhr.js` in the same folder. In this file we're going to create a new XHR request (XHR stands for asynchronous JavaScript and XML):

```
1 "use strict";
2
3 const request = new XMLHttpRequest();
```



Still “new” in JavaScript

At this point should be clear that `new` in JavaScript is a constructor call returning a new object of the desired type. Check out chapter 5 if you need a refresh.

The above call (called constructor call) creates a new object of type `XMLHttpRequest` which can be used for making HTTP requests. Opposed to asynchronous functions like `setTimeout`, expecting the callback as an argument:

```
1 setTimeout(callback, 10000);
2
3 function callback() {
4   console.log("hello timer!");
5 }
```

`XMLHttpRequest` is based on DOM events and the handler callback is registered on the `onload` object. The load event fires when the request succeeds:

```
1  "use strict";
2
3  const request = new XMLHttpRequest();
4
5  request.onload = callback;
6
7  function callback() {
8    console.log("Got the response!");
9  }
```

After registering the callback we can open the request with `open()`. It accepts an HTTP method (GET in our case) and an url to contact:

```
1  "use strict";
2
3  const request = new XMLHttpRequest();
4
5  request.onload = callback;
6
7  function callback() {
8    console.log("Got the response!");
9  }
10
11 request.open("GET", "https://academy.valentinog.com/api/link/");
```

Finally we can send the actual request with `send()`:

```
1  "use strict";
2
3  const request = new XMLHttpRequest();
4
5  request.onload = callback;
6
7  function callback() {
8    console.log("Got the response!");
9  }
10
11 request.open("GET", "https://academy.valentinog.com/api/link/");
12 request.send();
```

If you did everything right you'll see "Got the response!" in the browser console. Anyway, printing a string is almost useless. Instead we're interested in the response from the server. If you remember

from chapter 6, every regular JavaScript function holds a reference to the object in which is running: `this`. Since the callback is running inside the `XMLHttpRequest` we can access the server response on `this.response`

```
1  "use strict";
2
3  const request = new XMLHttpRequest();
4
5  request.onload = callback;
6
7  function callback() {
8    // this refers to the new XMLHttpRequest
9    // response is the server's response
10   console.log(this.response);
11 }
12
13 request.open("GET", "https://academy.valentinog.com/api/link/");
14 request.send();
```

Save the file and visit **build-list.html**. You should see the server's response in the console, as a text document. It's still not so useful because we're interested in a JSON document rather than on its textual representation. At this point we have two options:

- option 1: configuring the response type on the `XMLHttpRequest` object
- option 2: using `JSON.parse()`

The first option is more hacky, I don't like strings hanging around:

```
1  "use strict";
2
3  const request = new XMLHttpRequest();
4
5  request.onload = callback;
6
7  function callback() {
8    // this refers to the new XMLHttpRequest
9    // response is the server's response
10   console.log(this.response);
11 }
12
13 // configure the response type
14 request.responseType = "json";
15 request.open("GET", "https://academy.valentinog.com/api/link/");
16 request.send();
```


Option 2 (JSON.parse) is more descriptive and idiomatic:

```
1  "use strict";
2
3  const request = new XMLHttpRequest();
4
5  request.onload = callback;
6
7  function callback() {
8    const response = JSON.parse(this.response);
9    console.log(response);
10 }
11
12 request.open("GET", "https://academy.valentinog.com/api/link/");
13 request.send();
```

Now we're ready to use the JSON response for building a list of elements.

COMING SOON

Asynchronous evolution: from XMLHttpRequest to Fetch

TODO

Error handling with Fetch

TODO

Fetch with async/await

TODO

Rebuilding the Fetch API from scratch (for fun and profit)

Often overlooked in favor of libraries like Axios, the Fetch API is a native browser method. Fetch was born in 2015, the same year which saw the introduction of ECMAScript 2015 and Promise. But AJAX, a set of technologies for fetching data in the browser exists since 1999. Nowadays we take AJAX and Fetch for granted but few know that Fetch is nothing more than a glorified version of XMLHttpRequest. Fetch is similar but looks more concise, and more important is based on Promises:

```
1 fetch("https://academy.valentinog.com/api/link/").then(function(response) {  
2   console.log(response);  
3 });
```

In the following exercise you'll build Fetch from scratch, making use of Promises for wrapping XMLHttpRequest. As a JavaScript developer most of the time you will “consume” a Promise rather than create one from scratch. That means most developers know about Promises because they've used then and catch as consumers of other people code. But that's only half of the story. Library creators on the other hand use Promise for wrapping legacy code and offering a more modern API to developers. And that's exactly what we're going to do: we'll create a so called polyfill for Fetch. To start off create a new file named **fetch.html** and save it in a folder of choice:

```
1 <!DOCTYPE html>  
2 <html lang="en">  
3 <head>  
4   <meta charset="UTF-8">  
5   <title>Building Fetch from scratch</title>  
6 </head>  
7 <body>  
8  
9 </body>  
10 <script src="fetch.js"></script>  
11 </html>
```



What's a polyfill?

“Polyfill” is an hand-made version of a functionality that is not yet implemented in all browsers. In other words is custom code that any JavaScript developer can write and distribute, code that enables a new functionality in the language (or in browsers). For example before the real Fetch API was released to the public, developers eager to try it installed a Fetch polyfill for enabling the feature. All of that before the “real” Fetch was implemented by browser's vendors.

Then create another file named **fetch.js** in the same folder and start the code with the following statements:

```
1 "use strict";  
2  
3 window.fetch = null;
```

In the first line we ensure we're in strict mode and in the second line we “nullify” the original Fetch API. With everything in place we can start building our own Fetch API. But first let's reverse engineer the real Fetch. The way Fetch works is quite simple. It takes an url and makes a GET request against it:

```
1 fetch("https://academy.valentinog.com/api/link/").then(function(response) {
2   console.log(response);
3 });
```

When a function is “chainable” with `then` it means it’s returning a Promise. So inside `fetch.js` let’s create a function named `fetch`, which takes an url and returns a new Promise. For creating a Promise you can call the Promise constructor and pass in a callback taking `resolve` and `reject`:

```
1 function fetch(url) {
2   return new Promise(function(resolve, reject) {
3     // do stuff
4   });
5 }
```

We know that when a function returns a resolved Promise the `then` handler is triggered. Let’s give it a shot by resolving our Promise:

```
1 function fetch(url) {
2   return new Promise(function(resolve, reject) {
3     resolve("Fake response!");
4   });
5 }
```

Try the complete code. Now we assign our fake `fetch` to `window.fetch`:

```
1 "use strict";
2
3 window.fetch = fetch;
4
5 function fetch(url) {
6   return new Promise(function(resolve, reject) {
7     resolve("Fake response!");
8   });
9 }
10
11 fetch("https://academy.valentinog.com/api/link/").then(function(response) {
12   console.log(response);
13 });
```

You should get “Fake response!” inside the console. It works! Granted, it’s still a useless Fetch because nothing gets returned from the API. Let’s implement the real behaviour, with some help from XMLHttpRequest. We saw how to create a request with XMLHttpRequest:

```
1 var request = new XMLHttpRequest();
2 request.open("GET", "https://academy.valentinog.com/api/link/");
3 request.send();
4 request.onload = function() {
5   console.log(this.response);
6 };
```

for maximum compatibility we're going to use the legacy `onload` for listening to the load event. Let's put the knowledge to good use. What we're going to do now is wrapping `XMLHttpRequest` inside our `Promise`. Open up `fetch.js` and extend the `fetch` function to include a new `XMLHttpRequest`:

```
1 function fetch(url) {
2   return new Promise(function(resolve, reject) {
3     var request = new XMLHttpRequest();
4     request.open("GET", url);
5     request.onload = function() {
6       // handle the response
7     };
8     request.send();
9   });
10 }
```

Things start to make sense now, isn't it? We can also register an handler for failures on `onerror`:

```
1 function fetch(url) {
2   return new Promise(function(resolve, reject) {
3     var request = new XMLHttpRequest();
4     request.open("GET", url);
5     request.onload = function() {
6       // handle the response
7     };
8     request.onerror = function() {
9       // handle the error
10    };
11    request.send();
12  });
13 }
```

At this point it's a matter of resolving or rejecting the `Promise` depending on the request's fate. That means resolving the `Promise` inside the "load" handler with the response we've got from the server:

```
1 function fetch(url) {
2   return new Promise(function(resolve, reject) {
3     var request = new XMLHttpRequest();
4     request.open("GET", url);
5     request.onload = function() {
6       resolve(this.response);
7     };
8     request.onerror = function() {
9       // handle the error
10    };
11    request.send();
12  });
13 }
```

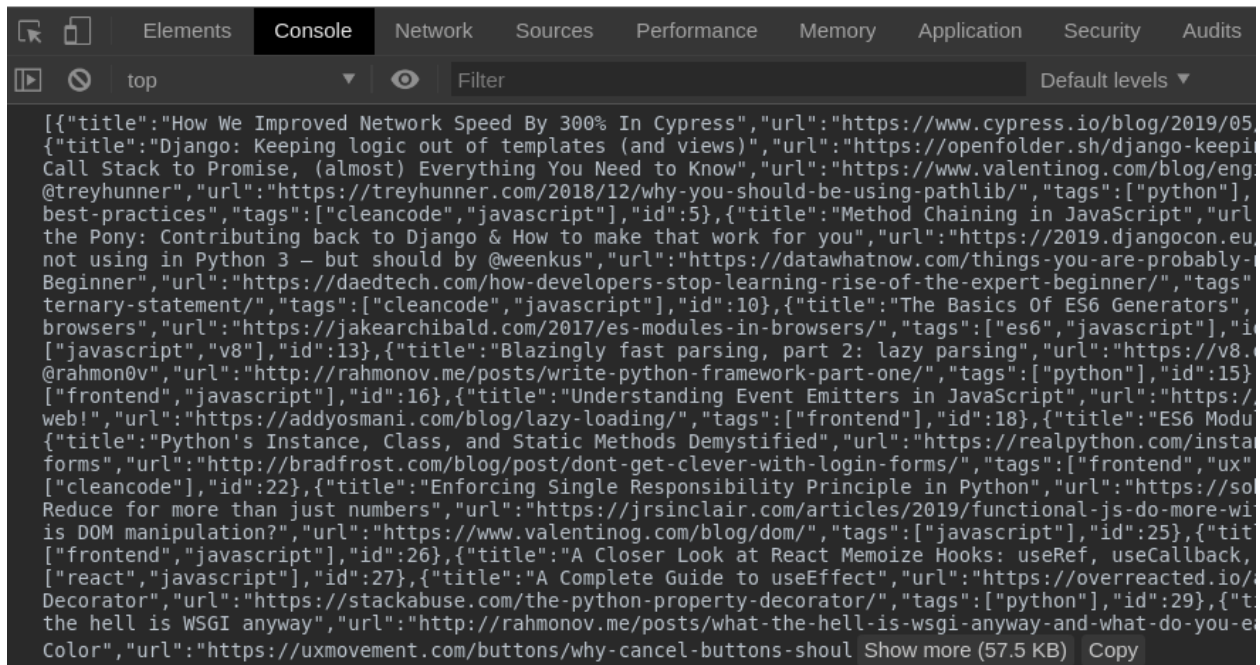
In case of error we reject the Promise inside the “error” handler:

```
1 function fetch(url) {
2   return new Promise(function(resolve, reject) {
3     var request = new XMLHttpRequest();
4     request.open("GET", url);
5     request.onload = function() {
6       resolve(this.response);
7     };
8     request.onerror = function() {
9       reject("Network error!");
10    };
11    request.send();
12  });
13 }
```

Rejected Promises are handled by the catch handler, that means you’ll be able to use your “fake” Fetch like so:

```
1 fetch("https://acdemy.valentinog.com/api/link/")
2   .then(function(response) {
3     console.log(response);
4   })
5   .catch(function(error) {
6     console.log(error);
7   });
```

Now in case of an incorrect url our Fetch will print “Network error!” to the console. If the url is correct you’ll get the response from the server:



The textual response from the server

Fantastic! Our Fetch polyfill works but there are some rough edges. First of all we need to implement a function for returning JSON. Let's see how. The actual Fetch API yields a response which can be later converted to JSON, blob or text, like so (for the scope of this exercise we'll implement only the JSON function):

```

1 fetch("https://academy.valentinog.com/api/link/")
2   .then(function(response) {
3     return response.json();
4   })
5   .then(function(json) {
6     console.log(json);
7   })

```

It should be easy to implement the functionality. Seems that “response” might be an entity on its own with a `json()` function attached. The JavaScript prototype system is a good candidate for structuring the code (check out chapter 5 for a refresh). Let's create a constructor function named `Response` and a method bound to its prototype (in `fetch.js`):

```
1  function Response(response) {
2    this.response = response;
3  }
4
5  Response.prototype.json = function() {
6    return JSON.parse(this.response);
7  };
```

That's all! Now we can use Response inside Fetch:

```
1  function fetch(url) {
2    return new Promise(function(resolve, reject) {
3      var request = new XMLHttpRequest();
4      request.open("GET", url);
5      request.onload = function() {
6        // Before:
7        // resolve(this.response);
8        // After:
9        var response = new Response(this.response);
10       resolve(response);
11     };
12     request.onerror = function() {
13       reject("Network error!");
14     };
15     request.send();
16   });
17 }
```

Here's the complete code so far:

```
1  window.fetch = fetch;
2
3  function Response(response) {
4    this.response = response;
5  }
6
7  Response.prototype.json = function() {
8    return JSON.parse(this.response);
9  };
10
11  function fetch(url) {
12    return new Promise(function(resolve, reject) {
```

```
13     var request = new XMLHttpRequest();
14     request.open("GET", url);
15     request.onload = function() {
16         // Before:
17         // resolve(this.response);
18         // After:
19         var response = new Response(this.response);
20         resolve(response);
21     };
22     request.onerror = function() {
23         reject("Network error!");
24     };
25     request.send();
26 });
27 }
28
29 fetch("https://academy.valentinog.com/api/link/")
30 .then(function(response) {
31     return response.json();
32 })
33 .then(function(json) {
34     console.log(json);
35 })
36 .catch(function(error) {
37     console.log(error);
38 });
```

If you did everything correct the above code should print an array of objects into the browser's console. Good job! And now let's take care of errors. The real version of Fetch is much more complicated than our polyfill but it's not so difficult to replicate the same exact behaviour. The Response object in Fetch has a property, a boolean called "ok". That boolean is set to true when the request succeeds and to false when the request fails. A developer can check the boolean and alt the Promise handler by throwing an error. Here's how you would check the status with the real Fetch:


```
1 fetch("https://academy.valentinog.com/api/link/")
2   .then(function(response) {
3     if (!response.ok) {
4       throw Error(response.statusText);
5     }
6     return response.json();
7   })
8   .then(function(json) {
9     console.log(json);
10  })
11  .catch(function(error) {
12    console.log(error);
13  });
```

As you can see there's also a "statusText". Seems easy to implement both "ok" and "statusText" in the Response object. In particular response.ok should be true when the server replies:

- with a status code equal or less than 200
- with a status code less than 300

Open up **fetch.js** and tweak the Response object to include the new properties. Also make sure to change the first line to read the response from response.response:

```
1 function Response(response) {
2   this.response = response.response;
3   this.ok = response.status >= 200 && response.status < 300;
4   this.statusText = response.statusText;
5 }
```

And guess what there's no need to make up "statusText" because it's already returning from the original XMLHttpRequest response object. That means we only need to pass the entire response when constructing our custom Response:

```
1  function fetch(url) {
2    return new Promise(function(resolve, reject) {
3      var request = new XMLHttpRequest();
4      request.open("GET", url);
5      request.onload = function() {
6        // Before:
7        // var response = new Response(this.response);
8        // After: pass the entire response
9        var response = new Response(this);
10       resolve(response);
11     };
12     request.onerror = function() {
13       reject("Network error!");
14     };
15     request.send();
16   });
17 }
```

Here's the complete code again:

```
1  "use strict";
2
3  window.fetch = fetch;
4
5  function Response(response) {
6    this.response = response.response;
7    this.ok = response.status >= 200 && response.status < 300;
8    this.statusText = response.statusText;
9  }
10
11 Response.prototype.json = function() {
12   return JSON.parse(this.response);
13 };
14
15 function fetch(url) {
16   return new Promise(function(resolve, reject) {
17     var request = new XMLHttpRequest();
18     request.open("GET", url);
19     request.onload = function() {
20       // Before:
21       // var response = new Response(this.response);
22       // After: pass the entire response
23       var response = new Response(this);
```

```
24     resolve(response);
25   };
26   request.onerror = function() {
27     reject("Network error!");
28   };
29   request.send();
30 });
31 }
32
33 fetch("https://academy.valentinog.com/api/link/")
34   .then(function(response) {
35     if (!response.ok) {
36       throw Error(response.statusText);
37     }
38     return response.json();
39   })
40   .then(function(json) {
41     console.log(json);
42   })
43   .catch(function(error) {
44     console.log(error);
45   });
```

As of now our Fetch polyfill is able to handle errors and success. You can check the error handling by providing a failing url like so:

```
1  fetch("https://httpstat.us/400")
2    .then(function(response) {
3      if (!response.ok) {
4        throw Error(response.statusText);
5      }
6      return response.json();
7    })
8    .then(function(json) {
9      console.log(json);
10   })
11   .catch(function(error) {
12     console.log(error);
13   });
```

You should see “Error: bad request” in the browser’s console. Way to go! And now let’s extend our polyfill for making POST requests. First of all let’s see how to make POST requests with the real Fetch. The entity we’re going to create on the server is a “link”, that is, made of a title and an url:

```
1  var link = {
2    title: "Building a Fetch Polyfill From Scratch",
3    url: "https://www.valentinog.com/fetch-api/"
4  };
```

Next up we configure the request for Fetch. It's done in an object which should contain at least the HTTP method, the request's content type (JSON in our case) and the request's body (the new link):

```
1  var link = {
2    title: "Building a Fetch Polyfill From Scratch",
3    url: "https://www.valentinog.com/fetch-api/"
4  };
5
6  var requestInit = {
7    method: "POST",
8    headers: {
9      "Content-Type": "application/json"
10   },
11   body: JSON.stringify(link)
12 };
```

Finally we make the request like so:

```
1  var link = {
2    title: "Building a Fetch Polyfill From Scratch",
3    url: "https://www.valentinog.com/fetch-api/"
4  };
5
6  var requestInit = {
7    method: "POST",
8    headers: {
9      "Content-Type": "application/json"
10   },
11   body: JSON.stringify(link)
12 };
13
14 fetch("https://academy.valentinog.com/api/link/create/", requestInit)
15   .then(function(response) {
16     if (!response.ok) {
17       throw Error(response.statusText);
18     }
19     return response.json();
```

```
20  })
21  .then(function(json) {
22    console.log(json);
23  })
24  .catch(function(error) {
25    console.log(error);
26  });
```

But now we have a problem with our polyfill. It accepts a single parameter “url” and makes only GET requests against it:

```
1  function fetch(url) {
2    return new Promise(function(resolve, reject) {
3      var request = new XMLHttpRequest();
4      request.open("GET", url);
5      request.onload = function() {
6        var response = new Response(this);
7        resolve(response);
8      };
9      request.onerror = function() {
10       reject("Network error!");
11     };
12     request.send();
13   });
14 }
```

Fixing the issue should be easy. First we can accept a second parameter named `requestInit`. Then based on that parameter we can construct a new `Request` object that:

- makes GET requests by default
- uses body, method, and headers from `requestInit` if the latter is provided

To start with we make a new “Request” function with some properties named `body`, `method`, and `headers`. Open up `fetch.js` and get coding:

```
1  function Request(requestInit) {
2    this.method = requestInit.method || "GET";
3    this.body = requestInit.body;
4    this.headers = requestInit.headers;
5  }
```

Next up we’re going to use the new function inside our polyfill. We prepare a request configuration object just before opening the request:

```
1 function fetch(url, requestInit) {
2   return new Promise(function(resolve, reject) {
3     var request = new XMLHttpRequest();
4     var requestConfiguration = new Request(requestInit || {});
5     // more soon
6   });
7 }
```

Now the request will use the method defined in the Request object:

```
1 function fetch(url, requestInit) {
2   return new Promise(function(resolve, reject) {
3     var request = new XMLHttpRequest();
4     var requestConfiguration = new Request(requestInit || {});
5     request.open(requestConfiguration.method, url);
6     // more soon
7   });
8 }
```

The logic for handling errors and success (load) remains the same:

```
1 function fetch(url, requestInit) {
2   return new Promise(function(resolve, reject) {
3     var request = new XMLHttpRequest();
4     var requestConfiguration = new Request(requestInit || {});
5     request.open(requestConfiguration.method, url);
6     request.onload = function() {
7       var response = new Response(this);
8       resolve(response);
9     };
10    request.onerror = function() {
11      reject("Network error!");
12    };
13    // more soon
14  });
15 }
```

But on top of that we can add a minimal logic for setting up request headers:

```
1 function fetch(url, requestInit) {
2   return new Promise(function(resolve, reject) {
3     var request = new XMLHttpRequest();
4     var requestConfiguration = new Request(requestInit || {});
5     request.open(requestConfiguration.method, url);
6     request.onload = function() {
7       var response = new Response(this);
8       resolve(response);
9     };
10    request.onerror = function() {
11      reject("Network error!");
12    };
13    // Set headers on the request
14    for (var header in requestConfiguration.headers) {
15      request.setRequestHeader(header, requestConfiguration.headers[header]);
16    }
17    // more soon
18  });
19 }
```

Headers are important for configuring the AJAX request. Most of the times you may want to set application/json or an authentication token in the headers. As the cherry on top we can complete our code to send:

- an empty request if there is no body
- a request with some payload if the body is provided:

```
1 function fetch(url, requestInit) {
2   return new Promise(function(resolve, reject) {
3     var request = new XMLHttpRequest();
4     var requestConfiguration = new Request(requestInit || {});
5     request.open(requestConfiguration.method, url);
6     request.onload = function() {
7       var response = new Response(this);
8       resolve(response);
9     };
10    request.onerror = function() {
11      reject("Network error!");
12    };
13    // Set headers on the request
14    for (var header in requestConfiguration.headers) {
15      request.setRequestHeader(header, requestConfiguration.headers[header]);
```

```

16     }
17     // If there's a body send it
18     // If not send an empty GET request
19     requestConfiguration.body && request.send(requestConfiguration.body);
20     requestConfiguration.body || request.send();
21 });
22 }

```

That should cover the POST request feature. Here's the complete code for our Fetch polyfill so far:

```

1  "use strict";
2
3  window.fetch = fetch;
4
5  function Response(response) {
6      this.response = response.response;
7      this.ok = response.status >= 200 && response.status < 300;
8      this.statusText = response.statusText;
9  }
10
11 Response.prototype.json = function() {
12     return JSON.parse(this.response);
13 };
14
15 function Request(requestInit) {
16     this.method = requestInit.method || "GET";
17     this.body = requestInit.body;
18     this.headers = requestInit.headers;
19 }
20
21 function fetch(url, requestInit) {
22     return new Promise(function(resolve, reject) {
23         var request = new XMLHttpRequest();
24         var requestConfiguration = new Request(requestInit || {});
25         request.open(requestConfiguration.method, url);
26         request.onload = function() {
27             var response = new Response(this);
28             resolve(response);
29         };
30         request.onerror = function() {
31             reject("Network error!");
32         };
33         for (var header in requestConfiguration.headers) {

```



```
34     request.setRequestHeader(header, requestConfiguration.headers[header]);
35   }
36   requestConfiguration.body && request.send(requestConfiguration.body);
37   requestConfiguration.body || request.send();
38 });
39 }
40
41 var link = {
42   title: "Building a Fetch Polyfill From Scratch",
43   url: "https://www.valentinog.com/fetch-api/"
44 };
45
46 var requestInit = {
47   method: "POST",
48   headers: {
49     "Content-Type": "application/json"
50   },
51   body: JSON.stringify(link)
52 };
53
54 fetch("https://academy.valentinog.com/api/link/create/", requestInit)
55   .then(function(response) {
56     if (!response.ok) {
57       throw Error(response.statusText);
58     }
59     return response.json();
60   })
61   .then(function(json) {
62     console.log(json);
63   })
64   .catch(function(error) {
65     console.log(error);
66   });
```

Give it a shot and maybe write a test for it. The real Fetch API implementation is much more complex and has support for more advanced features. We just scratched the surface of Fetch in this exercises but as you can see it's coming up quite well. Of course the code can be improved, for example the logic for adding headers can live on a method on it's own. Plus there's a lot of room for adding new features: support for PUT and DELETE and more functions for returning the response in different formats. I leave that as an exercise for you. If you're curious to see a more complex Fetch API polyfill check out [whatwg-fetch](https://github.com/whatwg-fetch)²⁸ from the engineers at Github. You'll see a lot of similarities with your handcrafted polyfill!

²⁸<https://github.com/whatwg-fetch>

Conclusions

TODO



Sharpen up your JavaScript skills

- What is a polyfill?
- TODO
- TODO