COMPLETE GUIDE TO MODERN JAVASCRIPT

2nd Edition

Everything from the Basics to the 2019 Version

ALBERTO MONTALESI

InspiredWebDev.com

2nd Edition

Introduction

This book was first published in 2018 as a result of months of self-study of `JavaScript`, and it was aimed to `JavaScript` developers who wanted to update their skills to the newest version of the `ECMAScript` specification.

I was really proud of the result, but at the same time, I felt this book was missing something. It was not suitable for newcomers to the language.

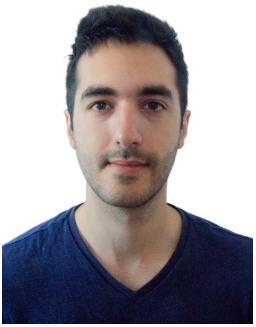
This 2019 edition addresses the problems of the first one and expands on it, targeting both `JavaScript` developers and complete beginners.

That is why I have added a new section entirely dedicated to introducing `JavaScript` to beginners.

Another addition is the `TypeScript` section. After a year of working with `TypeScript`, I can comfortably say that it is a must-know for any `JavaScript` developer and I think that even beginners should try to adopt it in their workflow.

A free version of this book can be found on Github at https://github.com/AlbertoMontalesi/The-complete-guide-to-modern-JavaScript
where you can find the core chapters of the book that will be updated with each new version of `ECMAScript`.

About Me



My name is Alberto Montalesi, I am from Italy and I am working in Vietnam as a Software Developer creating enterprise software.

My passion for programming started late in life, in 2016, at the age of 24 after a bachelor's degree in Law.

My path to becoming a self-taught software developer has not been easy, but it's definitely something I would do again.

You can read my story on DevTo at this link: https://dev.to/albertomontalesi/my-journey-from-esl-teacher-to-software-developer-5h30.

Writing a book that can help other aspiring developers fills me with pride as I know very well how hard it can be to find the motivation and the resources to continue studying and improving your skill.

Apart from programming, my other passions include photography, traveling, and gaming.

Get in touch

If you want to get in touch for any type of collaboration or discussion you can find me on:

- Twitter
- DevTo at https://dev.to/albertomontalesi
- Github at https://github.com/AlbertoMontalesi
- InspiredWebDev my personal blog https://inspiredwebdev.com

Contributions & Donations

Any contributions you make are of course greatly appreciated.

If you enjoy my content and you want to donate a cup of coffee to me, you can do so at https://www.paypal.me/albertomontalesi.

License

Set up your environment

If you already have a code editor installed on your computer and you already know how to use the `chrome developer tools` or similar tools, then you can skip this chapter and move on.

In order for you to play around with the code we are going to use in this book I suggest you download a code editor that you will use to write and run the `JavaScript` code we will talk about.

My personal choice is Visual Studio Code, made by Microsoft and available for free here: https://code.visualstudio.com/

Other alternatives are:

- Atom: https://atom.io/
- Sublime Text: https://www.sublimetext.com/

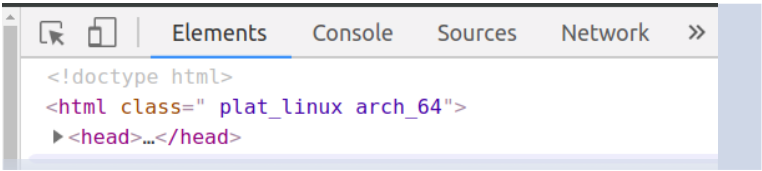
Whatever is your choice, they will all be enough for your needs.

After installing it you just have to open the software and create a new file and save it as `.js` and you will have your first `JavaScript` file.

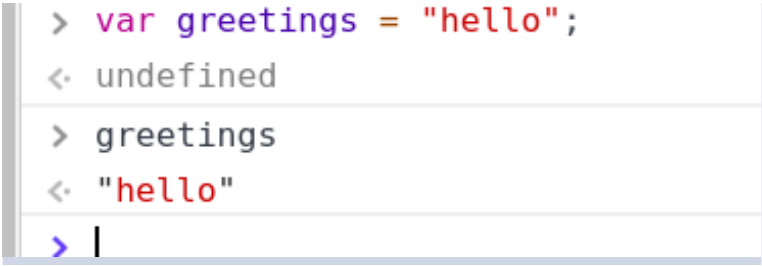
The second thing we will look at are the `developer tools`.

Open your browser to any page and right click somewhere. You will see a set of options, one of which being `inspect` or `inspect element`. If you click it it will open the `developer tools` that you can use to look at web pages code, play around with `JavaScript` or debug your application.

This is how the Chrome Developer Tools will look like.



The first tab `Elements` will let you look at the code of the page you are inspecting, the second tab `Console` is where you can write your `JavaScript` and experiment.



What I did here was defining a variable and calling it. You will learn about variables and much more in the next chapter where I will introduce you to the basics of `JavaScript`.

JavaScript Basics

JavaScript is a programming language created by Brendan Eich in 1995 that enables interactive web pages and is an essential part of web applications.

If you want to learn more about the history of the language and its name, I suggest that you read this brief article on Medium.com https://medium.com/@benastontweet/lesson-1a-the-history-of-javascript-8c1ce3bffb17 .

If you have ever opened the Chrome developer tools or a similar tool you may have already seen how `JavaScript` is inserted into an HTML page.

We do that by using the `script` tag and either inserting our `JavaScript` code directly inside of it or by referencing an external file.

Code inside of the script tag:

```
<script type="text/javascript">
[YOUR_SCRIPT_HERE] </script>
```

Reference an external file:

```
<script src="/home/script.js"></script>
```

Of course you can add as many scripts as you want and also use both relative, absolute and full path such as:

```
 <!-- absolute path from the root of our
project -->
<script src="/home/script.js"></script>
<!-- relative path to our current folder --
>
<script src="script.js"></script>
<!-- full url to the jquery library -->
<script src="https://cdnjs.cloudflare.com/
ajax/libs/jquery/3.3.1/core.js"></script>
```

It's better to not write your code inside of the script tag but instead put it in its own file so that it can be cached by the browser and downloaded only once, regardless of how many files import it. Those files will use the cached version of the file, improving performance.

Variables

We use variables to store values, which can be anything from a username, an address or an item from our e-commerce site, for example.

Prior to ES6 (ES2015) the way we would declare a variable was:

```
var username = "Alberto Montalesi"
```

Now we have 2 more options when it comes to declaring variables:

```
let username = "Alberto Montalesi"
const username = "Alberto Montalesi"
```

We will go deeper into the differences between these three keywords in the Chapter 1 but let me give you a quick explanation.

Variables created with the keyword `const` are, as the name implies, constant, meaning that they *cannot* be overwritten.

Open your Chrome Developer Tools and try typing the following:

```
const age = 26;
age = 27;
// Uncaught TypeError: Assignment to
constant variable
```

As you can see we are not allowed to assign a new value to our constant.

On the other hand if we were to do this:

```
let height  = 190;
height = 189
```

We get no complaint this time, `let` can be reassigned, similarly to the old `var` keyboard.

If both `var` and `let` can be reassigned, then why should we use `let` instead of `var`? The answer for that requires a bit more explanation of how `JavaScript` works and it will be discussed later in the Chapter 1.

Many people argue what the best practice is when it comes to `let` and `const`. Here is my take: se `const` all the time unless you know in advance that you will need to reassign its value.

If later on you need to reassign one of your `const`, simply make it a `let` and it will be enough. I find that for myself, it's better to make them `const` by default. Then I see errors if I accidentally try to reassign them rather than having to debug the code later just to find out that I was referencing the wrong variable.

A note about naming variables

There are certain rules to respect when it comes to naming variables. But don't worry, most of them are very easy to remember.

The following are all forbidden:

```
// variables name cannot start with a
number
let 1apple = "one apple";
// variables name cannot include any
character such as spaces, symbols and
```

```
punctuation marks
let hello! = "hello!";
```

There are also certain words that are reserved and cannot be used as names for variables and functions.

abstract	arguments	await	boolean
break	byte	case	catch
char	class	const	continue
debugger	default	delete	do
double	else	enum	eval
export	extends	false	final
finally	float	for	function
goto	if	implements	import
in	instanceof	int	interface
let	long	native	new
null	package	private	protected
public	return	short	static
super	switch	synchronized	this
throw	throws	transient	true
try	typeof	var	void
volatile	while	with	yield

The rule of thumb when it comes to choosing the name for your variable is to make them **descriptive**. Avoid using *acronyms, abbreviations and meaningless names*.

```
 // BAD
let cid = 12; // what is a `cid`
// GOOD
let clientID = 12; // oh, a `client id`

// BAD
let id = 12 // what id? userID? dogID?
catID?
// GOOD
let userID = 12 // be specific
```

If you want your variable names to be as descriptive as possible, chances are they are going to be multi-words.
In that case, the two most common ways of writing variable names are `camelCase` and `snake_case`.

```
 // BAD
let lastloggedin = '' // hard to read
// GOOD
let lastLoggedIn = '' // camelCase
let last_logged_in = '' // snake_case
```

Whether you choose to use `camelCase` and capitalize each word of the name after the first one, or you choose to use `snake_case` and put an underscore between each word, remember to be **consistent** and stick to your choice.

Data Types

JavaScript is a dynamic language, meaning that on the contrary to a static language, you **don't** have to define the type of your variables when you define them.

```
 // is this a string or a number ?
var userID;

userID = 12; // now it's a number
console.log(typeof userID); // number
userID = 'user1' // now it's a string
console.log(typeof userID); // string
```

This may seem convenient at first, but it can be a cause of problems when working on bigger projects. At the end of this book, after you have mastered the basics of `JavaScript` I will introduce you to `TypeScript`, which adds strong typing to `JavaScript`.

In `JavaScript`, there are 7 data types: 6 **primitives** and the `Object`.

Primitives

A **primitive** is a value is simply data that is not an `Object` and does not have methods.

They are:

- `string`
- `number`
- `boolean`
- `null`
- `undefined`
- `symbol` (the latest addition)

Let's have a quick look at all of them, some of which you may already know if you have prior experience in programming.

`string` is used to represent text data, whether it is a name, an address or a chapter of a book.

```
let userName = "Alberto";
console.log(userName) // Alberto
```

`number` is used to represent numerical values.
In `JavaScript` there is no specific type for Integers.

```
let age = 25;
```

`boolean` is used to represent a value that is either `true` or `false`.

```
let married = false;
```

`null` represents *absence* of value, while `undefined` represent an *undefined* value.

`symbol` represents a value that is unique and immutable. It was added in ES2015, making it the most recent addition to this list.
We will have a better look at it in Chapter 11.

Objects

While the previous 6 **primitives** that we discussed can hold only a single thing, whether it's a null value, true, false, etc., Objects are used to store the collection of properties

Let's first look at a simple `Object`

```
const car = {
  wheels: 4,
  color: "red",
}
```

This is a simple Object that i use to store properties of my car.

Each property has a key, in the case of the first line it's `wheels`, and a value, in this case *4*.

Key is of type `string` but the value can be of **any** type, they can also be functions and in that case we call them `methods`.

```
const car = {
  wheels: 4,
  color: "red",
  drive: function(){
    console.log("wroom wroom")
  }
}
console.log(Object.keys(car)[0]) // wheels
console.log(typeof Object.keys(car)[0]) //
string
car.drive();
// wroom wroom
```

As you can see now, we can call the function `drive` on the object `car`.

Don't worry, we will look at functions more in the next chapter.

Create an empty Object

We don't have to declare properties when we create an Object.

Here are two ways of creating an empty `Object`:

```
const car = new Object()
const car = {}
```

The more commonly used syntax is the second one, which is called `object literal`

Now that you have a new empty `car` object, to add new properties to it, you can simply do this:

```
car.color = 'red';
console.log(car)
// {color: "red"}
```

As you can see, we use the *dot notation* to add a new property to the `car` Object.

How about **accessing properties** on the Object?
It's very simple and we have two choices:

```
const car = {
  wheels: 4,
  color: "red",
}
```

```
console.log(car.wheels);
// 4
console.log(car['color']);
// 'red'
```

We have two different way of doing the same thing? Why?

Well, they are not completely the same.
In case of *multi-word* properties we cannot use the `dot notation`.

```
 const car = {
  wheels: 4,
  color: "red",
  "goes fast": true
}
console.log(car.goes fast);
// syntax error
console.log(car['goes fast'])
// true
```

When you want to use *multi-word* properties, you need to remember to wrap their name in quotation marks and you are able to access them only with *bracket notation*.

Another use for the `bracket notation` is to use it to access properties of an Object by its key.

Let's say that our application receives an input from a user, which is then saved into a variable that will be used to access our object.

The user is looking for cars and he/she has been asked to tell us the brand that he/she likes. That brand is a key that we will use to display back only the appropriate models.

For simplicity, in the example each brand will have only one model.

```
 const cars = {
   ferrari: "california",
   porsche: "911",
   bugatti: "veyron",
}

// user input
const key = "ferrari"
console.log(cars.key);
// undefined
console.log(cars['key']);
// undefined
console.log(cars[key]);
// california
```

As you can see, we need to use `bracket notation` to access the property of the Object via its key, stored in our variable.

Be careful, no strings are around `key` as it is a variable name and not a string.

Copying Objects

In contrast to **primitives**, objects are copied by reference, meaning that if we write:

```
let car = {
  color: 'red'
}
let secondCar = car;
```

Our `secondCar` will simply store a reference, an "address", to the `car` and not the `Object` itself.

It's easier to understand if you look at this:

```
let car = {
  color: 'red'
}
let secondCar = car;

car.wheels = 4
console.log(car);
```

```
// {color: 'red', wheels: 4}
console.log(secondCar);
// {color: 'red', wheels: 4}
```

As you can see, the `secondCar` simply stored a reference to `car`, therefore when we modified `car`, `secondCar` also changed.

If we compare the two objects, we can see something interesting:

```
 console.log(car == secondCar);
// true
console.log(car === secondCar);
// true
```

Whether we use **equality** (==) or **strict equality** (===) we get `true` meaning that the two objects are the same.

Only a comparison between the same `Object` will return `true`.

Look at this comparison between empty objects and objects with the same properties.

```
 const emptyObj1 = {};
const emptyObj2 = {};


emptyObj1 == emptyObj2;
```

```
// false
emptyObj1 === emptyObj2;
// false

const obj1 = {a: 1};
const obj2 = {a: 1};

obj1 == obj2;
// false
obj1 === obj2;
// false
```

As you can see, only a comparison between the **same object** returns `true`.

A quick way of making a clone of an `Object` in `JavaScript` is to use `Object.assign`.

```
 const car = {
   color:'red'
}

const secondCar = Object.assign({}, car)
car.wheels = 4;
console.log(car);
// {color: 'red', wheels: 4}
```

```
console.log(secondCar);
// {color: 'red'}
```

Updating `car` did not affect `secondCar`.
`Object.assign` takes a target object as the first argument, and a source as the second one.
In our example, we used an empty `Object` as our target and our `car` as the source.

If you are ready for a more in-depth look at copying Objects in JavaScript, I suggest you this article on Scotch.io at https://scotch.io/bar-talk/copying-objects-in-javascript

Arrays

As we have seen, Objects store data in a **key value** pair. Now we will have a look at what an `Array is.`

An `Array` is an `Object` that stores values in order.
In the example above we used an `Object` to store our car because it had specific properties that we wanted to be able to access easily via a key.
If we just want to store a list of items, then there is no need to create an `Object`. Instead, we can use an `Array`.

For example:

```
const fruitBasket =
['apple','banana','orange']
```

We access values of an array via their **index**.

Remember that arrays start at position 0.

```
const fruitBasket =
['apple','banana','orange']
console.log(fruitBasket[0])
// apple
console.log(fruitBasket[1])
// banana
console.log(fruitBasket[2])
// orange
```

There are many methods that we can call on an `Array`. Let's have a look at some of the most useful.

```
const fruitBasket =
['apple','banana','orange'];
// get the length of the Array
console.log(fruitBasket.length);
// 3

// add a new value at the end of the array
fruitBasket.push('pear')
console.log(fruitBasket);
// ["apple", "banana", "orange", "pear"]
```

```
// add a new value at the beginning of the
array
fruitBasket.unshift('melon')
console.log(fruitBasket);
// ["melon", "apple", "banana", "orange",
"pear"]

// remove a value from the end of the array
fruitBasket.pop()
console.log(fruitBasket);
// ["melon", "apple", "banana", "orange"]

// remove a value from the beginning of the
array
fruitBasket.shift()
console.log(fruitBasket);
// ["apple", "banana", "orange"]
```

As we can see we can easily add and remove elements from the beginning or the end of an `Array` with these methods.

You can find a longer list of methods on MDN at this link https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array .

Determining types using `typeof`

We can use `typeof` to determine the value of our variables. For example:

```
 const str = "hello"
typeof(str);
// string

const num = 12
typeof(num)
// number

const arr = [1,2,3]
typeof(arr)
// object

const obj = {prop: 'value'}
typeof(obj)
// object
```

Remember that 'Array' is not a type, Arrays are Objects!.

Everything seems pretty straight forward so far but what if we try something like this:

```
 typeof(null)
```

We know that `null` is a primitive, so should we expect to see `null` as the result?

```
typeof(null)
// object
```

Long story short, it's a bug from the first implementation of `JavaScript`. If you want to know more about it, this article offers a great explanation (http://2ality.com/2013/10/typeof-null.html)

Functions

Functions are a very important tool we use to perform tasks and calculations of all kinds.
In `JavaScript` we can declare a function in different ways.

Let's have a look at a **function definition**:

```
function greet(name){
  console.log("hello " + name);
}
greet("Alberto")
// hello Alberto
```

This is a simple function that when called will log a string.
The variable inside the parenthesis at line 1, is called a

parameter while the code inside of the curly brackets is a **statement**, in this case a very simple `console.log()`.

A very important thing to remember is that **primitives** are passed to a function **by value** meaning that the changes done to those values are not reflected globally. On the other hand if the value is not a primitive, such as an `Object` or an `Array` it is then passed **by reference** meaning that any modification done to it will be reflected in the original `Object`.

```
 let myInt = 1;


function increase(value){
  return value +=1;
}


console.log(myInt);
// 1
console.log(increase(myInt));
// 2
console.log(myInt);
// 1
```

As you can see, we increased the value of the integer but that did not affect the original variable.
Let's see an example with an `Object`.

```
let myCar = {
  make: "bmw",
  color: "red"
}

console.log(myCar)
// {make: "bmw", color: "red"}

function changeColor(car){
  car.color = "blue"
}

changeColor(myCar)
console.log(myCar)
// {make: "bmw", color: "blue"}
```

As you can see, since the parameter `car` was just a reference to the `Object` `myCar`, modifying it resulted in a change in the `myCar` `Object`.

Another way of declaring a function is by using a **function expression**.

```
const greeter = function greet(name){
  console.log("hello " + name);
}
```

```
greeter("Alberto")
// hello Alberto
```

Here we assigned our `function greet` to a `const`, called `greeter`.
We got the same result as in the first example but with a **function expression** we can also create **anonymous functions**.

```
 const greeter = function(name){
   console.log("hello " + name);
}
greeter("Alberto")
// hello Alberto
```

We changed `function greet` to `function` and we got an **anonymous function**.

We could also tweak it a little bit further by using an **arrow function**, introduced by ES2015.

```
 const greeter = (name) => {
   console.log("hello " + name);
}
greeter("Alberto")
// hello Alberto
```

The `function` keyword goes away and we get a nice fat arrow (=>) after the parameters.

We will look at arrow functions in detail in Chapter 2.

Understanding function scope and the this keyword

One of the most important concepts to understand in `JavaScript` is the scope.

What is Scope? \
The **scope** of a variable controls where that variable can be accessed from.
We can have a **global scope** meaning that the variable can be accessed from anywhere in our code or we can have a **block scope** meaning that the variable can be accessed only from inside of the block where it has been declared.

A `block` can be a function, a loop or anything delimited by *curly brackets*.

Let's have a look at two examples, first using the keyword `var`.

```
 var myInt = 1;


if(myInt === 1){
  var mySecondInt = 2
  console.log(mySecondInt);
  // 2
```

```
}
console.log(mySecondInt);
// 2
```

As you can see we were able to access the value of `mySecondInt` event outside of the `block` scope as variables declared with the keyword `var` are not bound to it.

Now let's use the keyword `let`.

```
 var myInt = 1;


if(myInt === 1){
  let mySecondInt = 2
  console.log(mySecondInt);
  // 2
}
console.log(mySecondInt);
// Uncaught ReferenceError: mySecondInt is
not defined
```

This time we couldn't access the variable from outside of the `block` scope and we got an error `mySecondInt is not defined`.

Variable declared with the keywords `let` or `const` are bound to the `block` scope where they have been declared.
You will learn more about them in Chapter 1.

The this keyword

The second important concept I want to discuss is the `this` keyword.

Let's first start with a simple example:

```
const myCar = {
  color: 'red',
  logColor: function(){
    console.log(this.color)
  }
}
myCar.logColor();
// red
```

As you can see in this example, it is self-explanatory that the `this` keyword referred to the `myCar` Object.

The value of `this` depends by how a function is called.
In the example above the function was called as a method of our `Object`.

Look at this other example:

```
function logThis(){
  console.log(this);
}
```

```
logThis();
// Window {...}
```

We called this function in the global context, therefore the value of `this` referred to the `Window Object`.

We can avoid accidentally referring to the `Window Object` by turning on `strict` mode.

You can do that by writing `'use strict';` at the beginning of your `JavaScript` file.

By doing so you will enable a stricter set of rules for `JavaScript`, among which there is one that sets the value of the `Global Object` to `undefined` instead of to the `Window Object` causing our `this` keyword to also become `undefined`.

If we want to manually set the value of `this` to something we can use `.bind`.

```
const myCar = {
  color: 'red',
  logColor: function(){
    console.log(this.color)
  }
}

const unboundGetColor = myCar.logColor;
```

```
console.log(unboundGetColor())
// undefined
const boundGetColor =
unboundGetColor.bind(myCar)
console.log(boundGetColor())
// red
```

Let's go through what we just did:

- First we created an `Object` similarly to the previous example
- We set `unboundGetColor` equal to the method of myCar
- When we try to call `unboundGetColor`, it tries to look for `this.color` but since it gets invoked in the global context, the value of `this` is the `Window Object` and there is no `color` on it, therefore we get `undefined`
- Lastly we use `.bind` to specifically tell `boundGetColor` that the `this` keyword will refer to the `Object` inside of the parenthesis, in this case `myCar`
- When we call `boundGetColor` you can see that this time we get the result that we were looking for

There are two other methods we can use to set the value of the `this` keyword: `.call()` and `.apply()`.

They are both similar in that both methods call a function with a given `this` value. The arguments they accept are a bit different.

`.call()` accepts a *list of arguments* while `.apply()` accepts a *single array of arguments*.

Look at this example using `.call()`

```
 function Car(make,color){
   this.carMake = make;
   this.carColor = color;
}

function MyCar(make,color){
   Car.call(this,make,color)
   this.age = 5
}
const myNewCar = new MyCar('bmw','red')
console.log(myNewCar.carMake)
// bmw
console.log(myNewCar.carColor)
// red
```

We are passing our `MyCar` Object inside of the `.call()` so that `this.carMake` will get set to the *make* that we passed as an argument of `MyCar`. Same for the *color*.

Look at this example with `.apply()` to see the differences between the two.

```
 function Car(make,color){
   this.carMake = make;
   this.carColor = color;
}

function MyCar(make,color){
   Car.apply(this,[make,color])
   this.age = 5
}
const myNewCar = new MyCar('bmw','red')
console.log(myNewCar.carMake)
// bmw
console.log(myNewCar.carColor)
// red
```

As you can see, the result was the same but in this case `.apply()` accepts an array with a list of arguments.

The major difference between the two comes in play when you are writing a function that does not need to know, or doesn't know the number of arguments required.
In that case, since `.call()` requires you to pass the arguments individually, it becomes problematic to do. The solution is to use `.apply()`, because you can just

pass the array and it will get unpacked inside of the function no matter how many arguments it contains.

```
 const ourFunction = function(item, method,
args){
  method.apply(args)
}
ourFunction(item,method, ['argument1',
'argument2'])
ourFunction(item,method, ['argument1',
'argument2', 'argument3'])
```

No matter how many arguments we pass, they will get *applied* individually when `.apply()` is called.

JavaScript Basics Quiz

JS-01 Which of the following ways of naming a variable is wrong?

- ☐ `var very_important = "very_important"`
- ☐ `var important_999 = "important_999"`
- ☐ `var important! = "important!"`
- ☐ `var VeRY_ImP_orTant = "VeRY_ImP_orTant"`

JS-02 Which one of the following is not a real Primitive?

- ☐ `symbol`
- ☐ `boolean`
- ☐ `null`
- ☐ `Object`

JS-03 What is the correct way of defining an `Object`?

- ☐ `const car: { color: "red"}`
- ☐ `const car = {color = "red"}`
- ☐ `const car = { color: "red" }`
- ☐ `const car: {color = "red"}`

JS-04 What is the correct output of the following code?

```
 const obj1 = {a: 1};
const obj2 = {a: 1};


console.log(obj1 === obj2);
```

- ☐ true
- ☐ undefined
- ☐ false
- ☐ null

JS-05 What is the correct output of the following code?

```
 const fruitBasket =
['apple','banana','orange'];
fruitBasket.unshift('melon')
console.log(fruitBasket);
```

- ☐ ["apple", "banana", "orange", "melon"]
- ☐ ["melon"]
- ☐ ["apple", "banana", "orange", "pear", "melon"]
- ☐ ["melon", "apple", "banana", "orange"]

Chapter 1: Var vs Let vs Const & the temporal dead zone

With the introduction of `let` and `const` in **ES6**, we can now better define our variables depending on our needs. During our `JavaScript` primer we looked at the basic differences between these 3 keywords, now we will go into more detail.

Var

Variables declared with the `var` keyword are **function scoped**, which means that if we declare them inside a `for` loop (which is a **block** scope) they will be available even outside of it.

```
 for (var i = 0; i < 10; i++) {
  var leak = "I am available outside of the
loop";
}

console.log(leak);
// I am available outside of the loop

function myFunc(){
  var functionScoped = "I am available
inside this function";
```

```
  console.log(functionScoped);
}
myFunc();
// I am available inside this function
console.log(functionScoped);
// ReferenceError: functionScoped is not
defined
```

In the first example the value of the `var` leaked out of the block-scope and could be accessed from outside of it, whereas in the second example `var` was confined inside a function-scope and we could not access it from outside.

Let

Variables declared with the `let` (and `const`) keyword are **block scoped**, meaning that they will be available only inside of the block where they are declared and its sub-blocks.

```
 // using `let`
let x = "global";

if (x === "global") {
  let x = "block-scoped";
```

```
  console.log(x);
  // expected output: block-scoped
}

console.log(x);
// expected output: global

// using `var`
var y = "global";

if (y === "global") {
  var  y= "block-scoped";

  console.log(y);
  // expected output: block-scoped
}

console.log(y);
// expected output: block-scoped
```

As you can see, when we assigned a new value to the variable declared with `let` inside our block-scope, it **did not** change its value in the outer scope. Whereas, when we did the same with the variable declared with var, it leaked outside of the block-scope and also changed it in the outer scope.

Const

Similarly to `let`, variables declared with `const` are also **block-scoped**, but they differ in the fact that their value **can't change through re-assignment or can't be re-declared.**

```
 const constant = 'I am a constant';
constant = " I can't be reassigned";

// Uncaught TypeError: Assignment to
constant variable
```

Important:
This **does not** mean that variables declared with `const` are immutable.

The content of a const is an Object

```
 const person = {
   name: 'Alberto',
   age: 25,
}

person.age = 26;
```

```
console.log(person.age);
// 26
```

In this case we are not reassigning the whole variable but just one of its properties, which is perfectly fine.

Note: We can still freeze the `const` object, which will not change the contents of the object (but trying to change the values of object `JavaScript` will not throw any error)

```
 const person = {
   name: 'Alberto',
   age: 25,
}

person.age = 26;
console.log(person.age);
// 26

Object.freeze(person)

person.age = 30;

console.log(person.age);
// 26
```

The temporal dead zone

Now we will have a look at a very important concept which may sound complicated from its name, but I assure you it is not.

First let's have a look at a simple example:

```
 console.log(i);
var i = "I am a variable";

//  expected output: undefined

console.log(j);
let j = "I am a let";

// expected output: ReferenceError: can't
access lexical declaration `j' before
initialization
```

`var` can be accessed **before** they are defined, but we can't access their **value**.
`let` and `const` can't be accessed **before we define them**.

Despite what you may read on other sources, both `var` and `let`(and `const`) are subject to **hoisting** which means that they are processed before any code is

executed and lifted up to the top of their scope (whether it's global or block).

The main differences lie in the fact that `var` can still be accessed before they are defined. This causes the value to be `undefined`. While on the other hand, `let` lets the variables sit in a **temporal dead zone** until they are declared. And this causes an error when accessed before initialization, which makes it easier to debug code rather than having an `undefined` as the result.

When to use Var, Let and Const

There is no rule stating where to use each of them and people have different opinions. Here I am going to present to you two opinions from popular developers in the `JavaScript` community.

The first opinion comes from Mathias Bynes:

- Use `const` by default
- Use `let` only if rebinding is needed.
- `var` should never be used in ES6.

The second opinion comes from Kyle Simpson:

- Use `var` for top-level variables that are shared across many (especially larger) scopes.
- Use `let` for localized variables in smaller scopes.

- Refactor `let` to `const` only after some code has to be written, and you're reasonably sure that you've got a case where there shouldn't be variable reassignment.

Which opinion to follow is entirely up to you. As always, do your own research and figure out which one you think is the best.

My personal opinion is to always use `const` by default and then switch to `let` if you see yourself in need of rebinding the value.

End of Chapter 1 Quiz

1.1 What is the correct output of the following code?

```
 var greeting = "Hello";


greeting = "Farewell";


for (var i = 0; i < 2; i++) {
  var greeting = "Good morning";
}


console.log(greeting);
```

- ☐ Hello
- ☐ Good morning
- ☐ Farewell;

1.2 What is the correct output of the following code?

```
 let value = 1;


if(true) {
  let value = 2;
  console.log(value);
}
```

```
value = 3;
```

- ☐ 1
- ☐ 2
- ☐ 3

1.3 What is the correct output of the following code?

```
 let x = 100;


if (x > 50) {
  let x = 10;
}


console.log(x);
```

- ☐ 10
- ☐ 100
- ☐ 50

1.4 What is the correct output of the following code?

```
console.log(constant);


const constant = 1;
```

- ☐ undefined
- ☐ ReferenceError
- ☐ 1

Chapter 2: Arrow functions

What is an arrow function?

ES6 introduced fat arrows (=>) as a way to declare functions.
This is how we would normally declare a function in ES5:

```
const greeting = function(name) {
  return "hello " + name;
}
```

The new syntax with a fat arrow looks like this:

```
var greeting = (name) => {
  return `hello ${name}`;
}
```

We can go further, if we only have one parameter we can drop the parenthesis and write:

```
var greeting = name => {
  return `hello ${name}`;
}
```

If we have no parameter at all we need to write empty parenthesis like this:

```
var greeting = () => {
  return "hello";
}
```

Implicitly return

With arrow functions we can skip the explicit `return` and return like this:

```
const greeting = name => `hello ${name}`;
```

Look at a side by side comparison with an old ES5 Function:

```
const oldFunction = function(name){
  return `hello ${name}`
}

const arrowFunction = name => `hello $
{name}`;
```

Both functions achieve the same result, but the new syntax allows you to be more concise.
Beware! Readability is more important than conciseness so you might want to write your function like this if you are working in a team and not everybody is totally up-to-date with ES6.

```
const arrowFunction = (name) => {
  return `hello ${name}`;
}
```

Let's say we want to implicitly return an **object literal**, we would do it like this:

```
const race = "100m dash";
const runners = [ "Usain Bolt", "Justin Gatlin", "Asafa Powell" ];

const results = runners.map((runner, i) => ({ name: runner, race, place: i + 1}));

console.log(results);
// [{name: "Usain Bolt", race: "100m dash", place: 1}
// {name: "Justin Gatlin", race: "100m dash", place: 2}
// {name: "Asafa Powell", race: "100m dash", place: 3}]
```

In this example, we are using the `map` function to iterate over the array `runners`. The first argument is the current item in the array and the `i` is the index of it. For each item in the array we are then adding into `results` an Object containing the properties `name`, `race`, and `place`.

To tell `JavaScript` what's inside the curly braces is an **object literal** we want to implicitly return, we need to wrap everything inside parenthesis.

Writing `race` or `race: race` is the same.

Arrow functions are anonymous

As you can see from the previous examples, arrow functions are **anonymous**.

If we want to have a name to reference them we can bind them to a variable:

```
const greeting = name => `hello ${name}`;

greeting("Tom");
```

Arrow function and the this keyword

You need to be careful when using arrow functions in conjunction with the `this` keyword, as they behave differently from normal functions.

When you use an arrow function, the `this` keyword is inherited from the parent scope.

This can be useful in cases like this one:

```
 <div class="box open">
        This is a box
</div>
 .opening {
        background-color:red;
}
 // grab our div with class box
const box = document.querySelector(".box");
// listen for a click event
box.addEventListener("click", function() {
  // toggle the class opening on the div
  this.classList.toggle("opening");
  setTimeout(function(){
    // try to toggle again the class
    this.classList.toggle("opening");
    },500);
});
```

The problem in this case is that the first `this` is bound to the `const` box but the second one, inside the `setTimeout`, will be set to the `Window` object, throwing this error:

```
 Uncaught TypeError: cannot read property
"toggle" of undefined
```

Since we know that **arrow functions** inherit the value of `this` from the parent scope, we can re-write our function like this:

```
 const box =
document.querySelector(".box");
// listen for a click event
box.addEventListener("click", function() {
  // toggle the class opening on the div
  this.classList.toggle("opening");
  setTimeout(()=>{
    // try to toggle again the class
    this.classList.toggle("opening");
    },500);
});
```

Here, the second `this` will inherit from its parent, and will be set to the `const` box.

Running the example code you should see our `div` turning red for just half a second.

When you should avoid arrow functions

Using what we know about the inheritance of the `this` keyword we can define some instances where you should **not** use arrow functions.

The next two examples show when to be careful using `this` inside of arrows.

Example 1

```
 const button =
document.querySelector("btn");
button.addEventListener("click", () => {
  // error: *this* refers to the `Window`
Object
  this.classList.toggle("on");
})
```

Example 2

```
 const person1 = {
  age: 10,
  grow: function() {
    this.age++;
    console.log(this.age);
  }
}

person1.grow();
// 11

const person2 = {
  age: 10,
```

```
  grow: () => {
    // error: *this* refers to the `Window`
Object
    this.age++;
    console.log(this.age);
  }
}

person2.grow();
```

Another difference between Arrow functions and normal functions is the access to the `arguments object`.
The `arguments object` is an array-like object that we can access from inside functions and contains the values of the arguments passed to that function.

A quick example:

```
 function example(){
  console.log(arguments[0])
}

example(1,2,3);
// 1
```

As you can see we accessed the first argument using an array notation `arguments[0]`.

Similarly to what we saw with the `this` keyword, Arrow functions inherit the value of the `arguments object` from their parent scope.

Let's have a look at this example with our previous list of runners:

```
const showWinner = () => {
  const winner = arguments[0];
  console.log(`${winner} was the winner`)
}

showWinner( "Usain Bolt", "Justin Gatlin",
"Asafa Powell" )
```

This code will return:

```
ReferenceError: arguments is not defined
```

To access all the arguments passed to the function we can either use the old function notation or the spread syntax(which we will discuss more in Chapter 9)

Remember that `arguments` is just a keyword, it's not a variable name.

Example with **arrow function**:

```
const showWinner = (...args) => {
  const winner = args[0];
  console.log(`${winner} was the winner`)
```

```
}
showWinner("Usain Bolt", "Justin Gatlin",
"Asafa Powell" )
// "Usain Bolt was the winner"
```

Example with **function**:

```
 const showWinner = function() {
   const winner = arguments[0];
   console.log(`${winner} was the winner`)
}
showWinner("Usain Bolt", "Justin Gatlin",
"Asafa Powell")
// "Usain Bolt was the winner"
```

End of Chapter 2 Quiz

2.1 What is the correct syntax for an arrow function?

```
let arr = [1,2,3];


//a)
let func = arr.map(n -> n+1);


//b)
let func = arr.map(n => n+1);


//c)
let func = arr.map(n ~> n+1);
```

- ☐ a
- ☐ b
- ☐ c

2.2 What is the correct output of the following code?

```
const person = {
  age: 10,
  grow: () => {
    this.age++;
  },
```

```
}
person.grow();


console.log(person.age);
```

- ☐ 10
- ☐ 11
- ☐ undefined

2.3 Refactor the following code to use the arrow function syntax:

```
 function(arg) {
  console.log(arg);
}
```

Chapter 3: Default function arguments

Prior to ES6, setting default values to function arguments was not so easy. Let's look at an example:

```
 function
getLocation(city,country,continent){
  if(typeof country === 'undefined'){
    country = 'Italy'
  }
  if(typeof continent === 'undefined'){
    continent = 'Europe'
  }
  console.log(continent,country,city)
}

getLocation('Milan')
// Europe Italy Milan

getLocation('Paris','France')
// Europe France Paris
```

As you can see our function takes three arguments, a city, a country and a continent. In the function body we are checking if either country or continent are

`undefined` and in that case we are giving them a **default value**.

When calling `getLocation('Milan')` the second and third parameter (country and continent) are undefined and get replaced by the default values of our functions.

But what if we want our default value to be at the beginning and not at the end of our list of arguments?

```
 function
getLocation(continent,country,city){
  if(typeof country === 'undefined'){
    country = 'Italy'
  }
  if(typeof continent === 'undefined'){
    continent = 'Europe'
  }
  console.log(continent,country,city)
}

getLocation(undefined, undefined,'Milan')
// Europe Italy Milan

getLocation(undefined,'Paris','France')
// Europe Paris France
```

If we want to replace any of the first arguments with our default we need to pass them as `undefined`, not really a nice looking solution. Luckily ES6 came to the rescue with default function arguments.

Default function arguments

ES6 makes it very easy to set default function arguments. Let's look at an example:

```
 function calculatePrice(total, tax = 0.1,
tip = 0.05){
// When no value is given for tax or tip,
the default 0.1 and 0.05 will be used
return total + (total * tax) + (total *
tip);
}
```

What if we don't want to pass the parameter at all, like this:

```
 // The 0.15 will be bound to the second
argument, tax even if in our intention it
was to set 0.15 as the tip
calculatePrice(100, 0.15)
```

We can solve by doing this:

```
 // In this case 0.15 will be bound to the
tip
calculatePrice(100, undefined, 0.15)
```

It works, but it's not very nice, how to improve it?

With **destructuring** we can write this:

```
 function calculatePrice({
  total = 0,
  tax = 0.1,
  tip = 0.05} = {}){
  return total + (total * tax) + (total *
tip);
}

const bill = calculatePrice({ tip: 0.15,
total:150 });
// 187.5
```

We made the argument of our function an Object. When calling the function, we don't even have to worry about the order of the parameters because they are matched based on their key.

In the example above the default value for *tip* was 0.05 and we overwrote it with 0.15 but we didn't give a value to tax which remained the default 0.1.

Notice this detail:

```
 {
  total = 0,
  tax = 0.1,
  tip = 0.05
} = {}
```

If we don't default our argument Object to an empty Object, and we were to try and run `calculatePrice()` we would get:

```
Cannot destructure property `total` of
'undefined' or 'null'.
```

By writing `= {}` we default our argument to an `Object` and no matter what argument we pass in the function, it will be an `Object`:

```
calculatePrice({});
// 0
calculatePrice();
// 0
calculatePrice(undefined)
// 0
```

No matter what we passed, the argument was defaulted to an `Object` which had three default properties of total, tax and tip.

```
function calculatePrice({
  total = 0,
```

```
  tax = 0.1,
  tip = 0.05}){
  return total + (total * tax) + (total *
tip);
}
calculatePrice({});
// cannot read property `total` of
'undefined' or 'null'.
calculatePrice();
// cannot read property `total` of
'undefined' or 'null'.
calculatePrice(undefined)
// cannot read property `total` of
'undefined' or 'null'.
```

Don't worry about destructuring, we will talk about it in Chapter 10.

End of Chapter 3 Quiz

3.1 Write the code to accomplish the following task:

In the following code, change `arg1` and `arg2` so that the first one represents the `tax` and the second one the `tip` value.

Give `tax` a default value of 0.1 and `tip` a default value of 0.05.

```
 function calculatePrice(total,arg1,arg2) {
   return total + (total * tax) + (total *
tip);
}
calculatePrice(10);
// expected result: 11.5
```

3.2 What is the correct output of the following code?

```
 var b = 3;
function multiply(a,b=2){
   return a * b;
}
console.log(multiply(5));
```

- ☐ 2

- ☐ 5
- ☐ 10
- ☐ 15

Chapter 4: Template literals

Prior to ES6 they were called *template strings*, now we call them *template literals*. Let's have a look at what changed in the way we interpolate strings in ES6.

Interpolating strings

In ES5 we used to write this, in order to interpolate strings:

```
 var name = "Alberto";
var greeting = 'Hello my name is ' + name;

console.log(greeting);
// Hello my name is Alberto
```

In ES6 we can use backticks to make our lives easier.

```
 let name  = "Alberto";
const greeting = `Hello my name is ${name}
`;

console.log(greeting);
// Hello my name is Alberto
```

Expression interpolations

In ES5 we used to write this:

```
var a = 1;
var b = 10;
console.log('1 * 10 is ' + ( a * b));
// 1 * 10 is 10
```

In ES6 we can use backticks to reduce our typing:

```
var a = 1;
var b = 10;
console.log(`1 * 10 is ${a * b}`);
// 1 * 10 is 10
```

Create HTML fragments

In ES5 we used to do this to write multi-line strings:

```
// We have to include a backslash on each
line
var text = "hello, \
my name is Alberto \
how are you?\ "
```

In ES6 we simply have to wrap everything inside backticks, no more backslashes on each line.

```
 const content = `hello,
my name is Alberto
how are you?`;
```

Nesting templates

It's very easy to nest a template inside another one, like this:

```
 const people = [{
         name: 'Alberto',
         age: 27
 },{
         name: 'Caroline',
         age: 27
},{
         name: 'Josh',
         age: 31
}];

const markup = `
<ul>
  ${people.map(person => `<li>  $
{person.name}</li>`)}
</ul>
`;
```

```
console.log(markup);

// <ul>
//   <li>  Alberto</li>,<li>  Caroline</
li>,<li>  Josh</li>
// </ul>
```

Here we are using the `map` function to loop over each of our `people` and display a `li` tag containing the `name` of the person.

Add a ternary operator

We can easily add some logic inside our template string by using a ternary operator.

The syntax for a ternary operator looks like this:

```
 const isDiscounted = false

function getPrice(){
        console.log(isDiscounted ? "$10" :
"$20");
}
getPrice();
// $20
```

If the condition before the `?` can be converted to `true` then the first value is returned, else it's the value after the `:` that gets returned.

```
 // create an artist with name and age
const artist = {
  name: "Bon Jovi",
  age: 56,
};

// only if the artist object has a song
property we then add it to our paragraph,
otherwise we return nothing
const text = `
  <div>
    <p>  ${artist.name} is ${artist.age}
years old ${artist.song ? `and wrote the
song ${artist.song}` : '' }
    </p>
  </div>
`
// <div>
//  <p>  Bon Jovi is 56 years old
//  </p>
// </div>
const artist = {
```

```
  name: "Trent Reznor",
  age: 53,
  song: 'Hurt'
};
// <div>
//   <p>  Trent Reznor is 53 years old and
wrote the song Hurt
//   </p>
// </div>
```

Pass a function inside a template literal

Similarly to the example above (line 10 of the code), if we want to, we can pass a function inside a template literal.

```
 const groceries = {
  meat: "pork chop",
  veggie: "salad",
  fruit: "apple",
  others: ['mushrooms', 'instant noodles',
'instant soup'],
}

// this function will map each individual
value of our groceries
```

```
function groceryList(others) {
  return `
    <p>
      ${others.map( other => ` <span>${other}</span>`).join('\n')}
    </p>
  `;
}

// display all our groceries in a p tag, the last one will include all the one from the array **others**
const markup = `
  <div>
    <p>${groceries.meat}</p>
    <p>${groceries.veggie}</p>
    <p>${groceries.fruit}</p>
    <p>${groceryList(groceries.others)}</p>
  <div>
`
//  <div>
//      <p>pork chop</p>
//      <p>salad</p>
//      <p>apple</p>
//      <p>
```

```
//      <p>
//          <span>mushrooms</span>
//          <span>instant noodles</span>
//          <span>instant soup</span>
//      </p>
//    </p>
//    <div>
```

Inside of the last `p` tag we are calling our function `groceryList` passing it all the `others` groceries as an argument.
Inside of the function we are returning a `p` tag and we are using `map` to loop over each of our items in the grocery list returning an Array of `span` tags containing each grocery. We are then using `.join('\n')` to add a new line after each of those span.

Tagged template literals

By tagging a function to a template literal we can run the template literal through the function, providing it with everything that's inside of the template.

The way it works is very simple: you just take the name of your function and put it in front of the template that you want to run it against.

```
 let person = "Alberto";
let age = 25;

function
myTag(strings,personName,personAge){
  let str = strings[1];
  let ageStr;

  personAge > 50 ? ageStr = "grandpa" :
ageStr = "youngster";

  return personName + str + ageStr;
}

let sentence = myTag`${person} is a ${age}
`;
console.log(sentence);
// Alberto is a youngster
```

We captured the value of the variable age and used a ternary operator to decide what to print.
`strings` will take all the strings of our `let` sentence whilst the other parameters will hold the variables.

In our example our string is divided in 3 pieces: `${person}`, `is a` and `${age}`.
We use array notation to access the string in the middle like this:

```
let str = strings[1];
```

To learn more about use cases of *template literals* check out this article (https://codeburst.io/javascript-es6-tagged-template-literals-a45c26e54761).

End of Chapter 4 Quiz

4.1 Write the code to accomplish the following task:

Utilizing template literals, combine the different variables to achieve the expected output.

```
 let a = "Hello,";
let b = "is";
let c = "my";
let d = "name";
let e = "Tom";

// edit result to achieve the expected
output
let result = '';

console.log(result);
//expected output
"Hello, my name is Tom"
```

4.2 Refactor the following code using template literals

```
 let a = "1";
let b = "2";
let c = "plus";
```

```
let d = "3";
let e = "equals";

// edit result to use template literals
let result = a + " " + c + " " + b + " " +
e + " " + d;

console.log(result);
// 1 plus 2 equals 3
```

4.3 Refactor the following code using template literals

```
 // edit str to use template literals
let str = 'this is a very long text\n' +
'a very long text';

console.log(str);

// this is a very long text
// a very long text
```

Chapter 5: Additional string methods

There are many methods that we can use against strings. Here's a list of a few of them:

`indexOf()`

Gets the position of the first occurrence of the specified value in a string.

```
const str = "this is a short sentence";
str.indexOf("short");
// Output: 10
```

`slice()`

Pulls a specified part of a string as a new string.

```
const str = "pizza, orange, cereals"
str.slice(0, 5);
// Output: "pizza"
```

`toUpperCase()`

Turns all characters of a string to uppercase.

```
const str = "i ate an apple"
str.toUpperCase()
// Output: "I ATE AN APPLE"
```

`toLowerCase()`

Turns all characters of a string to lowercase.

```
const str = "I ATE AN APPLE"
str.toLowerCase()
// Output: "i ate an apple"
```

There are many more methods; these were just a few as a reminder. Check the MDN documentation for a more in-depth description on the above methods.

Additional string methods

ES6 introduced 4 new string methods:

- `startsWith()`
- `endsWith()`
- `includes()`
- `repeat()`

`startsWith()`

This new method will check if the string starts with the value we pass in:

```
const code = "ABCDEFG";
```

```
code.startsWith("ABB");
// false
code.startsWith("abc");
// false, startsWith is case sensitive
code.startsWith("ABC");
// true
```

We can pass an additional parameter, which is the starting point where the method will begin checking.

```
 const code = "ABCDEFGHI"

code.startsWith("DEF",3);
// true, it will begin checking after 3
characters
```

endsWith()

Similarly to `startsWith()` this new method will check if the string ends with the value we pass in:

```
 const code = "ABCDEF";

code.endsWith("DDD");
// false
code.endsWith("def");
// false, endsWith is case sensitive
```

```
code.endsWith("DEF");
// true
```

We can pass an additional parameter, which is the number of digits we want to consider when checking the ending.

```
 const code = "ABCDEFGHI"

code.endsWith("EF", 6);
// true, 6 means that we consider only the
first 6 values ABCDEF, and yes this string
ends with EF therefore we get *true*
```

includes()

This method will check if our string includes the value we pass in.

```
 const code = "ABCDEF"

code.includes("ABB");
// false
code.includes("abc");
// false, includes is case sensitive
code.includes("CDE");
// true
```

`repeat()`

As the name suggests, this new method will take an argument that specifies the number of times it needs to repeat the `string`.

```
 let hello = "Hi";
console.log(hello.repeat(10));
// "HiHiHiHiHiHiHiHiHiHi"
```

End of Chapter 5 Quiz

5.1 What is the correct output of the following code?

```
const code = "ABCDEFGHI";


code.startsWith("DEF",3);
```

- ☐ true
- ☐ false

5.2 What is the correct output of the following code?

```
const code = "ABCDEF";
code.endsWith("def");
```

- ☐ true
- ☐ false

5.3 Write the code to accomplish the following task:

```
let str = "Na";
let bat = "BatMan";


let batman = ...
console.log(batman);
```

```
// expected output: "NaNaNaNaNaNaNaNa
Batman"
```

Chapter 6: Destructuring

MDN defines **destructuring** like this:

The **destructuring** assignment syntax is a `JavaScript` expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.

Let's start with **destructuring objects** first.

Destructuring Objects

To create variables from an object, we used to do the following:

```
var person  = {
  first: "Alberto",
  last: "Montalesi"
}

var first = person.first;
var last = person.last;
```

In ES6 we can instead do it as following:

```
const person = {
  first: "Alberto",
  last: "Montalesi"
```

```
}


const { first, last } = person;
```

Since our `const` variable, `person`, have the same name as the properties we want to grab, we don't have to specify `person.first` and `person.last` anymore.

The same applies even when we have nested data, such as what we could get from an API.

```
 const person = {
  name: "Alberto",
  last: "Montalesi",
  links:{
    social: {
      facebook: "https://www.facebook.com/
alberto.montalesi",
    },
    website: "http://
albertomontalesi.github.io/"
  }
}

const { facebook } = person.links.social;
```

We are not limited to name our variable the same as the property of the object. We can also rename as the following:

```
 const { facebook:fb } =
person.links.social;
// it will look for the property
person.links.social.facebook and name the
variable fb
console.log(fb); // https://
www.facebook.com/alberto.montalesi
console.log(facebook); //ReferenceError:
facebook is not defined
```

We are using the syntax `const { facebook:fb }` to specify that we want to target the property `facebook` of the Object `person.links.social` and we want the `const` variable to be called `fb`.
That is why when we try to log `facebook` we get an error.

We can also pass in **default values** like this:

```
 const { facebook:fb = "https://
www.facebook.com"} = person.links.social;
// we renamed the variable to *fb* and we
also set a default value to it
```

Destructuring Arrays

The first difference we notice when **destructuring arrays** is that we are going to use `[]` and not `{}`.

```
const person = ["Alberto","Montalesi",25];
const [name,surname,age] = person;
```

What if the number of variables that we create is less than the elements in the array?

```
const person = ["Alberto","Montalesi",25];
// we leave out age, we don't want it
const [name,surname] = person;
//the value of age will not be bound to any variable.
console.log(name,surname);
// Alberto Montalesi
```

Let's say we want to grab all the other values remaining, we can use the **rest operator**:

```
const person = ["Alberto", "Montalesi", "pizza", "ice cream", "cheese cake"];
// we use the **rest operator** to grab all the remaining values
const [name,surname,...food] = person ;
console.log(food);
```

```
// Array [ "pizza", "ice cream", "cheese
cake" ]
```

In the example above the first two values of the array were bound to `name` and `surname` while the rest (that's why it's called the **rest operator**) get assigned to `food`

The `...` is the syntax for the **rest operator**.

Swapping variables with destructuring

The destructuring assignment makes it **extremely easy** to swap variables, just look at this example:

```
let hungry = "yes";
let full = "no";
// after we eat we don't feel hungry
anymore, we feel full, let's swap the
values

[hungry, full] = [full, hungry];
console.log(hungry,full);
// no yes
```

It can't get easier than this to swap values.

End of Chapter 6 Quiz

6.1 Write the code to accomplish the following task:

Use destructuring to swap the value of the two variables.

```
let hungry = "yes";
let full = "no";

// your code goes here

console.log(hungry);
// no
console.log(full);
// yes
```

6.2 Write the code to accomplish the following task:

In one line of code, declare one variable to store each one of the values of the following array.

```
let arr = [ "one", "two", "three" ];

//expected output
console.log(one);
// "one"
```

```
console.log(two);
// "two"
console.log(three);
// "three"
```

Chapter 7: Iterables and looping

The for of loop

ES6 introduced a new type of loop, the `for of` loop. Let's take a look at how it is used

Iterating over an array

Usually, we would iterate using the following method:

```
 var fruits = ['apple','banana','orange'];
for (var i = 0; i < fruits.length; i++){
  console.log(fruits[i]);
}
// apple
// banana
// orange
```

This is a normal loop where at each iteration we increase the value of `i` by 1 as long as it is less than `fruits.length`. At that point the loop will stop.

Look at how we can achieve the same with a `for of` loop:

```
 const fruits =
['apple','banana','orange'];
for(const fruit of fruits){
```

```
  console.log(fruit);
}
// apple
// banana
// orange
```

Iterating over an object

Objects are **non iterable** so how do we iterate over them?
We have to first grab all the values of the object using something like `Object.keys()` or the new ES6 function: `Object.entries()`.

```
 const car = {
  maker: "BMW",
  color: "red",
  year : "2010",
}

for (const prop of Object.keys(car)){
  const value = car[prop];
  console.log(prop,value);
}
// maker BMW
```

```
// color red
// year 2010
```

The for in loop

Even though it is not a new ES6 loop, let's look at the `for in` loop to understand what differentiates it to the `for of` loop.

The `for in` loop is a bit different because it will iterate over all the enumerable properties of an object in no particular order.

It is therefore suggested not to add, modify or delete properties of the object during the iteration. As there is no guarantee that they will be visited, or if they will be visited before or after being modified.

```
const car = {
  maker: "BMW",
  color: "red",
  year : "2010",
}
for (const prop in car){
  console.log(prop, car[prop]);
}
// maker BMW
```

```
// color red
// year 2010
```

Difference between for of and for in

The first difference we can see is by looking at this example:

```
let list = [4, 5, 6];

// for...in returns a list of keys
for (let i in list) {
   console.log(i); // "0", "1", "2",
}


// for ...of returns the values
for (let i of list) {
   console.log(i); // "4", "5", "6"
}
```

`for in` will return a list of keys whereas the `for of` will return a list of values of the numeric properties of the object being iterated.

End of Chapter 7 Quiz

7.1 Which one of these loops was introduced in ES6?

- ☐ while
- ☐ for of
- ☐ for in

What is the correct output of the following code?

```
 let people = [ "Tom", "Jerry", "Mickey" ];


for (let person of people){
  console.log(person);
}
```

- ☐ Tom Jerry
- ☐ Tom
- ☐ Tom Jerry Mickey
- ☐ Mickey

Chapter 8: Array improvements

`Array.from()`

`Array.from()` is the first of many new array methods that ES6 introduced.

It will take something **arrayish**, meaning something that looks like an array but isn't, and transform it into a real array.

```
 <div class="fruits">
   <p> Apple </p>
   <p> Banana </p>
   <p> Orange </p>
</div>
 const fruits =
document.querySelectorAll(".fruits p");
// fruits is a nodelist (an array-like
collection) containng our three p tags
// now we convert it in an Array
const fruitArray = Array.from(fruits);

console.log(fruitArray);
// [p,p,p]

//since now we are dealing with an array we
can use map
```

```
const fruitNames = fruitArray.map( fruit =>
fruit.textContent);

console.log(fruitNames);
// ["Apple", "Banana", "Orange"]
```

We can also simplify like this:

```
 const fruits =
Array.from(document.querySelectorAll(".frui
ts p"));
const fruitNames = fruits.map(fruit =>
fruit.textContent);

console.log(fruitNames);
// ["Apple", "Banana", "Orange"]
```

Now we transformed **fruits** into a real array, meaning that we can use any sort of method such as `map` on it.

`Array.from()` also takes a second argument, a `map` function so we can write:

```
 const fruits =
document.querySelectorAll(".fruits p");
const fruitArray = Array.from(fruits, fruit
=> {
  console.log(fruit);
```

```
  // <p> Apple </p>
  // <p> Banana </p>
  // <p> Orange </p>
  return fruit.textContent;
  // we only want to grab the content not
the whole tag
});
console.log(fruitArray);
// ["Apple", "Banana", "Orange"]
```

In the example above we passed a `map` function to the `.from()` method to push into our newly formed array only the `textContent` of the `p` tags and not the whole tag.

`Array.of()`

`Array.of()` will create an array with all the arguments we pass into it.

```
 const digits = Array.of(1,2,3,4,5);
console.log(digits);


// Array [ 1, 2, 3, 4, 5];
```

Array.find()

`Array.find()` returns the value of the first element in the array that satisfies the provided testing function. Otherwise `undefined` is returned.

Let's looks at a simple example to see how `Array.find()` works.

```
const array = [1,2,3,4,5];

// this will return the first element in
the array with a value higher than 3
let found = array.find( e => e > 3 );
console.log(found);
// 4
```

As we mentioned, it will return the **first element** that matches our condition, that's why we only got **4** and not **5**.

Array.findIndex()

`Array.findIndex()` will return the *index* of the **first** element that matches our condition.

```
const greetings =
["hello","hi","byebye","goodbye","hi"];
```

```
let foundIndex = greetings.findIndex(e => e
=== "hi");
console.log(foundIndex);
// 1
```

Again, only the index of the **first element** that matches our condition is returned.

`Array.some()` & `Array.every()`

I'm grouping these two together because their use is self-explanatory: `.some()` will search if there are *some* items matching the condition and stop once it finds the first one. Whereas, .every() will check if all items match the given condition or not.

```
const array = [1,2,3,4,5,6,1,2,3,1];

let arraySome = array.some( e => e > 2);
console.log(arraySome);
// true

let arrayEvery = array.every(e => e > 2);
console.log(arrayEvery);
// false
```

Simply put, the first condition is true, because there are **some** elements greater than 2, but the second is false because **not every element** is greater than 2.

End of Chapter 8 Quiz

8.1 Write the code to accomplish the following task:

Given the following code, create a new array starting from the following string where each letter of the string becomes an item in the array.

```
const apple = "Apple";


const myArr = // add your code here
console.log(myArry);
// expected output = ["A", "p", "p", "l",
"e"]
```

8.2 What is the correct output of the following code?

```
const array = [1,2,3,4,5];
let found = array.find( e => e > 3 );


console.log(found);
```

- ☐ 3
- ☐ 5
- ☐ 4,5
- ☐ 4

8.3 What is the correct output of the following code?

```
 const array = [1,2,3,4,5,6,1,2,3,1];
let arraySome = array.some( e => e > 2);


console.log(arraySome);
```

- ☐ 2
- ☐ false
- ☐ 3
- ☐ true

8.4 What is the correct output of the following code:

```
Array.from([1, 2, 3], x => x * x);
```

- ☐ [1,2,3]
- ☐ [1,4,9]
- ☐ [1,3,5]

Chapter 9: Spread operator and rest parameters

The Spread operator

According to MDN:

Spread syntax allows an iterable such as an array expression or string to be expanded in places where zero or more arguments (for function calls) or elements (for array literals) are expected, or an object expression to be expanded in places where zero or more key-value pairs (for object literals) are expected.

Combine arrays

```
 const veggie =
["tomato","cucumber","beans"];
const meat = ["pork","beef","chicken"];


const menu = [...veggie, "pasta", ...meat];
console.log(menu);
// Array [ "tomato", "cucumber", "beans",
"pasta", "pork", "beef", "chicken" ]
```

The `...` is the spread syntax, and it allowed us to grab all the individual values of the arrays veggie and meat

and put them inside the array menu and at the same time add a new item in between them.

Copy arrays

The spread syntax is very helpful if we want to create a copy of an array.

```
 const veggie =
["tomato","cucumber","beans"];
const newVeggie = veggie;

// this may seem like we created a copy of
the veggie array but look now

veggie.push("peas");
console.log(veggie);
// Array [ "tomato", "cucumber", "beans",
"peas" ]

console.log(newVeggie);
// Array [ "tomato", "cucumber", "beans",
"peas" ]
```

Our new array also changed, but why? Because we did not actually create a copy but we just referenced our old array in the new one.

The following is how we would usually make a **copy** of an array in ES5 and earlier

```
 const veggie =
["tomato","cucumber","beans"];
// we created an empty array and put the
values of the old array inside of it
const newVeggie = [].concat(veggie);
veggie.push("peas");
console.log(veggie);
// Array [ "tomato", "cucumber", "beans",
"peas" ]
console.log(newVeggie);
// Array [ "tomato", "cucumber", "beans" ]
```

And this is how we would do the same using the spread syntax:

```
 const veggie =
["tomato","cucumber","beans"];
const newVeggie = [...veggie];
veggie.push("peas");
console.log(veggie);
// Array [ "tomato", "cucumber", "beans",
"peas" ]
console.log(newVeggie);
// Array [ "tomato", "cucumber", "beans" ]
```

The syntax for the **Spread** operator is the following `...YourArray`. Since we wanted the variable `newVeggie` to be a copy of the array `veggie` we assigned it to an Array and inside of it we spread all the values from the variable `veggie` like so: `[...veggie]`.

Spread into a function

Thanks to the **Spread** operator we have an easier way of calling a function with an Array of arguments.

```
 // OLD WAY
function doStuff (x, y, z) {
  console.log(x + y + z);
 }
var args = [0, 1, 2];

// Call the function, passing args
doStuff.apply(null, args);

// USE THE SPREAD SYNTAX

doStuff(...args);
// 3 (0+1+2);
console.log(args);
// Array [ 0, 1, 2 ]
```

As you can see, in the example, our `doStuff` function accepts 3 parameters and we are passing them by spreading the array `args` into the function like so: `...args` replacing the need of resorting to use `.apply()`.

Let's look at another example:

```
const name = ["Alberto", "Montalesi"];

function greet(first, last) {
  console.log(`Hello ${first} ${last}`);
}

greet(...name);
// Hello Alberto Montalesi
```

The two values of the array are automatically assigned to the two arguments of our function.

What if we provide more values than arguments?

```
const name = ["Jon", "Paul", "Jones"];

function greet(first, last) {
  console.log(`Hello ${first} ${last}`);
}
greet(...name);
// Hello Jon Paul
```

In the example given above, we provided three values inside the array but we only have two arguments in our function therefore the last one is left out.

Spread in Object Literals (ES 2018 / ES9)

This feature is not part of ES6, but as we are already discussing this topic, it is worth mentioning that ES2018 introduced the Spread operator for Objects. Let's look at an example:

```
 let person = {
   name : "Alberto",
   surname: "Montalesi",
   age: 25,
}

let clone = {...person};
console.log(clone);
// Object { name: "Alberto", surname:
"Montalesi", age: 25 }
```

You can read more about ES2018 in Chapter 20.

The Rest parameter

The **rest** syntax looks exactly the same as the **spread**, 3 dots `...` but it is quite the opposite of it. **Spread** expands an array, while **rest** condenses multiple elements into a single one.

```
const runners = ["Tom", "Paul", "Mark",
"Luke"];
const [first,second,...losers] = runners;

console.log(...losers);
// Mark Luke
```

We stored the first two values inside the `const` first and second and whatever was left we put it inside losers using the rest parameter.

End of Chapter 9 Quiz

9.1 What is the correct syntax to spread the values of an array?

- ☐ `[.]`
- ☐ `(...)`
- ☐ `[...]`
- ☐ `{...}`

9.2 Write the code to accomplish the following task:

Given the `const` veggie and meat. Create a new array called menu containing:

- all the values of `veggie`
- a new string of value `'pasta'`
- all the values of `meat`

```
 const veggie =
["tomato","cucumber","beans"];
const meat = ["pork","beef","chicken"];
```

9.3 Write the code to achieve the following task:

Given the following `Array` runners, create a new `Array` called losers that will contain all the values after the first two:

```
const runners = ["Tom", "Paul", "Mark", "Luke"];
```

9.4 What is the correct output of the following code?

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

let arr2 = arr;

arr2.push(5);
console.log(arr);
```

- ☐ `[1, 2, 3, 4]`
- ☐ `[1, 2, 4, 5]`
- ☐ `[1, 2, 3, 4, 5]`
- ☐ `"1, 2, 3, 4, 5"`

Chapter 10: Object literal upgrades

In this article we will look at the many upgrades brought by ES6 to the **Object literal** notation.

Deconstructing variables into keys and values

This is our initial situation:

```
const name = "Alberto";
const surname = "Montalesi";
const age = 25;
const nationality = "Italian";
```

Now if we wanted to create an object literal this is what we would usually do:

```
const person = {
  name: name,
  surname: surname,
  age: age,
  nationality: nationality,
}

console.log(person);
```

```
// { name: 'Alberto',
// surname: 'Montalesi',
// age: 25,
// nationality: 'Italian' }
```

In ES6 we can simplify like this:

```
const person = {
  name,
  surname,
  age,
  nationality,
}
console.log(person);
// {name: "Alberto", surname: "Montalesi",
age: 25, nationality: "Italian"}
```

As our `const` are named the same way as the properties we are using, we can reduce our typing.

Add functions to our Objects

Let's looks at an example from ES5:

```
const person = {
  name: "Alberto",
  greet: function(){
```

```
    console.log("Hello");
  },
}

person.greet();
// Hello
```

If we wanted to add a function to our Object we had to use the the `function` keyword. In ES6 it got easier, look here:

```
const person = {
  name: "Alberto",
  greet(){
    console.log("Hello");
  },
}

person.greet();
// Hello;
```

No more `function`, it's shorter and it behaves the same way.

Remember that **arrow functions** are anonymous, look at this example:

```
// this won't work, you need a key to
access the function
```

```
const person1 = {
  () => console.log("Hello"),
};

const person2 = {
  greet: () => console.log("Hello"),
}
person2.greet()
// Hello
```

Dynamically define properties of an Object

This is how we would dynamically define properties of an Object in ES5:

```
 var name = "myname";
// create empty object
var person = {}
// update the object
person[name] = "Alberto";
console.log(person.myname);
// Alberto
```

In the example given above, first we created the Object and then we modified it. However, in ES6 we

can do both things at the same time. Take a look at the following example:

```
 const name = "myname";
const person = {
  [name]:"Alberto",
};
console.log(person.myname);
// Alberto
```

End of Chapter 10 Quiz

10.1 Refactor the code to make it more concise:

```
const animal = {
  name: name,
  age: age,
  breed: breed,
}
```

10.2 What is the correct output of this code:

```
const name = "myname";
const person = {
  [name]:"Alberto",
};
console.log(person.myname);
```

- ☐ name
- ☐ "Alberto"
- ☐ myname
- ☐ "name"

10.3 What is the correct output of this code:

```
const name = "myname";
const age = 27;
const favoriteColor = "Green";
```

```
const person = {
  name,
  age,
  color
};
console.log(person.color);
```

- [ ] "Green"
- [ ] color
- [ ] color is not defined
- [ ] favoriteColor

Chapter 11: Symbols

ES6 added a new type of primitive called **Symbols**. What are they? And what do they do?

The unique property of Symbols

Symbols are **always unique** and we can use them as identifiers for object properties.

Let's create a `Symbol` together:

```
const me = Symbol("Alberto");
console.log(me);
// Symbol(Alberto)
```

We said that they are always unique, let's try to create a new symbol with the same value and see what happens:

```
const me = Symbol("Alberto");
console.log(me);
// Symbol(Alberto)

const clone = Symbol("Alberto");
console.log(clone);
// Symbol(Alberto)
```

```
console.log(me == clone);
// false
console.log(me === clone);
// false
```

They both have the same value, but we will never have naming collisions with Symbols as they are always unique.

Identifiers for object properties

As we mentioned earlier we can use them to create identifiers for object properties, so let's see an example:

```
 const office = {
   "Tom" : "CEO",
   "Mark": "CTO",
   "Mark": "CIO",
}

for (person in office){
   console.log(person);
}
// Tom
// Mark
```

Here we have our office object with 3 people, two of which share the same name.
To avoid naming collisions we can use symbols.

```
const office = {
  [Symbol("Tom")] : "CEO",
  [Symbol("Mark")] : "CTO",
  [Symbol("Mark")] : "CIO",
}

for(person in office) {
  console.log(person);
}
// undefined
```

We got `undefined` when we tried to loop over the symbols because they are **not enumerable**, so we can't loop over them with a `for in`.

If we want to retrieve their object properties we can use `Object.getOwnPropertySymbols()`.

```
const office = {
  [Symbol("Tom")] : "CEO",
  [Symbol("Mark")] : "CTO",
  [Symbol("Mark")] : "CIO",
};
```

```
const symbols =
Object.getOwnPropertySymbols(office);
console.log(symbols);
// 0: Symbol(Tom)
// 1: Symbol(Mark)
// 2: Symbol(Mark)
// length: 3
```

We retrieved the array but to be able to access the properties we have to use `map`.

```
 const symbols =
Object.getOwnPropertySymbols(office);
const value = symbols.map(symbol =>
office[symbol]);
console.log(value);
// 0: "CEO"
// 1: "CTO"
// 2: "CIO"
// length: 3
```

Now we finally got the array containing all the values of our symbols.

End of Chapter 11 Quiz

11.1 What is a `symbol`?

- ☐ a caveman
- ☐ a property
- ☐ a primitive
- ☐ a type of function

11.2 What is the main characteristic of a `symbol`?

- ☐ they will throw an error when trying to re-assign their value
- ☐ they don't work inside a for loop
- ☐ they are unique
- ☐ they can only hold strings and not integers

11.3 What is the correct output of the following code?

```
const friends = {
  "Tom" : "bff",
  "Jim" : "brother",
  "Tom" : "cousin",
}
```

```
for (friend in friends){
  console.log(friend);
}
```

- ☐ Jim Tom
- ☐ Error
- ☐ Tom Jim Tom
- ☐ Tom Jim

11.4 What is the correct output of the following code?

```
 const family = {
  [Symbol("Tom")] : "father",
  [Symbol("Jane")] : "mother",
  [Symbol("Tom")] : "brother",
};

const symbols =
Object.getOwnPropertySymbols(family);
console.log(symbols);
```

- ☐ Symbol(Tom) Symbol(Jane) Symbol(Tom)
- ☐ Symbol(Tom) Symbol(Jane)
- ☐ undefined
- ☐ Symbol(Jane) Symbol(Tom)

Chapter 12: classes

Quoting MDN:

classes are primarily syntactic sugar over `JavaScript`'s existing prototype-based inheritance. The class syntax **does not** introduce a new object-oriented inheritance model to `JavaScript`.

That being said, let's review prototypal inheritance before we jump into classes.

```
 function Person(name,age) {
   this.name = name;
   this.age = age;
}

Person.prototype.greet = function(){
   console.log("Hello, my name is " +
this.name);
}

const alberto = new Person("Alberto", 26);
const caroline = new Person("Caroline",26);

alberto.greet();
// Hello, my name is Alberto
```

```
caroline.greet();
// Hello, my name is Caroline
```

We added a new method to the prototype in order to make it accessible to all the new instances of `Person` that we created.

Ok, now that I refreshed your knowledge of prototypal inheritance, let's have a look at classes.

Create a class

There are two ways of creating a class:

- `class` declaration
- `class` expression

```
 // class declaration
class Person {


}

// class expression
const person = class Person {
}
```

Remember: `class` declaration (and expression) are **not hoisted** which means that unless you want to get a

ReferenceError you need to declare your class before you access it.

Let's start creating our first `class`.

We only need a method called `constructor` (remember to add only one constructor, a `SyntaxError` will be thrown if the class contains more than one constructor method).

```
class Person {
  constructor(name,age){
    this.name = name;
    this.age = age;
  }
  greet(){
    console.log(`Hello, my name is ${this.name} and I am ${this.age} years old`);
  } // no commas in between methods
  farewell(){
    console.log("goodbye friend");
  }
}

const alberto = new Person("Alberto",26);

alberto.greet();
```

```
// Hello, my name is Alberto and I am 26
years old
alberto.farewell();
// goodbye friend
```

As you can see everything works just like before. As we mentioned at the beginning, `classes` are just a syntactic sugar, a nicer way of doing inheritance.

Static methods

Right now the two new methods that we created, `greet()` and `farewell()` can be accessed by every new instance of `Person`, but what if we want a method that can only be accessed by the class itself, similarly to `Array.of()` for arrays?

```
class Person {
  constructor(name,age){
    this.name = name;
    this.age = age;
  }
  static info(){
    console.log("I am a Person class, nice
to meet you");
  }
}
```

```
const alberto = new Person("Alberto",26);

alberto.info();
// TypeError: alberto.info is not a
function

Person.info();
// I am a Person class, nice to meet you
```

set and get

We can use setter and getter methods to set and get values inside our `class`.

```
class Person {
  constructor(name,surname) {
    this.name = name;
    this.surname = surname;
    this.nickname = "";
  }
  set nicknames(value){
    this.nickname = value;
    console.log(this.nickname);
  }
  get nicknames(){
     console.log(`Your nickname is $
```

```
{this.nickname}`);
  }
}

const alberto = new
Person("Alberto","Montalesi");

// first we call the setter
alberto.nicknames = "Albi";
// "Albi"

// then we call the getter
alberto.nicknames;
// "Your nickname is Albi"
```

Extending our class

What if we want to have a new `class` that inherits from our previous one? We use `extends` keyword for this purpose. Take a look at the following example:

```
 // our initial class
class Person {
  constructor(name,age){
    this.name = name;
    this.age = age;
```

```
  }
  greet(){
    console.log(`Hello, my name is $
{this.name} and I am ${this.age} years old`
);
  }
}


// our new class
class Adult extends Person {
  constructor(name,age,work){
    this.name = name;
    this.age = age;
    this.work = work;
  }
}

const alberto = new
Adult("Alberto",26,"software developer");
```

We created a new `class Adult that inherits fromPerson` but if you try to run this code you will get an error:

```
ReferenceError: must call super
constructor before using |this| in Adult
class constructor
```

The error message tells us to call `super()` before using `this` in our new `class`.
What it means is that we basically have to create a new Person before we create a new Adult and the `super()` constructor will do exactly that.

```
class Adult extends Person {
  constructor(name,age,work){
    super(name,age);
    this.work = work;
  }
}
```

Why did we set `super(name,age)` ? Because our `Adult` class inherits name and age from the `Person` therefore we don't need to redeclare them.
Super will simply create a new Person for us.

If we now run the code again we will get this:

```
// our initial class
class Person {
  constructor(name,age){
    this.name = name;
    this.age = age;
```

```
  }
  greet(){
    console.log(`Hello, my name is ${this.name} and I am ${this.age} years old`);
  }
}


// our new class
class Adult extends Person {
  constructor(name,age,work){
    super(name,age);
    this.work = work;
  }
}

const alberto = new Adult("Alberto",26,"software developer");

console.log(alberto.age);
// 26
console.log(alberto.work);
// "software developer"
alberto.greet();
```

```
// Hello, my name is Alberto and I am 26
years old
```

As you can see our `Adult` inherited all the properties and methods from the `Person` class.

Extending Arrays

We want to create something like this, something similar to an array where the first value is a property to define our `class`room and the rest are our students and their marks.

```
 // we create a new `class`room
const my`class` = new `class`room('1A',[
  {name: "Tim", mark: 6},
  {name: "Tom", mark: 3},
  {name: "Jim", mark: 8},
  {name: "Jon", mark: 10},]
);
```

What we can do is create a new `class` that extends the array.

```
 class `class`room extends Array {
  // we use rest operator to grab all the
students
  constructor(name, ...students){
```

```
    // we use spread to place all the
students in the array individually
otherwise we would push an array into an
array
    super(...students);
    this.name = name;
    // we create a new method to add
students
  }
  add(student){
    this.push(student);
  }
}
const my`class` = new `class`room('1A',
  {name: "Tim", mark: 6},
  {name: "Tom", mark: 3},
  {name: "Jim", mark: 8},
  {name: "Jon", mark: 10},
);

// now we can call
my`class`.add({name: "Timmy", mark:7});
my`class`[4];
// Object { name: "Timmy", mark: 7 }
```

```
// we can also loop over with for of
for(const student of my`class`) {
  console.log(student);
  }
// Object { name: "Tim", grade: 6 }
// Object { name: "Tom", grade: 3 }
// Object { name: "Jim", grade: 8 }
// Object { name: "Jon", grade: 10 }
// Object { name: "Timmy", grade: 7 }
```

End of Chapter 12 Quiz

12.1 What is a `class`?

- [ ] a new primitive
- [ ] just syntactic sugar to perform prototypal inheritance
- [ ] a new type of array

12.2 How can you declare a `class`?

- [ ] `const person = class Person {...}`
- [ ] `const person = new class Person {...}`
- [ ] `class Person {...}`

12.3 What is a `static` method?

- [ ] A method that can't change
- [ ] A method that can be accessed by every instance of a class
- [ ] A method that can be accessed only by the class itself

12.4 What is the correct output of the following code?

```
class Person {
  constructor(name,age){
    this.name = name;
    this.age = age;
  }
}
class Adult extends Person {
  constructor(work){
    this.work = work;
  }
}

const my = new Adult('software developer');
console.log(my.work);
```

- ☐ "software developer"
- ☐ 'Error: age is not defined'
- ☐ ReferenceError: Must call super constructor in derived class before accessing 'this'

12.5 Write the code to accomplish the following task:

Given the following `class`, create a new one called `Adult` that extends the previous one and adds new property to it called `work`.

```
 // our initial class
class Person {
  constructor(name,age){
    this.name = name;
    this.age = age;
  }
  greet(){
    console.log(`Hello, my name is ${this.name} and I am ${this.age} years old`);
  }
}

// create a new class that extends Person and adds new property to it
const me = new Adult('Alberto',27,'software developer');
console.log(me.work);
// software developer
```

Chapter 13: Promises

`JavaScript` work **synchronously** which means that each block of code will be executed after the previous one.

```
 const data = fetch('your-api-url-goes-
here');
 console.log("Finished");
 console.log(data);
}
```

In the example above we are using `fetch` to retrieve data from a url (in this example we are only pretending to be doing so).

In case of synchronous code we would expect the line subsequent line to be called only after the `fetch` has been complete but in reality what is going to happen is that `fetch` will be called and straight away the subsequent two `console.log` will also be called, resulting in the last one `console.log(data)` to return `undefined`.

This happens because `fetch` performs **asynchronously**, which means that the code **won't stop** running once it hits that line but, instead, will continue executing.

What we need, is a way of waiting until `fetch` returns us something before we continue executing our code.

To avoid this we would use **callbacks** or **promises**.

Callback hell

You may have heard of something called **callback hell**, which is something that happens when we try to write **asynchronous** code as if it was **synchronous**, meaning that we try to chain each block of code after the other.

In simple words it would something like:

do A, If A do B, if B do C and so on and so forth.

The following is an example just to showcase the meaning of the **callback hell**. Imagine each step is **asynchronous** meaning that we need to send a request to our server for each step of preparing the pizza and we need to wait for the server to respond.

```
 const makePizza = (ingredients, callback)
=> {
    mixIngredients(ingredients,
function(mixedIngredients)){
      bakePizza(mixedIngredients,
function(bakedPizza)){
        console.log('finished!')
```

```
        }
    }
}
```

We try to write our code in a way where executions happens visually from top to bottom, causing excessive nesting on functions and result in what you can see above.

To improve your callbacks you can check out http://callbackhell.com/

Promises will help us escape this "hell" and improve our code.

What is a Promise?

From MDN:

A **Promise** is an object representing the eventual completion or failure of an asynchronous operation.

Have a quick look at the following example:

```
const data = fetch('your-api-url-goes-here');
console.log('Finished');
console.log(data);
```

Create your own promise

```
const myPromise = new Promise((resolve,
reject) => {
  // your code goes here
});
```

This is how you create your own promise, `resolve` and `reject` will be called once the promise is finished.

We can immediately return it to see what we would get:

```
const myPromise = new Promise((resolve,
reject) => {
  resolve("The value we get from the
promise");
});

myPromise.then(
  data => {
    console.log(data);
  });
// The value we get from the promise
```

We immediately resolved our promise and can see the result in the console.

We can combine a `setTimeout()` to wait a certain amount of time before resolving.

```
 const myPromise = new Promise((resolve,
reject) => {
  setTimeout(() => {
      resolve("The value we get from the
promise");
    }, 2000);
});

myPromise.then(
  data => {
    console.log(data);
  });
// after 2 seconds ...
// The value we get from the promise
```

These two examples are very simple but **promises** are very useful when dealing with big requests of data.

In the example above we kept it simple and only resolved our promise but in reality you will also encounter errors so let's see how to deal with them:

```
 const myPromise = new Promise((resolve,
reject) => {
  setTimeout(() => {
```

```
      reject(Error("this is our error"));
    }, 2000);
});

myPromise
  .then(data => {
    console.log(data);
  })
  .catch(err => {
    console.error(err);
  })
  // Error: this is our error
  // Stack trace:
  // myPromise</<@debugger eval code:3:14
```

We use `.then()` to grab the value when the promise resolves and `.catch()` when the promise rejects.

Looking at our error log you can see that it tells us where the error occurred, that is because we wrote `reject(Error("this is our error"));` and not simply `reject("this is our error");`.

Chaining promises

We can chain promises one after the other, using what was returned from the previous one as the base

for the subsequent one, whether the promise was resolved or rejected.

You can chain as many promises as you want and the code will still be more readable and shorter than what we have seen above in the **callback hell**.

```
 const myPromise = new Promise((resolve,
reject) => {
  resolve();
});

myPromise
  .then(data => {
    // return something new
    return 'working...'
  })
  .then(data => {
    // log the data that we got from the
previous promise
    console.log(data);
    throw 'failed!';
  })
  .catch(err => {
    // grab the error from the previous
promise and log it
    console.error(err);
```

```
    // failed!
  })
```

As you can see, the first `then` passed a value down to the second one where we logged it and we also threw an error that was caught in the `catch` clause.

We are not limited to chaining in case of success, we can also chain when we get a `reject`.

```
 const myPromise = new Promise((resolve,
reject) => {
  resolve();
});

myPromise
  .then(data => {
    throw new Error("ooops");
    console.log("first value");
  })
  .catch(() => {
    console.log("catch an error");
  })
  .then(data => {
  console.log("second value");
  });
  // catch an error
  // second value
```

We did not get "first value" because we threw an error therefore we only got the first `.catch()` and the last `.then()`.

`Promise.resolve()` & `Promise.reject()`

`Promise.resolve()` and `Promise.reject()` will create promises that automatically resolve or reject.

```
 //Promise.resolve()
Promise.resolve('Success').then(function(va
lue) {
  console.log();
  // "Success"
}, function(value) {
  cnsole.log('fail')
});

// Promise.reject()
Promise.reject(new
Error('fail')).then(function() {
  // not called
}, function(error) {
  console.log(error);
  // Error: fail
});
```

As you can see in the first example, the Promise created in the `then` clause has two arguments, one function that get's called when the Promise resolves, and one that gets called when the Promise rejects. Since `Promise.resolve()` immediately resolves the promise, the first functions is being called.

The opposite happens in the second example, where we use `Promise.reject()` to immediately reject the Promise, therefore the second argument of the `then` clause gets called.

`Promise.all()` & `Promise.race()`

`Promise.all()` returns a single Promise that resolves when all promises have resolved.

Let's look at this example where we have two promises.

```
 const promise1 =  new
Promise((resolve,reject) => {
  setTimeout(resolve, 500, 'first value');
});
const promise2 =  new
Promise((resolve,reject) => {
  setTimeout(resolve, 1000, 'second
value');
});
```

```
promise1.then(data => {
  console.log(data);
});
// after 500 ms
// first value
promise2.then(data => {
  console.log(data);
});
// after 1000 ms
// second value
```

They will resolve independently from one another but look at what happens when we use `Promise.all()`.

```
 const promise1 =  new
Promise((resolve,reject) => {
  setTimeout(resolve, 500, 'first value');
});
const promise2 =  new
Promise((resolve,reject) => {
  setTimeout(resolve, 1000, 'second
value');
});

Promise
  .all([promise1, promise2])
```

```
  .then(data => {
    const[promise1data, promise2data] =
data;
    console.log(promise1data,
promise2data);
  });
// after 1000 ms
// first value second value
```

Our values returned together, after 1000ms (the timeout of the *second* promise) meaning that the first one had to wait for the completion of the second one.

If we were to pass an empty iterable then it will return an already resolved promise.

If one of the promises was rejected, all of them would asynchronously reject with the value of that rejection, even if they resolved.

```
 const promise1 =  new
Promise((resolve,reject) => {
  resolve("my first value");
});
const promise2 =  new
Promise((resolve,reject) => {
  reject(Error("oooops error"));
});
```

```
// one of the two promises will fail, but
`.all` will return only a rejection.
Promise
  .all([promise1, promise2])
  .then(data => {
    const[promise1data, promise2data] =
data;
    console.log(promise1data,
promise2data);
  })
  .catch(err => {
    console.log(err);
  });
  // Error: oooops error
```

`Promise.race()` on the other hand returns a promise that resolves or rejects as soon as one of the promises in the iterable resolves or rejects, with the value from that promise.

```
 const promise1 =  new
Promise((resolve,reject) => {
  setTimeout(resolve, 500, 'first value');
});
const promise2 =  new
Promise((resolve,reject) => {
```

```
  setTimeout(resolve, 100, 'second value');
});

Promise.race([promise1,
promise2]).then(function(value) {
  console.log(value);
  // Both resolve, but promise2 is faster
});
// expected output: "second value"
```

If we passed an empty iterable, the race would be pending forever!

End of Chapter 13 Quiz

13.1 What is a `Promise`?

- ☐ a new primitive
- ☐ is an object representing the eventual completion or failure of an asynchronous operation
- ☐ a new type of loop

13.2 Write the code to accomplish the following task:

Write a simple `Promise` that will immediately resolve and print something to the console.

13.3 Which of the following promise methods does not exist?

- ☐ `Promise.race()`
- ☐ `Promise.some()`
- ☐ `Promise.all()`
- ☐ `Promise.reject()`

13.4 What is the correct output of the following code:

```
function myPromise(){
    return new Promise((resolve,reject) =>
```

```
{
        reject();
    })
}

myPromise()
    .then(() => {
        console.log('1')
    })
    .then(() => {
        console.log('2')
    })
    .catch(() => {
        console.log('3')
    })
    .then(() => {
        console.log('4')
    });
```

- ☐ 1,2,3,4
- ☐ 3,4,1,2
- ☐ 3,4
- ☐ 4

Chapter 14: Generators

What is a Generator?

A generator function is a function that we can start and stop, for an indefinite amount of time. And, restart with the possibility of passing additional data at a later point in time.

To create a generator function we write like this:

```
 function* fruitList(){
   yield 'Banana';
   yield 'Apple';
   yield 'Orange';
}

const fruits = fruitList();

fruits;
// Generator
fruits.next();
// Object { value: "Banana", done: false }
fruits.next();
// Object { value: "Apple", done: false }
fruits.next();
// Object { value: "Orange", done: false }
```

```
fruits.next();
// Object { value: undefined, done: true }
```

Let's have a look at the code piece by piece:

- we declared the function using `function*`
- we used the keyword `yield` before our content
- we start our function using `.next()`
- the last time we call `.next()` we receive and empty object and we get `done: true`

Our function is paused between each `.next()` call.

Looping over an array with a generator

We can use the `for of` loop to iterate over our generator and `yield` the content at each loop.

```
 // create an array of fruits
const fruitList =
['Banana','Apple','Orange','Melon','Cherry'
,'Mango'];

// create our looping generator
function* loop(arr) {
  for (const item of arr) {
    yield `I like to eat ${item}s`;
```

```
  }
}


const fruitGenerator = loop(fruitList);
fruitGenerator.next();
// Object { value: "I like to eat Bananas",
done: false }
fruitGenerator.next();
// Object { value: "I like to eat Apples",
done: false }
fruitGenerator.next().value;
// "I like to eat Oranges"
```

- Our new generator will loop over the array and print one value at a time every time we call `.next()`
- If you are only concerned about getting the value, then use `.next().value` and it will not print the status of the generator

Finish the generator with .return()

Using `.return()` we can return a given value and finish the generator.

```
function* fruitList(){
  yield 'Banana';
```

```
  yield 'Apple';
  yield 'Orange';
}

const fruits = fruitList();

fruits.return();
// Object { value: undefined, done: true }
```

In this case we got `value: undefined` because we did not pass anything in the `return()`.

Catching errors with .throw()

```
function* gen(){
  try {
    yield "Trying...";
    yield "Trying harder...";
    yield "Trying even harder..";
  }
  catch(err) {
    console.log("Error: " + err );
  }
}

const myGenerator = gen();
```

```
myGenerator.next();
// Object { value: "Trying...", done: false
}
myGenerator.next();
// Object { value: "Trying harder...",
done: false }
myGenerator.throw("ooops");
// Error: ooops
// Object { value: undefined, done: true }
```

As you can see when we called `.throw()` the `generator` returned us the error and finished even though we still had one more `yield` to execute.

Combining Generators with Promises

As we have previously seen, Promises are very useful for asynchronous programming, and by combining them with generators we have a very powerful tool at our disposal to avoid problems like the *callback hell*.

As we are solely discussing ES6, I won't be talking about async functions as they were introduced in ES2017, but know that the way they work is based on what you will see now.

You can read more about async functions in Chapter 19.

Using a Generator in combination with a `Promise` will allow us to write asynchronous code that feels like synchronous code.

What we want to do is wait for a promise to resolve and then pass the resolved value back into our generator in the `.next()` call.

```
 const myPromise = () => new
Promise((resolve) => {
  resolve("our value is...");
});

function* gen() {
  let result = "";
  // returns promise
  yield myPromise().then(data => { result =
data }) ;
  // wait for the promise and use its value
  yield result + ' 2';
};

// Call the async function and pass params
const asyncFunc = gen();
const val1 = asyncFunc.next();
console.log(val1);
// call the promise and wait for it to
```

```
resolve
// {value: Promise, done: false}
val1.value.then(() => {
  console.log(asyncFunc.next());
})
// Object { value: "our value is... 2",
done: false }
```

The first time we call `.next()` it will call our promise and wait for it to resolve (in our simple example it resolves immediately) and when we call `.next()` again it will utilize the value returned by the promise to do something else (in this case just interpolate a string).

End of Chapter 14 Quiz

14.1 What is the correct syntax of a generator function?

- ☐ `generator function() {...}`
- ☐ `new generator(){...}`
- ☐ `function*(){...}`

14.2 What is the main feature of a generator?

- ☐ It can't be stopped
- ☐ It can generate other functions
- ☐ It can't be overwritten
- ☐ It can be stopped and restarted

14.3 What is the correct output of the following code?

```
function* fruitList(){
  yield 'Banana';
  yield 'Apple';
  yield 'Pomelo';
  yield 'Mangosteen';
  yield 'Orange';
}
```

```
const fruits = fruitList();


fruits;
fruits.next();
fruits.next();
fruits.next();
```

What is the last output of `fruits.next();`?

- ☐ `Object { value: "Banana", done: false }`
- ☐ `Object { value: "Pomelo", done: true }`
- ☐ `Object { value: "Mangosteen", done: false }`
- ☐ `Object { value: "Pomelo", done: false }`

14.4 What is the correct output of the following code?

```
 function* fruitList(){
  yield 'Banana';
  yield 'Apple';
  yield 'Orange';
}


const fruits = fruitList();
```

```
fruits.return();
```

- Object { value: "Banana", done: true }
- Object { value: undefined, done: false }
- Object { value: "Banana", done: false }
- Object { value: undefined, done: true }

Chapter 15: Proxies

What is a Proxy?

From MDN:

The **Proxy** object is used to define custom behavior for fundamental operations (e.g. property lookup, assignment, enumeration, function invocation, etc).

How to use a Proxy ?

This is how we create a Proxy:

```
var x = new Proxy(target,handler)
```

- our `target` can be anything, from an object, to a function, to another `Proxy`
- a `handler` is an object which will define the behavior of our `Proxy` when an operation is performed on it

```
// our object
const dog = { breed: "German Shephard",
age: 5}

// our Proxy
const dogProxy = new Proxy(dog, {
  get(target,breed){
    return target[breed].toUpperCase();
```

```
  },
  set(target, breed, value){
    console.log("changing breed to...");
    target[breed] = value;
  }
});

console.log(dogProxy.breed);
// "GERMAN SHEPHARD"
console.log(dogProxy.breed = "Labrador")
// changing breed to...
// "Labrador"
console.log(dogProxy.breed);
// "LABRADOR"
```

When we call the `get` method we step inside the normal flow and change the value of the breed to uppercase.

When setting a new value we step in again and log a short message before setting the value.

Proxies can be very useful. For example, we can use them to validate data.

```
const validateAge = {
  set: function(object,property,value){
    if(property === 'age'){
```

```
      if(value < 18){
        throw new Error('you are too
young!');
      } else {
        // default behaviour
        object[property] = value;
        return true
      }
    }
  }
}

const user =  new Proxy({},validateAge)

user.age = 17
// Uncaught Error: you are too young!

user.age = 21
// 21
```

When we set the `age` property of the `user Object` we pass it through our `validateAge` function which checks if it is more or less than 18 and throws an error if it's less than 18.

Proxies can be very useful if we have many properties that would require a **getter** and **setter** each. By using a

`Proxy` we need to define only one **getter** and one **setter**. Let's look at this example:

```
const dog  = {
  _name: 'pup',
  _age: 7,

  get name() {
    console.log(this._name)
  },
  get age(){
    console.log(this._age)
  },

  set name(newName){
    this._name = newName;
    console.log(this._name)
  },
  set age(newAge){
    this._age = newAge;
    console.log(this._age)
  }
}

dog.name;
// pup
```

```
dog.age;
// 7
dog.breed;
// undefined
dog.name = 'Max';
// Max
dog.age = 8;
// 8
```

Notice that i'm writing `_name` instead of `name` etc.., the _ symbol is used in `JavaScript` convention to define **Private** properties, meaning properties that are should not be accessed by instances of the same class. That is not something that `JavaScript` enforces, it's just for developers to quickly identify **Private** properties. The reason why i'm using it here is because if i was to call:

```
 set name(newName){
   this.name = newName;
}
```

This would cause an infinite loop as `this.name =` would call the setter again and again. By putting the underscore in front of it it, I achieve the same result that I would get by renaming the setter to something else, such as:

```
 set rename(newName){
   this.name = newName;
}
```

As you can see we had two properties: `name` and `age`and for each of them we had to create a **getter** and a **setter**.
When we try to access a property that does not exist on the Object, such as `breed` we get `undefined`.

With a `Proxy` we can simplify the code by writing the following:

```
 const dog  = {
   name: 'pup',
   age: 7
}
const handler = {
   get: (target, property) => {
     property in target ?
console.log(target[property]) :
console.log('property not found');
   },
   set: (target, property, value) => {
     target[property] = value;
     console.log(target[property])
   }
}
```

```
const dogProxy = new Proxy(dog, handler);

dogProxy.name;
// pup
dogProxy.age;
// 7
dogProxy.name = 'Max';
// Max
dogProxy.age = 8;
// 8
dogProxy.breed;
// property not found
```

First, we created our `dog` Object but this time we did not set any **getter** and **setter** inside of it.
We created our `handler` that will handle each possible property with only one **getter** and **setter**.
In the **getter** what we are doing is check if the `property` is available on the `target` Object. If it is, we log it, otherwise we log a custom message.
The setter takes three argument, the `target` object, the `property` name and the `value`, nothing special happens here, we set the property to the new value and we log it.

As you can see, by using a `Proxy` we achieved two things:

- shorter, cleaner code
- we are logging a custom message if we try to access a property that is not available.

End of Chapter 15 Quiz

15.1 What is the use of a `Proxy`?

- ☐ to store unique values
- ☐ to define custom behavior of fundamental operations
- ☐ to make a value inaccessible from other functions

15.2 How many parameters can a `Proxy` take?

- ☐ 1
- ☐ 2
- ☐ 3
- ☐ 4

15.3 Which of the following is correct:

- ☐ The target parameter of a Proxy must be an Array
- ☐ The target parameter of a Proxy must not be an Array
- ☐ The target parameter of a Proxy can be another Proxy
- ☐ The target parameter of a Proxy cannot be another Proxy

Chapter 16: Sets, WeakSets, Maps and WeakMaps

What is a Set?

A `Set` is an `Object` where we can store **unique values** of any type.

```
 // create our set
const family = new Set();

// add values to it
family.add("Dad");
console.log(family);
// Set [ "Dad" ]

family.add("Mom");
console.log(family);
// Set [ "Dad", "Mom" ]

family.add("Son");
console.log(family);
// Set [ "Dad", "Mom", "Son" ]

family.add("Dad");
console.log(family);
// Set [ "Dad", "Mom", "Son" ]
```

As you can see, at the end we tried to add "Dad" again, but the `Set` still remained the same because a `Set` can only take **unique values**.

Let's continue using the same `Set` and see what methods we can use on it.

```
 family.size;
// 3
family.keys();
// SetIterator {"Dad", "Mom", "Son"}
family.entries();
// SetIterator {"Dad", "Mom", "Son"}
family.values();
// SetIterator {"Dad", "Mom", "Son"}
family.delete("Dad");
family;
// Set [ "Mom", "Son" ]
family.clear();
family;
// Set []
```

As you can see a `Set` has a `size` property and we can `delete` an item from it or use `clear` to delete all the items from it.

We can also notice that a `Set` does not have keys, so when we call `.keys()` we get the same result as calling `.values()` or `.entries()`.

Loop over a Set

We have two ways of iterating over a `Set`: using `.next()` or using a `for of` loop.

```
 // using `.next()`
const iterator = family.values();
iterator.next();
// Object { value: "Dad", done: false }
iterator.next();
// Object { value: "Mom", done: false }


// using a `for of` loop
for(const person of family) {
  console.log(person);
}
// Dad
// Mom
// Son
```

The method `values()` will return an `Iterator` object on which we can call `next()` similarly to how we did when we discussed about `generator` functions.

Remove duplicates from an array

We can use a `Set` to remove duplicates from an Array since we know it can only hold unique values.

```
 const myArray = ["dad", "mom", "son",
"dad", "mom", "daughter"];

const set = new Set(myArray);
console.log(set);
// Set [ "dad", "mom", "son", "daughter" ]
// transform the `Set` into an Array
const uniqueArray = Array.from(set);
console.log(uniqueArray);
// Array [ "dad", "mom", "son",
"daughter" ]

// write the same but in a single line
const uniqueArray = Array.from(new
Set(myArray));
// Array [ "dad", "mom", "son",
"daughter" ]
```

As you can see the new array only contains the unique values from the original array.

What is a WeakSet?

A `WeakSet` is similar to a `Set` but it can **only** contain Objects.

```
 let dad = {name: "Daddy", age: 50};
let mom = {name: "Mummy", age: 45};

const family = new WeakSet([dad,mom]);

for(const person of family){
  console.log(person);
}
// TypeError: family is not iterable
```

We created our new `WeakSet` but when we tried to use a `for of` loop it did not work, we can't iterate over a `WeakSet`.

A `WeakSet` cleans itself up after we delete an element from it.

```
 let dad = {name: "Daddy", age: 50};
let mom = {name: "Mummy", age: 45};

const family = new WeakSet([dad,mom]);

dad = null;
console.log(family);
```

```
// WeakSet [ {…}, {…} ]

// wait a few seconds
console.log(family);
// WeakSet [ {…} ]
```

You can try running the example above in the Dev Tools of your browser, as you can see after a few seconds **dad** was removed and *garbage collected*. That happened because the reference to it was lost when we set the value to `null`.

What is a Map?

A `Map` is similar to a `Set`, but they have key/value pairs.

```
const family = new Map();

family.set("Dad", 40);
family.set("Mom", 50);
family.set("Son", 20);

family;
// Map { Dad → 40, Mom → 50, Son → 20 }
family.size;
// 3
```

```
family.forEach((val,key) =>
console.log(key,val));
// Dad 40
// Mom 50
// Son 20

for(const [key,val] of family){
  console.log(key,val);
}
// Dad 40
// Mom 50
// Son 20
```

If you remember, we could iterate over a `Set` only with a `for of` loop, while we can iterate over a `Map` with both a `for of` and a `forEach` loop.

What is a WeakMap?

A `WeakMap` is a collection of key/value pairs and similarly to a `WeakSet`, even in a `WeakMap` the keys are *weakly* referenced, which means that when the reference is lost, the value will be removed from the `WeakMap` and *garbage collected*.

A `WeakMap` is **not** enumerable therefore we cannot loop over it.

```
 let dad = { name: "Daddy" };
let mom = { name: "Mommy" };

const myMap = new Map();
const myWeakMap = new WeakMap();

myMap.set(dad);
myWeakMap.set(mom);

dad = null;
mom = null;

console.log(myMap);
// Map(1) {{…}}
console.log(myWeakMap);
// WeakMap {}
```

As you can see *mom* was garbage collected after we set the its value to `null` whilst *dad* still remains inside our `Map`.

End of Chapter 16 Quiz

16.1 Which one of these does not exist?

- [ ] `Set`
- [ ] `WeakSet`
- [ ] `StrongSet`
- [ ] `WeakMap`

16.2 Which one of the following definition is correct?

- [ ] `A Set can only store objects`
- [ ] `A WeakSet can only store objects`
- [ ] `A WeakSet can be overwritten`
- [ ] `Both Set and WeakSet can only store objects`

16.3 Which one of the following definition is correct?

- [ ] `a Map only stores keys`
- [ ] `a Set stores both keys and values, a Map only values`
- [ ] `a Map stores both keys and values, a Set only values`
- [ ] `none, they both stores only values`

Chapter 17: Everything new in ES2016

ES2016 introduced only two new features:

- `Array.prototype.includes()`
- The exponential operator

`Array.prototype.includes()`

The `includes()` method will return `true` if our array includes a certain element, or `false` if it doesn't.

```
let array = [1,2,4,5];


array.includes(2);
// true
array.includes(3);
// false
```

Combine includes() with fromIndex

We can provide `.includes()` with an index to begin searching for an element. Default is 0, but we can also pass a negative value.

The first value we pass in is the element to search and the second one is the index:

```
 let array = [1,3,5,7,9,11];


array.includes(3,1);
// find the number 3 starting from array
index 1
// true
array.includes(5,4);
//false
array.includes(1,-1);
// find the number 1 starting from the
ending of the array going backwards
// false
array.includes(11,-3);
// true
```

`array.includes(5,4);` returned `false` because, despite the array actually containing the number 5, it is found at the index 2 but we started looking at position 4. That's why we couldn't find it and it returned `false`.

`array.includes(1,-1);` returned `false` because we started looking at the index -1 (which is the last element of the array) and then continued from that point onward.

`array.includes(11,-3);` returned `true` because we went back to the index -3 and moved up, finding the value 11 on our path.

The exponential operator

Prior to ES2016 we would have done the following:

```
Math.pow(2,2);
// 4
Math.pow(2,3);
// 8
```

Now with the new exponential operator, we can do the following:

```
2**2;
// 4
2**3;
// 8
```

It will get pretty useful when combining multiple operations like in this example:

```
2**2**2;
// 16
Math.pow(Math.pow(2,2),2);
// 16
```

Using `Math.pow()` you need to continuously concatenate them and it can get pretty long and messy. The exponential operator provides a faster and cleaner way of doing the same thing.

End of Chapter 17 Quiz

17.1 What new array method was introduced in ES2016?

- ☐ `Array.prototype.contains()`
- ☐ `Array.prototype.has()`
- ☐ `Array.prototype.includes()`
- ☐ `Array.prototype.find()`

17.2 What is the correct output of the following code?

```
let array = [1,3,5,7,9,11];


array.includes(5,4);
```

- ☐ 4
- ☐ true
- ☐ 5
- ☐ false

17.3 Write the code code to accomplish the following task:

Refactor the following code to utilize the new exponential operator.

```
Math.pow(Math.pow(2,2),2);
// 16
```

Chapter 18: ES2017 string padding, Object.entries(), Object.values() and more

ES2017 introduced many cool new features, which we are going to see here.

String padding (.padStart() and .padEnd())

We can now add some padding to our strings, either at the end (`.padEnd()`) or at the beginning (`.padStart()`) of them.

```
"hello".padStart(6);
// " hello"
"hello".padEnd(6);
// "hello "
```

We specified that we want 6 as our padding, so then why in both cases did we only get 1 space?
It happened because `padStart` and `padEnd` will go and fill the **empty spaces**. In our example "hello" is 5 letters, and our padding is 6, which leaves only 1 empty space.

Look at this example:

```
"hi".padStart(10);
// 10 - 2 = 8 empty spaces
```

```
// "        hi"
"welcome".padStart(10);
// 10 - 6 = 4 empty spaces
// "    welcome"
```

Right align with padStart

We can use `padStart` if we want to right align something.

```
 const strings = ["short", "medium length",
"very long string"];

const longestString = strings.sort(str =>
str.length).map(str => str.length)[0];

strings.forEach(str =>
console.log(str.padStart(longestString)));

// very long string
//    medium length
//             short
```

First we grabbed the longest of our strings and measured its length. We then applied a `padStart` to all the strings based on the length of the longest so

that we now have all of them perfectly aligned to the right.

Add a custom value to the padding

We are not bound to just add a white space as a padding, we can pass both strings and numbers.

```
 "hello".padEnd(13," Alberto");
// "hello Alberto"
"1".padStart(3,0);
// "001"
"99".padStart(3,0);
// "099"
```

Object.entries() and Object.values()

Let's first create an Object.

```
 const family = {
   father: "Jonathan Kent",
   mother: "Martha Kent",
   son: "Clark Kent",
}
```

In previous versions of `JavaScript` we would have accessed the values inside the object like this:

```
Object.keys(family);
// ["father", "mother", "son"]
family.father;
"Jonathan Kent"
```

`Object.keys()` returned only the keys of the object that we then had to use to access the values.

We now have two more ways of accessing our objects:

```
Object.values(family);
// ["Jonathan Kent", "Martha Kent", "Clark
Kent"]

Object.entries(family);
// ["father", "Jonathan Kent"]
// ["mother", "Martha Kent"]
// ["son", "Clark Kent"]
```

`Object.values()` returns an array of all the values whilst `Object.entries()` returns an array of arrays containing both keys and values.

`Object.getOwnPropertyDescriptors()`

This method will return all the own property descriptors of an object.
The attributes it can return are `value`, `writable`, `get`, `set`, `configurable` and `enumerable`.

```
const myObj = {
  name: "Alberto",
  age: 25,
  greet() {
    console.log("hello");
  },
}
Object.getOwnPropertyDescriptors(myObj);
// age:{value: 25, writable: true,
enumerable: true, configurable: true}

// greet:{value: ƒ, writable: true,
enumerable: true, configurable: true}

// name:{value: "Alberto", writable: true,
enumerable: true, configurable: true}
```

Trailing commas

This is just a minor change to a syntax. Now, when writing objects we can leave a trailing comma after each parameter, whether or not it is the last one.

```
// from this
const object = {
  prop1: "prop",
```

```
  prop2: "propop"
}

// to this
const object = {
  prop1: "prop",
  prop2: "propop",
}
```

Notice how I wrote a comma at the end of the second property.
It will not throw any error if you don't put it, but it's a better practice to follow as it make your colleague's or team member's life easier.

```
 // I write
const object = {
  prop1: "prop",
  prop2: "propop"
}

// my colleague updates the code, adding a new property
const object = {
  prop1: "prop",
  prop2: "propop"
  prop3: "propopop"
```

```
}
// suddenly, he gets an error because he
did not notice that I forgot to leave a
comma at the end of the last parameter.
```

Shared memory and Atomics

From MDN:

When memory is shared, multiple threads can read and write the same data in memory. **Atomic** operations make sure that predictable values are written and read, that operations are finished before the next operation starts and that operations are not interrupted.

`Atomics` is not a constructor, all of its properties and methods are static (just like `Math`) therefore we cannot use it with a new operator or invoke the `Atomics` object as a function.

Examples of its methods are:

- add / sub
- and / or / xor
- load / store

Atomics are used with `SharedArrayBuffer` (generic fixed-length binary data buffer) objects which represent generic, fixed-length raw binary data buffer.

Let's have a look at some examples of `Atomics` methods:

`Atomics.add()`, `Atomics.sub()`, `Atomics.load()` and `Atomics.store()`

`Atomics.add()` will take three arguments, an array, an index and a value and will return the previous value at that index before performing an addition.

```
 // create a `SharedArrayBuffer`
const buffer = new SharedArrayBuffer(16);
const uint8 = new Uint8Array(buffer);

// add a value at the first position
uint8[0] = 10;

console.log(Atomics.add(uint8, 0, 5));
// 10

// 10 + 5 = 15
console.log(uint8[0])
// 15
console.log(Atomics.load(uint8,0));
// 15
```

As you can see, calling `Atomics.add()` will return the previous value at the array position we are targeting. when we call again `uint8[0]` we see that the addition was performed and we got 15.

To retrieve a specific value from our array we can use `Atomics.load` and pass two argument, an array and an index.

`Atomics.sub()` works the same way as `Atomics.add()` but it will subtract a value.

```
 // create a `SharedArrayBuffer`
const buffer = new SharedArrayBuffer(16);
const uint8 = new Uint8Array(buffer);

// add a value at the first position
uint8[0] = 10;

console.log(Atomics.sub(uint8, 0, 5));
// 10

// 10 - 5 = 5
console.log(uint8[0])
// 5
console.log(Atomics.store(uint8,0,3));
// 3
```

```
console.log(Atomics.load(uint8,0));
// 3
```

Here we are using `Atomics.sub()` to substract 5 from the value at position `uint8[0]` which is equivalent to 10 - 5.
Same as with `Atomics.add()`, the method will return the previous value at that index, in this case 10.

We are then using `Atomics.store()` to store a specific value, in this case 3, at a specific index of the array, in this case 0, the first position.
`Atomics.store()` will return the value that we just passed, in this case 3. You can see that when we call `Atomics.load()` on that specific index we get 3 and not 5 anymore.

`Atomics.and()`, `Atomics.or()` and `Atomics.xor()`

These three methods all perform bitwise AND, OR and XOR operations at a given position of the array. You can read more about bitwise operations on Wikipedia at this link https://en.wikipedia.org/wiki/Bitwise_operation

End of Chapter 18 Quiz

18.1 What is the correct output of the following code?

```
"hello".padStart(6);
```

- ☐ " hello"
- ☐ "hello "
- ☐ " hello"
- ☐ "hello"

18.2 Write the code to accomplish the following task:

Use `padStart` to right align all three of the following strings.

```
const strings = ["short", "medium length",
"very long string"];


//expected output
//              short
//    medium length
// very long string
```

18.3 Which of the following was not added in ES2016?

- ☐ Object.entries()
- ☐ Object.keys()
- ☐ Object.values()

18.4 What is the correct output of the following code:

```
const buffer = new SharedArrayBuffer(16);
const uint8 = new Uint8Array(buffer);

uint8[0] = 10;

console.log(Atomics.add(uint8, 0, 5));
```

- ☐ 0
- ☐ 10
- ☐ 15
- ☐ 5

18.5 What is the correct output of the following code:

```
const buffer = new SharedArrayBuffer(16);
const uint8 = new Uint8Array(buffer);
```

```
uint8[0] = 10;

Atomics.sub(uint8, 0, 6);

console.log(Atomics.load(uint8,0));
```

- [ ] 0
- [ ] 10
- [ ] 6
- [ ] 4

Chapter 19: ES2017 Async and Await

ES2017 introduced a new way of working with promises, called "async/await".

Promise review

Before we dive in this new syntax let's quickly review how we would usually write a promise:

```
 // fetch a user from github
fetch('api.github.com/user/
AlbertoMontalesi').then( res => {
  // return the data in json format
  return res.json();
}).then(res => {
  // if everything went well, print the
data
  console.log(res);
}).catch( err => {
  // or print the error
  console.log(err);
})
```

This is a very simple promise to fetch a user from GitHub and print it to the console.

Let's see a different example:

```
 function walk(amount) {
  return new Promise((resolve,reject) => {
    if (amount < 500) {
      reject ("the value is too small");
    }
    setTimeout(() => resolve(`you walked
for ${amount}ms`),amount);
  });
}

walk(1000).then(res => {
  console.log(res);
  return walk(500);
}).then(res => {
  console.log(res);
  return walk(700);
}).then(res => {
  console.log(res);
  return walk(800);
}).then(res => {
  console.log(res);
  return walk(100);
}).then(res => {
  console.log(res);
```

```
  return walk(400);
}).then(res => {
  console.log(res);
  return walk(600);
});

// you walked for 1000ms
// you walked for 500ms
// you walked for 700ms
// you walked for 800ms
// uncaught exception: the value is too
small
```

Let's see how we can rewrite this `Promise` with the new async/await syntax.

Async and Await

```
function walk(amount) {
  return new Promise((resolve,reject) => {
    if (amount < 500) {
      reject ("the value is too small");
    }
    setTimeout(() => resolve(`you walked
for ${amount}ms`),amount);
  });
```

```
}

// create an async function
async function go() {
  // use the keyword `await` to wait for
the response
  const res = await walk(500);
  console.log(res);
  const res2 = await walk(900);
  console.log(res2);
  const res3 = await walk(600);
  console.log(res3);
  const res4 = await walk(700);
  console.log(res4);
  const res5 = await walk(400);
  console.log(res5);
  console.log("finished");
}

go();

// you walked for 500ms
// you walked for 900ms
// you walked for 600ms
// you walked for 700ms
```

```
// uncaught exception: the value is too
small
```

Let's break down what we just did:

- to create an `async` function we need to put the `async` keyword in front of it
- the keyword will tell `JavaScript` to always return a promise
- if we specify to `return <non-promise>` it will return a value wrapped inside a promise
- the `await` keyword **only** works inside an `async` function
- as the name implies, `await` will tell `JavaScript` to wait until the promise returns its result

Let's see what happens if we try to use `await` outside an `async` function

```
 // use await inside a normal function
function func() {
  let promise = Promise.resolve(1);
  let result = await promise;
}
func();
// SyntaxError: await is only valid in
async functions and async generators
```

```
// use await in the top-level code
let response = Promise.resolve("hi");
let result = await response;
// SyntaxError: await is only valid in
async functions and async generators
```

Remember: You can only use `await` inside an `async` function.

Error handling

In a normal promise we would use `.catch()` to catch eventual errors returned by the promise.
Here, it is not much different:

```
 async function asyncFunc() {

  try {
    let response = await fetch('http:your-
url');
    } catch(err) {
        console.log(err);
      }
}
```

```
asyncFunc();
// TypeError: failed to fetch
```

We use `try...catch` to grab the error, but in a case where we do not have them we can still catch the error like this:

```
 async function asyncFunc(){
  let response = await fetch('http:your-
url');
}
asyncFunc();
// Uncaught (in promise) TypeError: Failed
to fetch

asyncFunc().catch(console.log);
// TypeError: Failed to fetch
```

End of Chapter 19 Quiz

19.1 What is the correct output of the following code?

```
function func() {
  let promise = Promise.resolve(1);
  let result = await promise;
}
func();
```

- ☐ 1
- ☐ true
- ☐ undefined
- ☐ SyntaxError

19.2 What is the last output of the following code:

```
function walk(amount) {
  return new Promise((resolve,reject) => {
    if (amount > 500) {
      reject ("the value is too big");
    }
    setTimeout(() => resolve(`you walked
for ${amount}ms`),amount);
  });
```

```
}

async function go() {
  const res = await walk(500);
  console.log(res);
  const res2 = await walk(300);
  console.log(res2);
  const res3 = await walk(200);
  console.log(res3);
  const res4 = await walk(700);
  console.log(res4);
  const res5 = await walk(400);
  console.log(res5);
  console.log("finished");
}

go();
```

- ☐ "you walked for 700ms"
- ☐ "you walked for 400ms"
- ☐ uncaught exception: the value is too big
- ☐ "finished"

Chapter 20: ES2018 Async Iteration and more?

In this chapter we will look at what was introduced with ES2018.

Rest / Spread for Objects

Remember how ES6 (ES2015) allowed us to do this?

```
 const veggie =
["tomato","cucumber","beans"];
const meat = ["pork","beef","chicken"];


const menu = [...veggie, "pasta", ...meat];
console.log(menu);
// Array [ "tomato", "cucumber", "beans",
"pasta", "pork", "beef", "chicken" ]
```

Now we can use the rest/spread syntax for objects too, let's look at how:

```
let myObj = {
  a:1,
  b:3,
  c:5,
  d:8,
```

```
}

// we use the rest operator to grab
everything else left in the object.
let { a, b, ...z } = myObj;
console.log(a);      // 1
console.log(b);      // 3
console.log(z);      // {c: 5, d: 8}

// using the spread syntax we cloned our
Object
let clone = { ...myObj };
console.log(clone);
// {a: 1, b: 3, c: 5, d: 8}
myObj.e = 15;
console.log(clone)
// {a: 1, b: 3, c: 5, d: 8}
console.log(myObj)
// {a: 1, b: 3, c: 5, d: 8, e: 15}
```

With the spread operator we can easily create a clone of our `Object` so that when we modify the original `Object`, the clone does not get modified, similarly to what we saw when we talked about arrays.

Asynchronous Iteration

With Asynchronous Iteration we can iterate asynchronously over our data.

From the documentation:

An async iterator is much like an iterator, except that its `next()` method returns a promise for a `{ value, done }` pair.

To do so, we will use a `for-await-of` loop which works by converting our iterables to a Promise, unless they already are one.

```
 const iterables = [1,2,3];


async function test() {
    for await (const value of iterables) {
        console.log(value);
    }
}
test();
// 1
// 2
// 3
```

During execution, an async iterator is created from the data source using the `[Symbol.asyncIterator]()` method.

Each time we access the next value in the sequence, we implicitly await the promise returned from the iterator method.

Promise.prototype.finally()

After our promise has finished we can invoke a callback.

```
 const myPromise = new
Promise((resolve,reject) => {
  resolve();
})
myPromise
  .then( () => {
    console.log('still working');
  })
  .catch( () => {
    console.log('there was an error');
  })
  .finally(()=> {
    console.log('Done!');
  })
```

`.finally()` will also return a `Promise` so we can chain more `then` and `catch` after it but those

Promises will fulfill based on the `Promise` they were chained onto.

```
 const myPromise = new
Promise((resolve,reject) => {
  resolve();
})
myPromise
.then( () => {
    console.log('still working');
    return 'still working';
  })
  .finally(()=> {
    console.log('Done!');
    return 'Done!';
  })
  .then( res => {
    console.log(res);
  })
// still working
// Done!
// still working
```

As you can see the `then` chained after `finally` returned the value that was returned by the `Promise` created not by `finally` but by the first `then`.

RegExp features

Four new RegExp related features made it to the new version of `ECMAScript`. They are:

- `s(dotAll)` flag for regular expressions
- RegExp named capture groups
- RegExp Lookbehind Assertions
- RegExp Unicode Property Escapes

`s (dotAll)` flag for regular expression

This introduces a new `s` flag for `ECMAScript` regular expressions that makes `.` match any character, including line terminators.

```
/foo.bar/s.test('foo\nbar');
// true
```

RegExp named capture groups

From the documentation:

Numbered capture groups allow one to refer to certain portions of a string that a regular expression matches. Each capture group is assigned a unique number and can be referenced using that number, but this can make a regular expression hard to grasp and

refactor. For example, given `/(\d{4})-(\d{2})-(\d{2})/` that matches a date, one cannot be sure which group corresponds to the month and which one is the day without examining the surrounding code. Also, if one wants to swap the order of the month and the day, the group references should also be updated. A capture group can be given a name using the `(?<name>...)` syntax, for any identifier `name`. The regular expression for a date then can be written as `/(?<year>\d{4})-(?<month>\d{2})-(?<day>\d{2})/u`. Each name should be unique and follow the grammar for `ECMAScript` IdentifierName. Named groups can be accessed from properties of a `groups` property of the regular expression result. Numbered references to the groups are also created, just as for non-named groups. For example:

```
let re = /(?<year>\d{4})-(?<month>\d{2})-(?<day>\d{2})/u;
let result = re.exec('2015-01-02');
// result.groups.year === '2015';
// result.groups.month === '01';
// result.groups.day === '02';

// result[0] === '2015-01-02';
// result[1] === '2015';
// result[2] === '01';
// result[3] === '02';
```

```
let {groups: {one, two}} = /^(?<one>.*):(?
<two>.*)$/u.exec('foo:bar');
console.log(`one: ${one}, two: ${two}`);
// one: foo, two: bar
```

RegExp Lookbehind Assertions

From the documentation:

With lookbehind assertions, one can make sure that a pattern is or isn't preceded by another, e.g. matching a dollar amount without capturing the dollar sign. Positive lookbehind assertions are denoted as `(?<=...)` and they ensure that the pattern contained within precedes the pattern following the assertion. For example, if one wants to match a dollar amount without capturing the dollar sign, `/(?<=\$)\d+(\.\d*)?/` can be used, matching `'$10.53'` and returning `'10.53'`. This, however, wouldn't match `€10.53`. Negative lookbehind assertions are denoted as `(?<!...)`and, on the other hand, make sure that the pattern within doesn't precede the pattern following the assertion. For example, `/(?<!\$)\d+(?:\.\d*)/` wouldn't match `'$10.53'`, but would `'€10.53'`.

RegExp Unicode Property Escapes

From the documentation:

This brings the addition of Unicode property escapes of the form `\p{…}` and`\P{…}`. Unicode property escapes are a new type of escape sequence available in regular expressions that have the `u` flag set. With this feature, we could write:

```
const regexGreekSymbol = /
\p{Script=Greek}/u;
regexGreekSymbol.test('π');
// true
```

Lifting template literals restriction

When using *tagged* template literals the restriction on escape sequences are removed.

You can read more here(https://tc39.github.io/proposal-template-literal-revision/#sec-template-literals).

End of Chapter 20 Quiz

20.1 What is the correct syntax for the spread operator for objects?

- ☐ `[...]`
- ☐ `(...)`
- ☐ `{...}`
- ☐ `=>`

20.2 What is the correct output of the following code:

```
let myObj = {
  a:1,
  b:2,
  c:3,
  d:4,
}

let { a, b, ...z } = myObj;
console.log(z);
```

- ☐ `[3,4]`
- ☐ `{c:3, d:4}`
- ☐ `undefined`
- ☐ `[c,d]`

20.3 What is the correct output of the following code:

```
 const myPromise = new
Promise((resolve,reject) => {
  resolve();
})
myPromise
.then( () => {
    return '1';
  })
  .finally(()=> {
    return '2!';
  })
  .then( res => {
    console.log(res);
  })
```

- ☐ 1
- ☐ 2
- ☐ 1,2

Chapter 21: What's new in ES2019?

In this chapter we will look at what is included in the latest version of `ECMAScript`: ES2019.

`Array.prototype.flat()` / `Array.prototype.flatMap()`

`Array.prototype.flat()` will flatten the array recursively up to the depth that we specify. If no depth argument is specified, 1 is the default value. We can use `Infinity` to flatten all nested arrays.

```
 const letters = ['a', 'b', ['c', 'd',
['e', 'f']]];
// default depth of 1
letters.flat();
// ['a', 'b', 'c', 'd', ['e', 'f']]

// depth of 2
letters.flat(2);
// ['a', 'b', 'c', 'd', 'e', 'f']

// which is the same as executing flat with
depth of 1 twice
letters.flat().flat();
```

```
// ['a', 'b', 'c', 'd', 'e', 'f']

// Flattens recursively until the array
contains no nested arrays
letters.flat(Infinity)
// ['a', 'b', 'c', 'd', 'e', 'f']
```

`Array.prototype.flatMap()` is identical to the previous one with regards to the way it handles the 'depth' argument but instead of simply flattening an array, with `flatMap()` we can also map over it and return the result in the new array.

```
 let greeting = ["Greetings from", " ",
"Vietnam"];

// let's first try using a normal `map()`
function
greeting.map(x => x.split(" "));
// ["Greetings", "from"]
// ["", ""]
// ["Vietnam"]



greeting.flatMap(x => x.split(" "))
// ["Greetings", "from", "", "", "Vietnam"]
```

As you can see, if we use `.map()` we will get a multi level array, a problem that we can solve by using `.flatMap()` which will also flatten our array.

`Object.fromEntries()`

`Object.fromEntries()` transforms a list of key-value pairs into an object.

```
const keyValueArray = [
  ['key1', 'value1'],
  ['key2', 'value2']
]

const obj =
Object.fromEntries(keyValueArray)
// {key1: "value1", key2: "value2"}
```

We can pass any iterable as argument of `Object.fromEntries()`, whether it's an `Array`, a `Map` or other objects implementing the iterable protocol.

You can read more about the iterable protocol here: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration*protocols#The*iterable_protocol

`String.prototype.trimStart()` / `.trimEnd()`

`String.prototype.trimStart()` removes white space from the beginning of a string while `String.prototype.trimEnd()` removes them from the end.

```
 let str = "    this string has a lot of
whitespace    ";

str.length;
// 42

str = str.trimStart();
// "this string has a lot of whitespace    "
str.length;
// 38

str = str.trimEnd();
// "this string has a lot of whitespace"
str.length;
// 35
```

We can also use `.trimLeft()` as an alias of `.trimStart()` and `.trimRight()` as an alias of `.trimEnd()`.

Optional Catch Binding

Prior to ES2019 you had to always include an exception variable in your `catch` clause. E2019 allows you to omit it.

```
 // Before
try {
    ...
} catch(error) {
    ...
}

// ES2019
try {
    ...
} catch {
    ...
}
```

This is useful when you want to ignore the error. For a more detailed list of use cases for this I highly recommend this article: http://2ality.com/2017/08/optional-catch-binding.html

Function.prototype.toString()

The `.toString()` method returns a string representing the source code of the function.

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

console.log(sum.toString());
// function sum(a, b) {
//     return a + b;
//   }
```

It also includes comments.

```
function sum(a, b) {
  // perform a sum
  return a + b;
}

console.log(sum.toString());
// function sum(a, b) {
//   // perform a sum
//   return a + b;
// }
```

`Symbol.prototype.description`

`.description` returns the optional description of a `Symbol` Object.

```
 const me = Symbol("Alberto");
me.description;
// "Alberto"


me.toString()
//  "Symbol(Alberto)"
```

End of Chapter 21 Quiz

21.1 Transform the following multidimensional array into a single dimensional array in one line of code using the new Array method introduced in ES2019

```
const letters = ['a', 'b', ['c', 'd',
['e', 'f']]];
// expected : ['a', 'b', 'c', 'd', 'e',
'f']
```

21.2 Starting from the given array, create an Object using ES2019 new features

```
const keyValueArray = [
  ['key1', 'value1'],
  ['key2', 'value2']
]

// expected:  {key1: "value1", key2:
"value2"}
```

21.3 What is the correct output of the following code?

```
const me = Symbol("Alberto");
console.log(me.description);
```

- ☐ Symbol
- ☐ Symbol(Alberto)
- ☐ "Alberto"
- ☐ undefined

21.4 What is the correct output of the following code?

```
const letters = ['a', 'b', ['c', 'd',
['e', 'f']]];
console.log(letters.flat())
```

- ☐ ['a', 'b', ['c', 'd', ['e', 'f']]]
- ☐ ['a', 'b', 'c', 'd', 'e', 'f']
- ☐ ['a', 'b', 'c', 'd', ['e', 'f']]
- ☐ ['a', 'b', ['c', 'd', 'e', 'f']]

21.5 What is the correct output of the following code?

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


console.log(sum.toString());
```

- ☐ Function
- ☐ "a + b"

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

An Intro To TypeScript

Now that you have a clear idea of how `JavaScript` looks like in 2019, I think it's a good moment to introduce you `TypeScript`.

Albeit not necessary as a skill for a `JavaScript` developer, I believe it to be extremely useful, especially when working in team on bigger projects.

As you already know, `JavaScript` is not a **strongly typed** language meaning that you do not have to define the type of your variables upon declaration.

That means that they can be more flexible and accept different values but at the same time it can make the code more confusing and prone to have bugs.

Look at this example:

```
function getUserByID(userID){
  // perform an api call to your server to retrieve a user by its id
}
```

What is `userID` ? Is it an `integer` or a `string`? We can assume it's an integer, but maybe it's an alphanumeric string (e.g.: 'A123').

Unless we wrote that piece of code, we have no way of knowing what is the type of the argument.

That is where `TypeScript` comes to help. This is how the same code would look like:

```
function getUserByID(userID:number){
  // perform an api call to your server to retrieve a user by its id
}
```

Perfect! Now we know for sure that if we pass a string to the function that will cause an error.

A simple addition made the code easier to use.

What is TypeScript?

Created just a few year ago by Microsoft, `TypeScript` is a typed superset of `JavaScript` that compiles to plain `JavaScript`.

Being a superset means that you can write plain `JavaScript` in a `TypeScript` file and that will cause no error.

Browsers do not understand `TypeScript`, which means it has to be transpiled to plain `JavaScript`.

We will now look at how to set up our environment to write `TypeScript`.

How to use TypeScript

Getting started with `TypeScript` takes literally 5 minutes.

The first thing to do is to install it. Run this command in your terminal:

`npm install -g` TypeScript

Next you can open your code editor and create a file called `greeter.ts` (NOT `.js`).

```
const greeter = (name:string) => {
    console.log(`hello ${name}`)
}
greeter('Alberto');
// hello Alberto
```

Now what we have to do to generate the `JavaScript` file is run this command in the terminal in the same directory as our file:

`tsc greeter.ts`

Now you should have a `greeter.js` file with this code inside:

```
var greeter = function (name) {
    console.log("hello " + name);
};
```

```
greeter('Alberto');
// hello Alberto
```

Where did `name:string` go? As we know `JavaScript` is not strongly typed therefore the type declaration get removed when the code is transpiled.

Typing your code will help *YOU* to debug it and to cause less errors but it won't create a different `JavaScript` output than what you would have gotten if you didn't use types.

TypeScript basic types

In this section we will go over the basic types supported by `TypeScript`. You should already know the meaning of most of them from the Introduction to `JavaScript` section.

The basic types supported by `TypeScript` are:

- `boolean`
- `number`
- `string`
- `Array`
- `object`
- `Tuple`
- `enum`

- `any`
- `void`
- `null` and `undefined`
- `never`

boolean

Defines a value that can be true or false:

```
const active: boolean = true;
```

number

The type `number` supports hexadecimal, decimal, binary and octal literals:

```
const decimal: number = 9;
const hex: number = 0xf00d;
const binary: number = 0b1010;
const octal: number = 0o744;
```

string

The type `string` is used to store textual data:

```
const message: string = 'Welcome";
```

Array

There are two ways of defining `Array` types:

```
 // first way -> type[]
const firstArray: number[] = [1,2,3];

// second way -> Array<type>
const secondArray: Array<number> = [4,5,6];
```

In simple cases like this one there is no difference between them but if we are working with something more complex than a type of number then we won't be able to use the first notation.

Look at this example:

```
 // our function accepts an array of
objects with a label and a value as its
properties
function example(arg:
Array<{label:string,value:string}> ){
  // do something
}
```

`Array<{label:string,value:string}>` simply means that the argument is an `Array` of objects with properties of label and value, both of type string.

`object`

`object` represents a value that is not one of the primitives;

Let's say our function takes an argument of type `object`:

```
 function greetUser(user: object){
  // property name does not exist on type
object
  console.log(`hello ${user.name}`)
}
greetUser({name: 'Alberto', age:27});
// hello Alberto
```

The `Typescript` compiler will complain that the property `name` does not exist on the type `object`. Let's define better the properties of that Object.

```
 function greetUser(user:
{name:string,age:number}){
  console.log(`hello ${user.name}`)
}
greetUser({name: 'Alberto', age:27});
// hello Alberto
```

Now we specified all the properties of the user `object`, making it easier for us and for anybody who

looks at the code to know what they can and what they cannot do with that `object`.

Tuple

A `Tuple` allows you to define the type of the known elements of an Array.

```
let myTuple: [string,number,string];
myTuple = ['hi',5,'hello'];

console.log(myTuple);
// [ 'hi', 5, 'hello' ]
```

`TypeScript` will know the type of the elements of the know indexes that we define in the tuple but it won't be able to know the type of additional elements added to the array.

enum

An `enum` is a way of giving names to a set of numeric values:

```
enum Status {deleted, pending, active}

const blogPostStatus: Status =
Status.active;
```

```
console.log(blogPostStatus);
// 2
```

The values inside of an enum start from 0 so in our example before `active` corresponds to 2, `pending` to 1 and `deleted` to 0.

It is much more meaningful to say that the status of a blog post is `active` rather than 2.

If you want, you can override the starting point of an enum by specifying it like this:

```
 enum Status {deleted = -1, pending,
active}

const blogPostStatus: Status =
Status.active;

console.log(blogPostStatus);
// 1
```

Now `deleted` corresponds to -1, `pending` to 0 and `active` to 1, much better than before.

We can also access values of an enum based on their value:

```
 enum Status {deleted = -1, pending,
active}
console.log(Status[0]);
// pending
```

any

As the name implies, `any` means that the value of a certain variable can be anything.
We may use it when dealing with 3rd party libraries that do not support `TypeScript` or when upgrading our existing code from plain `JavaScript`.

`any` allows us to access properties and methods that may not exist.

We can also use `any` when we only know part of our types:

```
 let firstUser: Object<any> = {
   name: 'Alberto',
   age: 27
}

let secondUser: Object<any> = {
   name: 'Caroline'
}
```

We expect both variable to be of type `object` but we are not sure of their properties, therefore we use `any`.

`void`

`void`, as the name implies, defines the absence of type.
It is often used in scenarios like this:

```
function storeValueInDatabase(objectToStore): void {
  // store your value in the database
}
```

This function takes an `object` and stores it in our database but does not return anything, that's why we gave it a return value of `void`.

When declaring variables of type `void` you will only be able to assign values of `null` and `undefined` to them.

`null` and `undefined`

Similarly to `void`, it is not very useful to create variables of type `null` or `undefined` because we would only be able to assign them `null` and `undefined` as values.

When talking about *union types* you will see the use of these two types.

`never`

`never` is a value that never occur, for example we can use it for a function that never returns or that always throws an error.

```
function throwError(error:string): never{
  throw new Error(error)
}
```

This function only throws an error, it will *never* return any value.

Interfaces, Classes and more

Interfaces

In one of the prior examples I set the type of our variable like this:
`Array<{label:string,value:string}>`

But what if the shape of our variable is much more complicated and we need to reuse it in multiple places?
We can use an `interface` to define the shape that that variable should have.

```
interface Car {
  wheels: number;
  color: string;
  brand: string;
}
```

Be careful, an `interface` is not an `object`, we use `;` and not `,`.

We can also set **optional properties** like this:

```
interface Car {
  wheels: number;
  color: string;
  brand: string;
  coupe?: boolean;
}
```

The `?` defines the property as optional, even if we create a new car object without it, `TypeScript` won't raise any error.

Maybe we don't want certain properties to be editable after the creation of the object, in that case we can mark them as **readonly**.

```
interface Car {
  readonly wheels: number;
  color: string;
```

```
  brand: string;
  coupe?: boolean;
}
```

Upon creation of our car object we will be able to set the number of wheels but we won't be able to change it afterwards.

With interfaces we can also create the shape for functions, not justs objects.

```
 interface Greet {
  (greeting: string, name: string): string
}

let greetingFunction: Greet;

greetingFunction = (greeting: string, name:
string): string => {
   console.log(`${greeting} ${name}`);
   return `${greeting} ${name}`;
}
greetingFunction('Bye','Alberto');
```

We create the shape that the function should have and then we assign the variable to a function of that said shape.
If we assigned the variable to a function with a different shape that would have thrown an error.

Extending Interfaces

An interface can extend another interface and inherit the members of the previous one:

```
interface Vehicle {
  wheels: number;
  color: string;
}

interface Airplane extends Vehicle {
  wings: number;
  rotors: number;
}
```

As you can see an `object` of type airplane will expect to have all four properties, not just the two defined in the interface Airplane.

Classes

`TypeScript` classes are very similar to ES6 classes and allow us to perform prototypal inheritance to build reusable components for our applications.

Just as a reminder of how they look:

```
class Animal {
  eat = ()=> {
    console.log('gnam gnam');
  }
  sleep = () => {
    console.log('zzzz');
  }
}

class Human extends Animal {
  work = ()=> {
    console.log('zzzzzzz');
  }
}
const me = new Human();
me.work();
// zzzzzzz
me.eat();
// gnam gnam
me.sleep()
// zzzz
```

We have created two classes, the second one inherits two methods from the first one.

A difference with ES6 classes is that `TypeScript` allows us to define the way that class members will be accessed from our application.

If we want them to be accessible anywhere we need to use the keyword `public`.

```
class Animal {
  public eat = ()=> {
    console.log('gnam gnam')
  }
  public sleep = () => {
    console.log('zzzz')
  }
}

const dog = new Animal();
dog.eat();
// gnam gnam
```

In `JavaScript` all class members are public, we cannot restrict access to them.

`TypeScript` also offers us to mark a member as `private` which means that it will not be accessible from outside of the class.

```
class Animal {
  public eat = ()=> {
```

```
    console.log('gnam gnam')
  }
  public sleep = () => {
    console.log('zzzz')
  }
}

class Human extends Animal {
  private work = ()=> {
    console.log('zzzzzzz')
  }
}

const me = new Human();
me.work();
// Property 'work' is private and only
accessible within class 'Human'
```

We can also mark a member as `protected` which means that it will be accessible only inside the class where it's declared and the classes extending it.

```
class Human {
  protected work = ()=> {
    console.log('zzzzzzz')
  }
}
```

```
class Adult extends Human {
  public doWork = () => {
    console.log(this.work)
  }
}
```

The class `Adult` extends the class `Human`, therefore it can access the same `protected` methods.

Intersection Types and Union Types

We have seen how we can define basic types with `TypeScript`, now let's dive into more advances types.

Intersection Types

As the name implies, Intersection Types allow us to join togther multiple types. Let's look at how to use them:

```
 interface Person {
  sex: 'male' | 'female' | 'N/A'
  age:number,
}

interface Worker {
```

```
  job: string
}

type Adult =  Person & Worker;

const me: Adult = {
  sex: 'male',
  age: 27,
  job: 'software developer'
}
console.log(me);
// { sex: 'male', age: 27, job: 'software
developer' }
```

As you can see, we have two distinct types, `Person` and `Worker` and we combined them together to create the type `Adult` which is a combination of the two.

Be careful when combining types that have two properties of the same but of different types because that will cause an error in the compiler.

Union Types

```
const attendee = string | string[];
```

The variable can be either a `string` or an `array of strings`.

Another example may be:

```
 const identifier = string | number |
string[];
```

We use the pipe (|) symbol to separate each type.

Remember that we can only access common members of all types of the union.

```
 interface Kid {
   age:number
}
interface Adult {
   age: number,
   job: string
}

function person(): Kid | Adult {
   return { age: 27 }
   }

const me = person();
me.age // ok
me.job // error
```

The property `job` exists only on the interface `Adult` and not on the `Kid` one therefore we cannot access it in our Union Type.

Conclusion

This section showed you why `TypeScript` can be a useful tool when working with `JavaScript`, especially when working together with other people.

What is left to do now is to start working on a simple project and get familiar with it.

At first it may seems like a chore to have to type your variables but once the project grows and it gets more complicated, you will start noticing how beneficial it is, especially when coming back to a project after months spent working on something else.

For every further query you have regarding `TypeScript`, the best resource I would suggest is the official documentation that can be found at this link: https://www.`TypeScript`lang.org/docs/ .

TypeScript Quiz

TS-01 Which one of these is not a real basic type of TypeScript?

- ☐ Tuple
- ☐ void
- ☐ enum
- ☐ every

TS-02 What is the correct type of the following variable?

```
const x = 0xf00d;
```

- ☐ string
- ☐ void
- ☐ hex
- ☐ number

TS-03 Which of the following type definition is wrong?

- ☐ `const firstArray: number[] = [1,2,3]`
- ☐ `const firstArray: Array<number> = [1,2,3]`
- ☐ `const firstArray: number<Array> = [1,2,3]`

TS-04 What is the correct output of the following code?

```
 enum Rank {first, second, third}


const myRank: Rank = Rank.second;
console.log(myRank);
```

- ☐ "second"
- ☐ {second:1}
- ☐ true
- ☐ 2
- ☐ 1

TS-05 What is the correct way of defining an `Interface`?

- ☐ interface Car = { wheels: number }
- ☐ interface Car { wheels = number }
- ☐ interface Car { wheels: number }
- ☐ interface Car = { wheels = number }

Conclusion

I sincerely thank you for making it this far. This is my first book and knowing that somebody read it all fills me with joy.

I hope that you found it useful and if that is the case I suggest you to follow my new website (inspiredwebdev.com) and my DevTo (https://dev.to/albertomontalesi) where I write articles about Web Development.

Thank you very much for your time and your support.

A special thanks to all the users on GitHub that made pull requests and opened issues to help fix mistakes and improve the quality of the book.

Quiz Solutions

Introduction to JavaScript solutions

JS-01

- `var important! = "important!"`

Variable names cannot contain punctuation marks

JS-02

- `Object`

JS-03

- `const car = { color: "red" }`

JS-04

- `false`

JS-05

- `["melon", "apple", "banana", “orange"]`

End of Chapter 1 Quiz Solutions

1.1

- Good morning

1.2

- 2

1.3

- 100

1.4

- ReferenceError

End of Chapter 2 Quiz

2.1

- B

The correct syntax for an arrow function is the so called fat arrow =>

2.2

- 10

Inside an arrow function, the `this` keyword inherits from its parent scope which in this case it's the window, therefore the `age` remains 10.

2.3

```
 (arg) => {
  console.log(arg);
}
```

End of Chapter 3 Quiz

3.1

```
 function calculatePrice(total, tax = 0.1,
tip = 0.05){
  // When no value is given for tax or tip,
the default 0.1 and 0.05 will be used
  return total + (total * tax) + (total *
tip);,
}
```

3.2

- 10

End of Chapter 4 Quiz

4.1

```
let result = `${a} ${c} ${d} ${b} ${e}`;
```

4.2

```
let result = `${a} ${c} ${b} ${e} ${d}`;

console.log(result);
// 1 plus 2 equals 3
```

4.3

```
console.log(`this is a very long text
a very long text`);

// this is a very long text
// a very long text
```

End of Chapter 5 Quiz

5.1

- true

it will begin checking after 3 characters

5.2

- false

`endsWith` is case sensitive

5.3

```
let batman = `${str.repeat(8)} ${bat}`;
console.log(batman);
// "NaNaNaNaNaNaNaNa BatMan"
```

End of Chapter 6 Quiz

6.1

```
let hungry = "yes";
let full = "no";

[hungry, full] = [full, hungry];
console.log(hungry);
```

```
// no
console.log(full);
// yes
```

6.2

```
 let arr = [ "one", "two", "three" ];
// solution:
let [one,two,three] = arr;

console.log(one);
// "one"
console.log(two);
// "two"
console.log(three);
// "three"
```

End of Chapter 7 Quiz

7.1

- for of

7.2

- Tom Jerry Mickey

End of Chapter 8 Quiz

8.1

```
Array.from(apple);
```

8.2

- 4

`Array.find()` will return the *first* element that matches our condition.

8.3

- true

`Array.some()` will return `true` if one element matches the give condition

8.4

- [1,4,9]

The `map` function we passed will return "x * x" for each of the values in the array

End of Chapter 9 Quiz

9.1

- [...]

9.2

```
const menu = [...veggie,
"pasta", ...meat];
```

9.3

```
const [first,second,...losers] = runners;
console.log(...losers)
```

We use the rest operator to grab all the values remaining after the first two

9.4

- [1, 2, 3, 4, 5]

End of Chapter 10 Quiz

10.1

```
const animal = {
  name,
  age,
  breed,
}
```

10.2

- "Alberto"

10.3

- `color is not defined`

since the variable name and the property name don't match we needed to write: `color: favoriteColor` instead of just `color`

End of Chapter 11 Quiz

11.1

- a primitive

Symbols are a new type of primitive introduced in ES6

11.2

- they are unique

Symbols are always unique

11.3

- Tom Jim

The second time we assigned "Tom" we overwrote the first "Tom" because we did not use a `symbol`.

11.4

- Symbol(Tom) Symbol(Jane) Symbol(Tom)

This time we see all three of them because we stored them in a `symbol`, keeping them unique and avoiding naming collisions.

End of Chapter 12 Quiz

12.1

- just syntactic sugar

classes are primarily syntactic sugar over JavaScript's existing prototype-based inheritance. The class syntax does not introduce a new object-oriented inheritance model to `JavaScript`.

12.2

- const person = class Person {...}
- class Person {...}

both answer 1 and 3 are correct. The first one is called *class expression* while the second one is called *class declaration*.

12.3

- A method that can be accessed only by the `class` itself

A `static` method is A method that can only be accessed by the `class` itself.

12.4

- ReferenceError: Must call super constructor in derived class before accessing 'this'

12.5

```
class Adult extends Person {
  constructor(name,age,work){
    super(name,age);
```

```
    this.work = work;
  }
}
```

Remember to add the keyword `super` to create the original class before you extend it.

End of Chapter 13 Quiz

13.1

- is an object representing the eventual completion or failure of an asynchronous operation

13.2

```
 const myPromise = new Promise((resolve,
reject) => {
  resolve();
  console.log("Good job!");
});
```

13.3

- `Promise.some()`

13.4

- 3,4

For the first two `.then` we did only passed one argument, meaning that we were calling `console.log` only if the promise resolved, but in our example it was being rejected. In the case of `.catch` the first argument is the callback for when a promise is being rejected and that is why we can see 3 being logged in the console. The latest `.then` is chained off of the promise created by `.catch` which resolved and therefore we also see 4 being logged.

End of Chapter 14 Quiz

14.1

- function*(){...}

14.2

- It can be stopped and restarted

14.3

- `Object { value: "Pomelo", done: false }`

The last output of `fruits.next();` is `Object { value: "Pomelo", done: false }`. Be careful that `done` is set to `false` and not `true`.

14.4

- `Object { value: undefined, done: true }`

The correct output is `Object { value: undefined, done: true }` because `.return()` will end the generator and return the value that we passed in. In this case we got `undefined` because we did not pass anything inside `.return()`.

End of Chapter 15 Quiz

15.1

- to define custom behavior of fundamental operations

From MDN:

the Proxy object is used to define custom behavior for fundamental operations (e.g. property lookup, assignment, enumeration, function invocation, etc).

15.2

- 2

A `Proxy` can take a handler and a target

15.3

- the **target** parameter of a `Proxy` **can be** another `Proxy`

It can be any sort of object, including a native array, a function or even another proxy.

End of Chapter 16 Quiz

16.1

- StrongSet

16.2

- A WeakSet can only store objects

16.3

- a Map stores both keys and values, a Set only values

End of Chapter 17 Quiz

17.1

- `Array.prototype.includes()`

17.2

- false

The solution is `false` because we start looking for the value 5 from index 4, but the value 5 is present only that index 2.

17.3

```
2**2**2;
// 16
```

End of Chapter 18 Quiz

18.1

- " hello"

`padStart(6)` will add only 1 space to our string because it has length of 5 (6-5=1).

18.2

```
 strings.forEach(str =>
console.log(str.padStart(16)));
```

18.3

- Object.keys()

18.4

- 10

`Atomics.add()` will return the previous value of that index.

18.5

- 4

`Atomics.load()` will return the value of that index

End of Chapter 19 Quiz

19.1

- SyntaxError

Remember: `await` is only valid in `async` functions and `async` generators

19.2

- uncaught exception: the value is too big

End of Chapter 20 Quiz

20.1

- {...}

20.2

- `{c:3, d:4}`

20.3

- 1

Remember that the `Promise` created after `finally` will returned the value of the `Promise` onto which `finally` was chained.

End of Chapter 21 Quiz

21.1

```
 // many options
letters.flat(2);
// or
letters.flat().flat()
// or
letters.flat(Infinity)
```

21.2

```
Object.fromEntries(keyValueArray)
```

21.3

- "Alberto"

21.4

- `['a', 'b', 'c', 'd', ['e', 'f']]`

21.5

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

Introduction to TypeScript solutions

TS-01

- `every`

TS-02

- `number`

TS-03

- `const firstArray: number<Array> = [1,2,3]`

TS-04

- 1

TS-05

- `interface Car {  wheels: number }`