ALBERTO MONTALESI

JavaScript ES6 & beyond

from ES6 to ES9

Learn more at **albertomontalesi.github.io**

Introduction

Disclaimer

This book is intended for somebody already familiar with the basics of JavaScript, as I am only focusing on the new features introduced by ES6 and I won't be explaining what is a `var`, how to create a function, etc...

About me

My name is Alberto, I'm from Italy and I love programming.

As I was studying ES6 I decided that the best way for me to test my understanding of it was to write articles about it. I have now packaged those articles in a free ebook that you can read here or on my blog here.

Contributions & Donations

Any contributions you make are of course greatly appreciated, you can find the repository of this book here. If you enjoy my content and you want to donate me a cup of coffee, you can do so here.

License

Table of content

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

With the introduction of `let` and `const` in **ES6** we can know better define our variable depending on our needs. Let's have a look at the major differences between them.

`Var`

`var` are **function scoped**, which means that if we declare them inside a `for` loop (which is a **block** scope) they will be available globally.

```
for (var i = 0; i < 10; i++) {
  var global = "I am available globally";
}

console.log(global);
// I am available globally

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

`Let`

`let` (and `const` 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 our `let` inside our block-scope it **did not** change the value in the global scope, whereas when did the same with our `var` it leaked outside of the block-scope and also change it in the global scope.

`Const`

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

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

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

Important This **does not** mean that **const are immutable**.

The content of a `const` is an Object

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

person.age = 26;

// in this case no error will be raised, we are not re-assigning the variable but just
one of its properties.
```

The temporal dead zone

According to **MDN**:

> In ECMAScript 2015, let bindings are not subject to **Variable Hoisting**, which means that let declarations do not move to the top of the current execution context. Referencing the variable in the block before the initialization results in a ReferenceError (contrary to a variable declared with var, which will just have the undefined value). The variable is in a "temporal dead zone" from the start of the block until the initialization is processed.

Let's look at an 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**.

This happens because `var` are subject to **hoisting** which means that they are processed before any code is executed. Declaring a `var` anywhere is equivalent to **declaring it at the top**. This is why we can still access the `var` but we can't yet see its content, hence the `undefined` 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.

You may want to read this article to understand how `let` affects your performances compared to `var` before you choose to follow either Mathias Bynes or Kyle Simpson.

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:

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

The new syntax with a fat arrow looks like this:

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

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

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

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

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

Implicitly return

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

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

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

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

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

console.log(winner);
// 0: {name: "Usain Bolt", race: "100m dash", place: 1}
// 1: {name: "Justin Gatlin", race: "100m dash", place: 2}
// 2: {name: "Asafa Powell", race: "100m dash", place: 3}
```

To tell JavaScript that what's inside the curly braces is an **object literal** that 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:

```
// 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("open");
    })
})
```

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, trowing 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:

```
// 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(() => {
    // try to  toggle again the class
    this.classList.toggle("open");
    })
})
```

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

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 2 examples all show when to be careful using `this` inside of arrows.

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

```
const person = {
  age: 10,
  grow: () => {
    // error: *this* refers to the window
    this.age ++,
    },
}
```

Here's another example of when you should use a normal function instead of an arrow.

```
const orderRunners = () => {
  const runners = Array.from(arguments);
  return runners.map((runner, i) => {
    return ` #{runner} was number #{i +1}`;
    })
    console.log(arguments);
}
```

This code will return:

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

We don't have access to the `arguments` object in arrow functions, we need to use a normal function.

Chapter 3: 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:

```
const Bill = calculatePrice({ tip: 0.15, total:150});
```

We don't even have to pass the parameters in the same order as when we declared our function, since we are calling them the same way as the arguments JavaScript will know how to match them.

Don't worry about destructuring, we will talk about it in a later chapter.

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

Add a ternary operator

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

```
// 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>
`
```

Pass a function inside a template literal

Similarly to the example above, 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(' ')}
    </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>
`
```

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.

To learn more about use cases of *template literals* check out this article.

Chapter 5: Additional string methods

We are going to cover 4 new strings method:

- `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 repeat what we pass in.

```

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

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 this:

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

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

In ES6 we can now write this:

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

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

Since our `const` 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 it like this:

```
const { facebook:fb } = person.links.social;
// we rename the variable as *fb*
```

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" ]
```

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.

Destructuring functions

```
function totalBill({ total, tax = 0.1 }) {
  return total + (total * tax);
}

// as you see since we are using the same names, we don't have to pass the arguments in
the same order as when we declared the function
// we are also overriding the default value we set for the tax
const bill = totalBill({ tax: 0.15, total: 150});
```

Chapter 7: Iterables and looping

ES6 introduced a new type of loop, the `for of` loop.

The `for of` loop

Iterating over an array

Usually we would iterate using something like this:

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

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 `Object.entries()`.

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

for (const prop of Object.keys(car)){
  const value = car[prop];
  console.log(value,prop);
}
// BMW maker
// red color
// 2010 year
```

The `for in` loop

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

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.

Another differences is that we **can** stop a `for of` loop but we can't do the same with a `for in` loop.

```
const digits = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];

for (const digit of digits) {
  if (digit % 2 === 0) {
    continue;
  }
  console.log(digit);
}
// 1 3 5 7 9
```

The last important difference I want to talk is that the `for in` loop will iterate over new properties added to the object.

```
const fruit = ["apple","banana", "orange"];

fruit.eat = "gnam gnam";

for (const prop of fruit){
  console.log(fruit);
}
// apple
// banana
// orange
```

```
for (const prop in fruit){
  console.log(fruit[prop]);
}

// apple
// banana
// orange
// gnam gnam
```

Chapter 8: Array improvements

`Array.from()`

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

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

```
// our html
<div class="fruits">
  <p> Apple </p>
  <p> Banana </p>
  <p> Orange </p>
</div>

const fruits = document.querySelectorAll(".fruits p");
const fruitArray = Array.from(fruits);

console.log(fruits);
//it will return us an array containing 3 p tags [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(".fruits 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"]
```

`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.

It can be useful in instances where we have a json file, maybe coming from an API from instagram or something similar and we want to grab a specific post with a specific code that identifies it.

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

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

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 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, `.every()` will check that all items match the given condition.

```
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
```

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 changed aswell, but why? Because we did not actually create a copy but we just referenced our old array in the new one. This is how we would usually make a copy of an array in ES5 and earlier.

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

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

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

Spread into a function

```
// 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 ]
```

We can replace the `.apply()` syntax and just use the spread operator.

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
```

We provided 3 values inside our array but only have 2 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 already discussing this topic, it is worth mentioning that ES9 will introduce 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 }
```

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.

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,
}
```

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` is named the same way as the properties we are using we can reduce our typing by a lot.

Add functions to our Objects

Let's looks at an example from ES5:

```
const person = {
  name: "Alberto",
  greet: function(){
    console.log("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 does the same.

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

```
// arrow functions are anonymous, in this case you need to have a key
const person1 = {
  () => console.log("Hello"),
};

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

Dynamically define properties of an Object

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

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

First we created the Object and then we modified it.

In ES6 we can do both things at the same time, look here:

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

Chapter 11: Symbols

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

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.

As we mentioned earlier we can use them to create as 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, a common situation. 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 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.

Chapter 12: Classes

Quoting MDN:

> classes are primarily syntactical sugar over js'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", 25);
const caroline = new Person("Caroline",25);

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 way of creating a class:

- class declaration
- class expression

```
// class declaration
class Person {

}

// class expression

const person = class Person {


}
```

> Remember: class declaration (and expression) and **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 methods).

```
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",25);

alberto.greet();
// Hello, my name is Alberto and I am 25 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 syntactical 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, similarily to `Array.of()` for arrays?

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

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

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

`set` and `get`

We 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;
  }
  get nicknames(){
    return `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`:

```
// 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",25,"teacher");
```

We created a new `Class Adult` that inherits from `Person` 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:

```
alberto.age
// 25
alberto.work
// "teacher"
alberto.greet();
// Hello, my name is Alberto and I am 25 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 Classroom and the rest are our students and their marks.

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

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

// we can also loop over with for of
for(const student of myClass) {
  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 }
```

Chapter 13: Promises

What is a Promise?

From MDN:

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

JavaScript works almost entirely asynchronously which means that when we are retrieving something from an API, for example, our code won't stop executing. Look at this example to understand what is going to happen:

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

The code won't stop once it hits the fetch, therefore our next console.log will be executed before we actually get some value in return, meaning that the `console.log(data)` will be empty.

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

Callback hell

You may have heard of something called **callback hell** which looks roughly like this:

```
fs.readdir(source, function (err, files) {
  if (err) {
    console.log('Error finding files: ' + err)
  } else {
    files.forEach(function (filename, fileIndex) {
      console.log(filename)
      gm(source + filename).size(function (err, values) {
        if (err) {
          console.log('Error identifying file size: ' + err)
        } else {
          console.log(filename + ' : ' + values)
          aspect = (values.width / values.height)
          widths.forEach(function (width, widthIndex) {
            height = Math.round(width / aspect)
            console.log('resizing ' + filename + 'to ' + height + 'x' + height)
            this.resize(width, height).write(dest + 'w' + width + '_' + filename,
function(err) {
              if (err) console.log('Error writing file: ' + err)
            })
          }.bind(this))
        }
      })
    })
```

```
  }
})
```

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 read this article.

Here we will focus on how to write promises.

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 see the resuult 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 we will get:
// The value we get from the promise
```

These two examples are very simple but **promises** are very useful when dealing 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.

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

Chaining promises

We can chaing promises one after the other, using what was returned from the previous one as the base for the subsequent one, whether the promise resolved or got rejected.

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

myPromise
  .then(data => {
    // take the data returned and call a function on it
    return doSomething(data);
  })
  .then(data => {
    // log the data that we got from the previous promise
    console.log(data);
  })
  .catch(err => {
    console.error(err);
  })
```

We called a function (it can do whatever you want, in this case it does nothing) and we passed then value down to the next step where we logged it.

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**.

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(value) {
  console.log(value);
  // "Success"
}, function(value) {
  // not called
});

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

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

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

```
Promise
  .all([promise1, promise2])
  .then(data => {
    const[promise1data, promise2data] = data;
    console.log(promise1data, promise2data);
  });
// after 1000 ms
// my first value my second value
```

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

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

If one of the promise was rejected, all of them would asynchronously reject with the value of that rejection, no matter 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 promise 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 promises that resolves or rejects as soon as one of the promises in the iterable resolves or reject, 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!.

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 fruit of fruitList) {
```

```
    yield `I like to eat ${fruit}`;
  }
}


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

- 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 can 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 introduce in ES8 (ES2017) but know that the way they work is based on what you will see now.

Using a Generator in combination with a Promise will allow us to write asynchronous code that feels like synchronous.

What we want to do is to 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();
asyncFunc.next();
// call the promise and wait for it to resolve
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).

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;
  }
});

dogProxy.breed;
// "GERMAN SHEPHARD"
dogProxy.breed = "Labrador";
// changing breed to...
// "Labrador"
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 if your object is a phone number.

You can take the value given by the user and format it to match the standard formatting of your country.

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 methd 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");
// true
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 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
```

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`.

Another big difference that we can see is by trying to use `.clear` on a `WeakSet`: nothing will happen because a `WeakSet` cleans itself up after we delete an element from it.

```
dad = null;
family;
// WeakSet [ {…}, {…} ]

// wait a few seconds
family;
// WeakSet [ {…} ]
```

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 and 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((key,val) => console.log(val,key));
```

```
// 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;

myMap;
// Map(1) {{…}}
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`.

Chapter 17: Everything new in ES7 (ES2016)

ES7 (or ES 2016) 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 where to begin searching for an element. Default is 0, but we can also pass a negative value.

The first value we pass is the element to search and the second one is the index:

```
let array = [1,3,5,7,9,11];

array.includes(3,1);
// true
array.includes(5,4);
//false
array.includes(1,-1);
// false
array.includes(11,-3);
// true
```

`array.includes(5,4);` returned `false` because, despite the array actually contains the number 5, it is 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 onwards.

`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 ES7 we would have done this:

```
Math.pow(2,2);
// 4
Math.pow(2,3);
// 8
```

Now with the new exponential operator we can do this:

```
2**2;
// 4
2**3;
// 4
```

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.

Chapter 18: ES8 string padding, `Object.entries()`, `Object.values()` and more

ES8 (ES2017) introduced many new cool features, which we are going to see here. I will discuss `Async` and `Await` later as they deserve more attention.

String padding(`padStart` and `padEnd`)

We can now add some padding to our strings, either at the end (`padEnd`) or at the beginning (`padStart`).

```
"hello".padStart(6);
// " hello"
"hello".padEnd(6);
// "hello "
```

We said we want 6 as our padding, but why in both cases we got only 1 space? It happens 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);
// (3) ["father", "mother", "son"]
family.father;
"Jonathan Kent"
```

`Object.keys()` returned us 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);
// (3) ["Jonathan Kent", "Martha Kent", "Clark Kent"]

Object.entries(family);
// (2) ["father", "Jonathan Kent"]
// (2) ["mother", "Martha Kent"]
// (2) ["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 in function parameter lists and calls

This is just a minor change to a syntax. Now, when writing objects we need to 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 will make the life easier to your colleague or team members.

```
// 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

Chapter 19: ES8 Async and Await

ES8 (ES2017) introuced 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, log the data
  console.log(res);
}).catch( err => {
  // or log 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 promie returns its result

Let's see what happens if we try to use `await` outside an `async` function

```
// use async 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 async 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
```

Chapter 20: ES9 what is coming?

ES 2018 (ES9) has not been released yet but we can look at the proposals for features that have reached the stage 4 (the final stage) and that will be included in the new upcoming version of ECMAScript. You can find the list on github.

Rest / Spred for objects

Now we can use the rest/spread syntax for objects, 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}
```

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.

```
for await (const line of readLines(filePath)) {
  console.log(line);
}
```

> 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.

```
fetch("your-url")
  .then(result => {
    // do something with the result
  })
  .catch(error => {
    // do something with the error
  })
  .finally(()=> {
    // do something once the promise is finished
  })
```

RegExp features

4 new RegExp related features will make 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}`);  // prints 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.

Conclusion

Thank you for making it this far, I hope you enjoyed this book. You can continue following me on Medium.

Any contribution you make to this book is of course greatly appreciated, you can find the repository of this book here.

If you enjoy my content and you want to donate me a cup of coffee, you can do so here.