

UNIT 1

JAVASCRIPT



Part 1 JavaScript Basics. Collections.

Client-side Web Development
2nd course – DAW
IES San Vicente 2018/2019
Author: Arturo Bernal Mayordomo

Index

Introduction.....	3
Some JavaScript history.....	4
Editors and tools for learning / coding JavaScript.....	5
Browser console.....	5
Desktop editors.....	5
Web editors.....	6
Javascript basics.....	7
Integrating JavaScript within HTML.....	7
Variables.....	8
Constants.....	9
Functions.....	9
Arrow (lambda) functions.....	10
Default parameters.....	10
Conditional statements.....	11
Loops.....	12
Basic data types.....	13
Variable scope.....	17
Operators.....	18
Collections.....	22
Arrays.....	22
Iterating through Arrays.....	22
Array methods.....	24
New methods in ES2015.....	26
Rest and spread.....	28
Destructuring arrays.....	29
Map.....	30
Set.....	31

Introduction

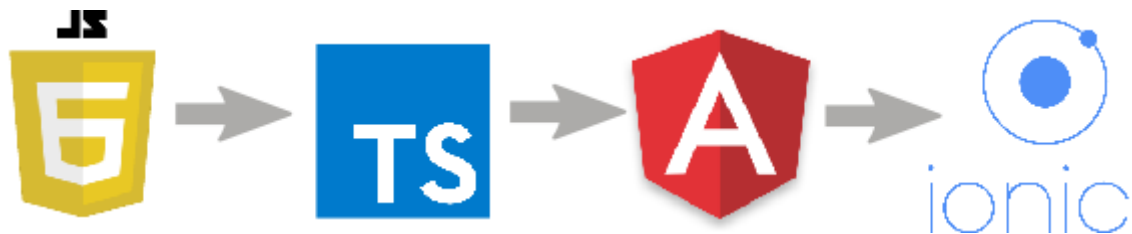
In this unit, we will learn the fundamentals of programming with **JavaScript** using the newest versions (ES2015 and beyond). We'll also review some APIs like Geolocation, Google Maps, etc.. Finally we'll learn TypeScript, which is a superset of JavaScript and adds some really useful features.

This is not a “fundamentals of programming” unit. Basic programming skills (structured and object oriented programming) are assumed. Instead, we'll learn the syntax and peculiarities of JavaScript (reviewing some fundamental programming skills during that process).

[TypeScript](#) is becoming more and more popular every day, and it will be much more used in the future because Angular (version 2 and beyond) is based on it.

Angular is the new version of a very popular and powerful JavaScript framework (AngularJS). And, as said before, it's now built on top of Typescript and has changed a lot since the previous version. Unit 2 is all about Angular.

Finally, in Unit 3, we'll learn the fundamentals of [Ionic](#), which is a framework designed to build hybrid mobile apps based on web technologies. It's built on top of Angular and uses [Cordova](#) to interact with the native device sensors and services like the camera, push messages, touch sensor, accelerometer, etc.



Unfortunately, due to time restrictions, technologies like [Electron](#), used to develop desktop applications with web technologies (like Visual Studio Code), will be left out the course.

Some JavaScript history

Back in 1995, **Netscape** wanted to implement their own Java VM into its browser, but they decided to go for a more friendly language, oriented to casual or amateur programmers. This language had to be very permissive (optional semicolons at the end of instructions, no variable types, ...), which is not necessarily good. The name of this project was **Mocha** (Java means coffee). They initially called this language LiveScript (Java was a registered trademark), but few months later, Sun agreed to let them use the word Java, so it became JavaScript (December 1995).

In 1996, Microsoft complained because the JavaScript API changed too frequently and they wanted to implement their own “compatible” version. Netscape thought that Microsoft wanted to use JavaScript to make incompatible versions (which indeed they tried to) and become the standard. In order to protect JavaScript standard, Netscape created an independent standard called **ECMAScript**.

To understand the difference, ECMAScript is the specification (what does the language have to support), and JavaScript is a implementation of that (Other implementation is Adobe’s ActionScript). This standard was adopted by many companies including IBM, Microsoft, Netscape, Sun, ... Everything went well until version 3...

From 1999 until 2006, no new version of ECMAScript came out, essentially because Microsoft became a monopoly with their Internet Explorer and didn’t push any innovation to the standard. Fortunately, when Netscape died, it was open sourced and Mozilla Firefox was born from it (other browsers like Opera already existed but their market share was very low). In 2006, there were discussions whether to push some minimal changes to the standard and release an ES3.1 (Microsoft, Yahoo,...) version or to push bigger changes and release an ES4 (Mozilla, Opera, Adobe) version.

In the end ES3.1 won the battle and ES4 was postponed. For marketing reasons, ES3.1 was renamed to **ES5** (ES4 never existed), which is the most common JavaScript implementation used nowadays, and **ES6** should have what was going to be included in the original ES4. ES5 was released in 2009 and ES5.1 in 2011.

ES6 was released in June 2015 and renamed to **ES2015** (marketing again). From then on, the goal was to release a new specification of the language more often (with less added features), so in June 2016 **ES2016** was released with minor changes over the previous version. The next version of ECMAScript is always called **ES.Next**.

Today, only the most recent desktop browsers implement the ES2015 standard, so if you want to make a web application compatible with most browsers you need a tool called **transpiler** that will convert your ES2015 code into ES5 compatible code. You can check browser’s compatibility with ES standards here:

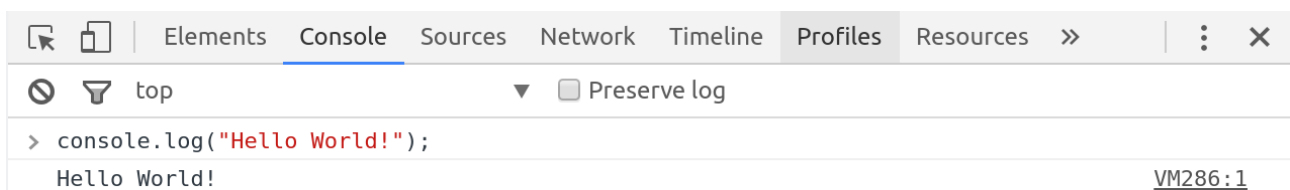
<http://kangax.github.io/compat-table/es6/>

Editors and tools for learning / coding JavaScript

In this course, you can choose whichever editor or IDE you're most comfortable with. However, it's better that you choose one the editors used in the examples (essentially Visual Studio Code), because all instructions will be given having that editor in mind. There are tens of options available, and in the beginning, it's not so important which one you use.

Browser console

Open your browser (recommended Chrome or Firefox) and press F12 to show the integrated developer tools. Go to the **console** tab. There you can execute JavaScript instructions and see the result immediately. This choice is good for testing small pieces of code.



Just for curiosity, you can try this piece of code to reproduce one of JavaScript's most famous bugs (decimal precision in some mathematical operations)

```
> console.log(5.1 + 3.3);  
8.399999999999999
```

Desktop editors

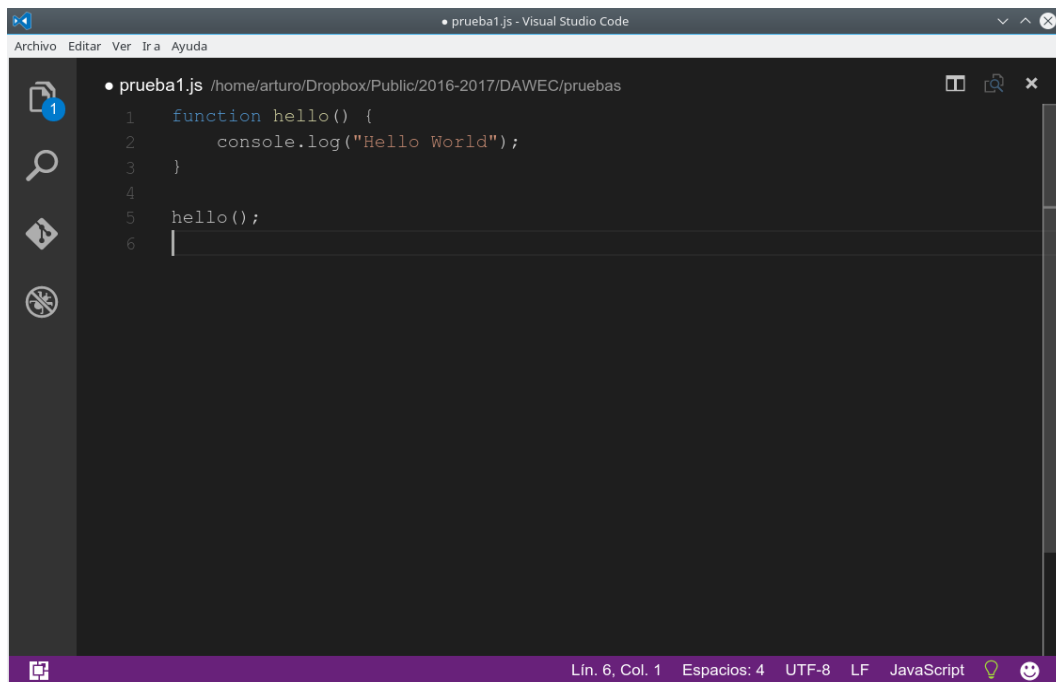
Most of code editors and IDEs support at least JavaScript syntax highlighting, while some of them also support code completion, integration with jQuery, Angular, etc. You can use the one you're more comfortable with (Visual Studio, Netbeans, Webstorm, Atom, Sublime, Kate, Notepad++, ...), but all examples will be made using Visual Studio Code, because it integrates very well with Javascript, Node, Typescript, and Angular.

Visual Studio Code

This editor (with some IDE characteristics) is developed by Microsoft and open source. It's available for Windows, Mac and GNU/Linux. It's been developed using [Electron](#), a framework that uses **Chromium** and **Node.js** for developing desktop applications using HTML, CSS and JavaScript.

The main features of Visual Studio Code are that is lightweight, supports syntax highlighting, and code completion for **JavaScript** and **TypeScript** (we'll need it for Angular 2) among others. It also integrates very well with Angular (1 and 2), and its functionality can be extended with **extensions**. Other editors that integrate with [TypeScript](#) are: Visual Studio, Atom, Sublime Text, Webstorm, Emacs or Vim (most of them through plug-ins). VS Code is the recommended choice for this course!.

You can download it from: <https://code.visualstudio.com/Download>



Web editors

Using an online editor is also a valid choice, at least for testing code which is not too complex or large. This option is good for testing code quickly without needing to have an editor installed, and also for easily sharing your code with other people. You usually don't have many advantages present in desktop editors like code completion or plugins, for example.

Two popular web editors available on Internet are Fiddle (<https://jsfiddle.net/>) and Plunker (<https://plnkr.co/>). You can sign in and save your projects so you can continue them anywhere with any device.

Javascript basics

It's probable that you have at least some basic knowledge of JavaScript, or even some experience with it creating web pages. However, it's not a bad thing to review some of the basic aspects of the language. It will help you to consolidate the basics and also you'll learn some new things that you didn't know before, like the new ES2015+ syntax and other improvements.

Integrating JavaScript within HTML

To integrate JavaScript code in your HTML, you must use the **<script>** tag. The recommended place to put this tag is just before the closing **</html>** tag so the browser can load and build the DOM structure before processing the JavaScript code, otherwise, the browser will block the page rendering until it has processed all JS code.

¿Where can we put our JavaScript code?:

Inside the script tag

```
<!DOCTYPE>
<html>
  <head>
    <title>JS Example</title>
  </head>
  <body>
    <p>Hello World!</p>
    <script>
      console.log("Hello World!");
    </script>
  </body>
</html>
```

In a separate file

File: example1.html

```
<!DOCTYPE>
<html>
  <head>
    <title>JS Example</title>
  </head>
  <body>
    <p>Hello World!</p>
    <script src="example1.js"></script>
  </body>
</html>
```

File: example1.js

```
console.log("Hello World!");
```

By the way, what **console.log()** does is write whatever you pass to it in the browser's console (F12 to open the developer tools and see the result). You can also use the method **console.error()** instead to print error messages.

You can use the **<noscript>** tag to put some HTML that will be rendered only

when the browser doesn't support JavaScript or it has been disabled. This is useful to tell the user that your web application needs JavaScript enabled in order to run.

```
<!DOCTYPE>
<html>
  <head>
    <title>JS Example</title>
  </head>
  <body>
    <p>Hello World!</p>
    <noscript>
      <h1>JavaScript is not enabled. Please, enable it or the application
        won't run.</h1>
    </script>
  </body>
</html>
```

Variables

You can declare a new variable using the reserved word **let** (you can also use **var** but it's not recommended since ES2015), and after that the variable's name (in **CamelCase** format with the first letter in lowercase), which can begin with a letter, underscore (`_name`) or dollar (`$name`). In JavaScript variables have **no explicit type**. Instead, the type of a variable changes internally depending which value you assign to it. You can assign a string value to a variable and then reassign an integer later.

```
let v1 = "Hello World!";
console.log(typeof v1); // Prints -> string

v1 = 123;
console.log(typeof v1); // Prints -> number
```

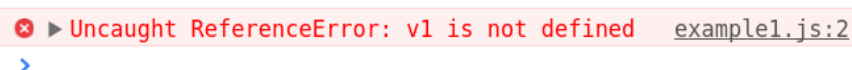
What happens if a variable has been declared but with no value?. Until a value is assigned, it will have a special type called **undefined**. Note that this is different from **null** (which is a value).

```
let v1;
console.log(typeof v1); // Prints -> undefined
if (v1 === undefined) { // (!v1) or (typeof v1 === "undefined") also works
  console.log("You forgot to set a value to v1");
}
```

And, what if we forget to put the keyword **let** or **var**?. JavaScript will notice that and declare a new **global** variable for you. This is **NOT** recommended because global variables are dangerous if we don't know what we're doing. Use always **let** the first time a variable appears in your code.

How to avoid forgetting **let**: Use the special declaration '**use strict**' at the **beginning** of your JavaScript file. This way, it won't let you declare global variables by omitting **let** keyword.

```
'use strict';
v1 = "Hello World!";
```



```
✖ ▶ Uncaught ReferenceError: v1 is not defined example1.js:2
>
```


Constants

We can also declare constants with keyword **const**. The interpreter will throw an error if we try to modify a constant. Constants **must** be declared with a value.

```
'use strict';  
const MY_CONST=10;  
MY_CONST=200; → Uncaught TypeError: Assignment to constant variable.
```

Functions

In JavaScript we declare a function by using the keyword **function** before the function's name. Arguments will go inside the parenthesis after the name (remember that there are no types in JavaScript). Then, we'll write function's body inside brackets. Function's names (like variables) should be written in **CamelCase** format with the first letter lowercase.

```
function sayHello(name) {  
  console.log("Hello " + name);  
}  
  
sayHello("Tom"); // Prints "Hello Tom"
```

You don't need to have your function declaration before calling it, because the JavaScript interpreter processes first function declarations and then, executes code.

You can send more or fewer arguments that were established in the declaration. If you send more arguments, those that are not in the declaration will be ignored. If you send less, values not received will be assigned as **undefined**.

```
sayHello(); // Prints "Hello undefined"
```

Returning values

You can use the keyword **return** to return a value from a function. If we try to get a value from a function that doesn't return anything, we'll get **undefined**.

```
function totalPrice(priceUnit, units) {  
  return priceUnit * units;  
}  
  
let total = totalPrice(5.95, 6); // Returns 37.5  
console.log(total); // Prints 37.5
```

Anonymous functions

You can declare an anonymous functions by not using a name for that function. You can then assign it to a variable or pass it as an argument to a function. Because it's a type of value, it can be assigned to (or referenced from) multiple variables.

```
let totalPrice = function(priceUnit, units) {  
  return priceUnit * units;  
}  
  
console.log(typeof totalPrice); // Prints "function"
```

```
console.log(totalPrice(5.95, 6)); // Prints 37.5
let getTotal = totalPrice;
console.log(getTotal(5.95, 6)); // Prints 37.5. It works
```

Arrow (lambda) functions

An important addition to the language is the possibility to use **arrow functions** (sometimes also called **lambda functions** or **lambda expressions**). This is another way to create anonymous functions but with some advantages when working with objects as we'll learn in the future.

To notice the difference, let's see two equivalent functions (one anonymous function and one arrow function that do the same):

```
let sum = function(num1, num2) {
  return num1 + num2;
}
console.log(sum(12,5)); // Prints 17
```

```
let sum = (num1, num2) => num1 + num2;
console.log(sum(12,5)); // Prints 17
```

When declaring an arrow function, the **function** keyword is removed. Only if one parameter is received, the parenthesis can be omitted. After the parameters, we must write an **arrow** (**=>**), and then the function's contents.

```
let square = num => num * num;
console.log(square(3)); // Prints 9
```

If there's only a return instruction inside the arrow function, we can omit the curly brackets **{ }** and the return keyword. If there's more than one instruction, we, must use curly brackets, and it works like a normal function (if needed, use return).

```
let sumInterest = (price,percentage) => {
  let interest = price * percentage / 100;
  return price + interest;
}
console.log(sumInterest(200,15)); // Prints 230
```

Default parameters

If a parameter is declared in a function and it's not passed when we call that function, its value is set to **undefined**.

```
function Person(name) {
  this.name = name;

  this.sayHello = function() {
    console.log("Hello! I'm " + this.name);
  }
}
let p = new Person();
p.sayHello(); // Prints "Hello! I'm undefined"
```

A workaround that was used in ES5 to set default parameters was using the '||' (or) operator, so if a parameter evaluates to undefined, a second value is assigned.

```
function Person(name) {  
  this.name = name || "Anonymous";  
  ...  
}
```

However, in **ES2015** we have the option to set default values directly.

```
function Person(name = "Anonymous") {  
  this.name = name;  
  ...  
}
```

We can assign a value based on an expression (and we can use other parameters in that expression).

```
function getTotalPrice(price, tax = price * 0.07) {  
  return price + tax;  
}  
console.log(getTotalPrice(100)); // Prints 107
```

Conditional statements

The **if statement** behaves like in most programming languages. It evaluates a condition that produces a boolean result and, if it's true, executes the code inside the **if** block. You can optionally add **else if** blocks and an **else** block.

```
let price = 65;  
  
if(price < 50) {  
  console.log("This is cheap!");  
} else if (price < 100) {  
  console.log("This is not so cheap...");  
} else {  
  console.log("This is expensive!");  
}
```

The **switch statement** also behaves similarly to other known programming languages. As you know, it evaluates a variable and executes the block corresponding to its current value. You usually need to put a **break** instruction at the end of each block, or it will continue executing instructions from the next block. Here's an example where two values execute the same code (you can also evaluate **strings**):

```
let userType = 1;  
  
switch(userType) {  
  case 1:  
  case 2:  
    console.log("You can access this area");  
    break;  
  case 3:  
    console.log("You don't have permission to access");  
    break;  
  default:  
    console.error("Invalid user type!");  
}
```

```
}
```

In JavaScript (this is not common among other languages), you can make a **switch** behave like an **if** statement. If you evaluate the boolean **true** instead of other type of value, **case** values can become conditions:

```
let age = 12;

switch(true) {
  case age < 18:
    console.log("You are too young to enter");
    break;
  case age < 65:
    console.log("You can enter");
    break;
  default:
    console.log("You are too old to enter");
}
```

Loops

We have the typical **while loop** that behaves like an if statement and it repeats over and over until the condition is false.

```
let value = 1;

while (value <= 5) { // Prints 1 2 3 4 5
  console.log(value++); // Value post-increment
}
```

Instead of **while**, we can use **do..while**. The difference is that the latter evaluates the condition at the end of the loop, so it will always execute the code at least once.

```
let value = 1;

do { // Prints 1 2 3 4 5
  console.log(value++); // Value post increment
} while (value <= 5);
```

The **for loop** also behaves like in other programming languages. You initialize one or more values, establish a condition, and do the increments (or the instructions that will be executed after each iteration) inside the parenthesis.

```
let limit = 5;

for (let i = 1; i <= limit; i++) { // Prints 1 2 3 4 5
  console.log(i);
}
```

As you should know, you can initialize more than one variable and also execute more than one instruction in each iteration separating them by a comma.

```
let limit = 5;

for (let i = 1, j = limit; i <= limit && j > 0; i++, j--) {
  console.log(i + " - " + j);
}
```

```
/* prints
1 - 5
2 - 4
3 - 3
4 - 2
5 - 1
*/
```

Inside a loop, we can use **break** and **continue** instructions. The first one exits (breaks) the loop immediately when executed and the second one goes to the next iteration skipping the rest of instructions (if there are more) inside the loop block (it also executes the corresponding increments if you're in a **for** loop).

Basic data types

Numbers

In JavaScript there's no difference between integers and decimal (float, double) numbers. The common type for all numbers is **number**.

```
console.log(typeof 3); // Prints number
console.log(typeof 3.56); // Prints number
```

You can also write numbers in scientific (exponential) notation:

```
let num = 3.2e-3; // 3.2*(10^-3)
console.log(num); // Prints 0.0032
```

Numbers are objects

In JavaScript everything is an **object**, even types which are primitives in other languages (Java, C++, etc.) like integers. That's why, if we type a dot after the number, we can access some methods or properties.

```
console.log(3.32924325.toFixed(2)); // Prints 3.33
console.log(5435.45.toExponential()); // Prints 5.43545e+3
console.log((3).toFixed(2)); // Prints 3.00 (integer numbers need to be inside parenthesis to access properties)
```

There's also a JavaScript internal object called **Number**, where we can access some other useful properties to work with numbers.

```
console.log(Number.MIN_VALUE); // Prints 5e-324 (smallest number)
console.log(Number.MAX_VALUE); // Prints 1.7976931348623157e+308 (biggest number)
```

There are also special values for numbers that are out of range (**Infinity** and **-Infinity**). We can compare if a number has one of those special values directly using **value === Infinity** or **value === -Infinity** for example. However there's a built-in function called **isFinite(value)** that returns false when the value is Infinity or -Infinity.

```
console.log(Number.MAX_VALUE * 2); // Prints Infinity
console.log(Number.POSITIVE_INFINITY); // Prints Infinity
console.log(Number.NEGATIVE_INFINITY); // Prints -Infinity
console.log(typeof Number.POSITIVE_INFINITY); // Prints number
```

```
let number = Number.POSITIVE_INFINITY / 2; // Still infinite!!
```

```

if(isFinite(number)) { // Same as (number !== Infinity && number !== -Infinity)
  console.log("Number is " + number);
} else { // Enters here
  console.log("Number is infinite!");
}

```

Operations with numbers

With numbers you can do all the typical operations (+, -, *, /, %, ...). But what happens if a value you operate with is not a number?. For example, a string. When you do number operations with values that are not numbers, it tries to cast those values into numbers. If it doesn't work, it returns a special value called NaN (Not a Number).

```

let a = 3;
let b = "asdf";
let r1 = a * b; // b is "asdf", and cannot be transformed into a number
console.log(r1); // Prints NaN

let c;
let r3 = a + c; // c is undefined, cannot be transformed into a number
console.log(r3); // Prints NaN

let d = "12";
console.log(a * d); // Prints 36. d can be transformed into a number 12
console.log(a + d); // Prints 312. The + operator concatenates if there's a string
console.log(a + +d); // Prints 15. Operator '+' in front of a value (+d) transforms it into a number

```

undefined and null

In JavaScript when a variable (or function parameter) has been created without a value, it's initialized to the special value **undefined**.

You shouldn't mistake **undefined** with **null**. The second is an empty value that you explicitly assign to a variable. Let's see the type of both values.

```

let value; // Value not assigned (undefined)
console.log(typeof value); // Prints undefined

value = null;
console.log(typeof value); // Prints object

```

As you can see, **null** is considered a type of object (or a value that should be an object but is empty). **undefined** is a special value that warns you that a variable has not been initialized yet.

Boolean

In JavaScript you represent boolean values always in lowercase (**true**, **false**). You can negate them using **!** before the value. We'll see more about booleans soon.

Strings

String values are represented inside 'single quotes' or "double quotes". You can also use the operator **+** to concatenate strings.

```

let s1 = "This is a string";

```

```
let s2 = 'This is another string';
```

```
console.log(s1 + " - " + s2); // Prints: This is a string - This is another string
```

When the string is inside double quotes, you can use single quotes inside directly and vice versa. However, if you want to store double quotes inside a string declared with double quotes also (or single quotes inside single quotes), you'll need to escape them in order not to close the string.

```
console.log("Hello 'World'"); // Prints: Hello 'World'  
console.log('Hello \'World\''); // Prints: Hello 'World'
```

```
console.log("Hello \"World\""); // Prints: Hello "World"  
console.log('Hello "World"'); // Prints: Hello "World"
```

Like numbers, strings are objects and have some methods you can use. All these methods don't change the original variable (unless you reassign it, of course)

```
let s1 = "This is a string";
```

```
// Get the length of a string
```

```
console.log(s1.length); // Prints 16
```

```
// Get the character at a position of the string (starting at 0)
```

```
console.log(s1.charAt(0)); // Prints "T"
```

```
// Gets the index when a substring first appears
```

```
console.log(s1.indexOf("s")); // Prints 3
```

```
// Gets the index when a substring last appears
```

```
console.log(s1.lastIndexOf("s")); // Prints 10
```

```
// Returns an array with all coincidences of a regular expression
```

```
console.log(s1.match(/.s/g)); // Prints ["is", "is", " s"]
```

```
// Gets the position of the first coincidence of a regular expression
```

```
console.log(s1.search(/[aeiou]/)); // Prints 2
```

```
// Replaces a regular expression coincidence (or a string) with a string (/g option replaces all)
```

```
console.log(s1.replace(/i/g, "e")); // Prints "Thes es a streng"
```

```
// Returns a substring (start position: included, end position: not included)
```

```
console.log(s1.slice(5, 7)); // Prints "is"
```

```
// Same as slice
```

```
console.log(s1.substring(5, 7)); // Prints "is"
```

```
// Like substring but with a difference (start position, number of characters from there)
```

```
console.log(s1.substr(5, 7)); // Prints "is a st"
```

```
// Transform into lowercase. toLowerCase doesn't work with special characters (ñ, á, é, ...)
```

```
console.log(s1.toLocaleLowerCase()); // Prints "this is a string"
```

```
// Transform into uppercase
```

```
console.log(s1.toLocaleUpperCase()); // Prints "THIS IS A STRING"
```

```
// Get string removing spaces, tabs and line breaks from the beginning and end
```

```
console.log(" String with spaces ".trim()); // Prints "String with spaces"
```

startsWith(substring) → Check if the current string starts with a substring

```
console.log(str.startsWith("This")); // Prints true
```

endsWith(substring) → Check if the current string ends with a substring

```
console.log(str.endsWith("string")); // Prints true
```

includes(substring) → Check if the current string contains a substring

```
console.log(str.includes("is")); // Prints true
```

repeat(times) → Returns the string repeated a number of times

```
console.log("la".repeat(6)); // Prints "lalalalala"
```

Since ES2015 string supports unicode characters with more than two bytes (4 hexadecimal characters), sometimes called *astral plane* values, using curly braces `\u{}`. Example:

```
let uString = "Unicode astral plane: \u{1f3c4}";  
console.log(uString); // Prints "Unicode astral plane: 🏄" (surfer icon)
```

These special characters return a value of 2 characters when the string length is measured:

```
let surfer = "\u{1f3c4}"; // ONE character: 🏄  
console.log(surfer.length); // Prints 2
```

However, if we transform the string into an array (of characters), it will be split correctly, each character into one array position:

```
let surfer2 = "\u{1f30a}\u{1f3c4}\u{1f40b}"; // 🏆🏄🏊 THREE characters:  
console.log(surfer2.length); // Prints 6  
console.log(Array.from(surfer2).length); // Prints 3 (converted to array correctly)
```

Template literals

Since ES2015 JavaScript supports multi-line string literals with variable substitution. We simply put the string between ``` (backquote) characters instead of simple or double quotes. Variables, operations, function calls, etc. Are put inside `${}` for substitution.

```
let num = 13;
```

```
console.log(`Example of multi-line string  
the value of num is ${num}`);
```

```
Example of multi-line string  
the value of num is 13
```

Converting between types

You can convert a value to a **string** using the **String(value)** function. You also can concatenate an empty string to force the conversion of what's being concatenated.

```
let num1 = 32;  
let num2 = 14;
```



```
// When concatenating a string, everything else is also converted to string
console.log(String(32) + 14); // Prints 3214
console.log("" + 32 + 14); // Prints 3214
```

You can convert a value to a **number** using the `Number(value)` function. You also can add the prefix `+` before the variable to get the same result.

```
let s1 = "32";
let s2 = "14";

console.log(Number(s1) + Number(s2)); // Prints 46
console.log(+s1 + +s2); // Prints 46
```

You can convert a value to a **boolean** using `Boolean(value)` function. You also can add `!!` (double negation) prefix before the value to force conversion. These are values that will return false: **empty string** (`""`), **null**, **undefined**, **0**. Other values should return true.

```
let v = null;
let s = "Hello";

console.log(Boolean(v)); // Prints false
console.log(!s); // Prints true
```

Variable scope

Global variables

When you declare a variable in the main block (outside of any function), it becomes a global variable. Variables declared without the **let** keyword are also global (unless you are in **strict mode**, which doesn't allow that). When executing JavaScript in a browser, the **window** global object will hold all global variables.

```
let global = "Hello";

function changeGlobal() {
  global = "GoodBye";
}

changeGlobal();
console.log(global); // Prints "GoodBye" (window. Is implied by default)
console.log(window.global); // Prints "GoodBye"
```

Let's try to declare a global variable inside a function in strict mode. If you don't use strict (not recommended), it would allow you to create a global variable.

```
'use strict';

function changeGlobal() {
  global = "GoodBye";
}

changeGlobal(); // Error → Uncaught ReferenceError: global is not defined
```

Function variables

All variables declared inside a function are local to that function.

```
function setPerson() {  
  let person = "Peter";  
}  
  
setPerson();  
console.log(person); // Error → Uncaught ReferenceError: person is not defined
```

If a global variable with the same name exists, the local variable will still be treated as different (won't update the global variable's value).

```
function setPerson() {  
  let person = "Peter";  
}  
  
let person = "John";  
setPerson();  
console.log(person); // Prints John
```

Operators

Addition '+'

This operator can be used with numbers (add) or strings (concatenation). But, what happens if we try to add a number with a string or anything that's not a number or a string?. Let's see.

```
console.log(4 + 6); // Prints 10  
console.log("Hello " + "world!"); // Prints "Hello world!"  
console.log("23" + 12); // Prints "2312"  
console.log("42" + true); // Prints "42true"  
console.log("42" + undefined); // Prints "42undefined"  
console.log("42" + null); // Prints "42null"  
console.log(42 + "hello"); // Prints "42hello"  
console.log(42 + true); // Prints 43 (true => 1)  
console.log(42 + false); // Prints 42 (false => 0)  
console.log(42 + undefined); // Prints NaN (undefined cannot be converted into a number)  
console.log(42 + null); // Prints 42 (null => 0)  
console.log(13 + 10 + "12"); // Prints "3212" (13 + 10 = 23, 23 + "12" = "3212")
```

When there's a string present, it always performs **concatenation**, so it will try to transform the other value into a string (if it's not). If there's no string, it will convert the values that aren't numbers into numbers and try to do an **addition operation**. If it fails to convert a value into a number, it will return **NaN**.

Arithmetic operators

Operation symbols other than addition, like **subtraction** (-), **multiplication** (*), **division** (/), and **module** (%), will always operate with numbers, so every value will be transformed into a number (if it's not already a number).

```
console.log(4 * 6); // Prints 24  
console.log("Hello " * "world!"); // Prints NaN  
console.log("24" / 12); // Prints 2 (24 / 12)  
console.log("42" * true); // Prints 42 (42 * 1)
```

```

console.log("42" * false); // Prints 0 (42 * 0)
console.log("42" * undefined); // Prints NaN
console.log("42" - null); // Prints 42 (42 - 0)
console.log(12 * "hello"); // Prints NaN ("hello" can't be converted into a number)
console.log(13 * 10 - "12"); // Prints 118 ((13 * 10) - 12)

```

Unary operators

In JavaScript we can do preincrement (++variable), postincrement (variable++), predecrement (--variable) and postdecrement (variable--). You should know the difference by now.

```

let a = 1;
let b = 5;
console.log(a++); // Prints 1 and increments a (2)
console.log(++a); // Increments a (3), and prints 3
console.log(++a + ++b); // Increments a (4) and b (6). Adds (4+6), and prints 10
console.log(a-- + --b); // Decrements b (5). Adds (4+5). Prints 9. Decrements a (3)

```

Also, we can use the negative (-) or plus (+) sign in front of a number to change or maintain its sign. If you apply these operators to a value that's not a number, it will be converted to a number first. That's why it's a good option to use **+value** to convert to a number, which is equivalent to write **Number(value)**.

```

let a = "12";
let b = "13";
let c = true;
console.log(a + b); // Prints "1213"
console.log(+a + +b); // Prints 25 (12 + 13)
console.log(+b + +c); // Prints 14 (13 + 1). True -> 1

```

Relational operators

Comparison or relational operators compare 2 values and return a boolean (true or false). These operators are almost the same we find in most languages except for equality as we are going to see now.

You can use == or === to compare for equality (or the opposite with != or !==). The main difference is that the first doesn't care which type the compared values have, so it returns true when both values are **equivalent**. When using ===, values have to also have the same type. If the type of value is different (or if it's the same type but different value) it will return false. It returns true when both values are **identical**.

```

console.log(3 == "3"); // true
console.log(3 === "3"); // false
console.log(3 != "3"); // false
console.log(3 !== "3"); // true
// Equivalent to false values (everything else is equivalent to true)
console.log("" == false); // true
console.log(false == null); // false (null is not equivalent to any boolean).
console.log(false == undefined); // false (undefined is not equivalent to any boolean).
console.log(null == undefined); // true (special JavaScript rule)
console.log(0 == false); // true
console.log({} >= false); // Empty object -> false
console.log([] >= false); // Empty array -> true

```

Other relational operators like less than (<), greater than (>), less or equal (<=),

and greater or equal (\geq) usually work with numbers or strings. When strings are compared with these operators, the character's order in Unicode is used to determine if its lower (situated before) or greater (situated after). Unlike addition operator (+), when one of the 2 values is a number, the other will be transformed to a number. Both need to be strings in order to make a comparison between strings.

```
console.log(6 >= 6); // true
console.log(3 < "5"); // true ("5" => 5)
console.log("adios" < "bye"); // true
console.log("Bye" > "Adios"); // true
console.log("Bye" > "adios"); // false. Uppercase letters go first always
console.log("ad" < "adios"); // true
```

Boolean operators

Boolean operators are: **negation** (!), **and** (&&), **or** (||). They're often used in combination with relational operations to form more complex conditions which at the end, will evaluate to true or false.

```
console.log(!true); // Prints false
console.log(!(5 < 3)); // Prints true (!false)

console.log(4 < 5 && 4 < 2); // Prints false (both conditions need to be true)
console.log(4 < 5 || 4 < 2); // Prints true (only one condition has to be true)
```

If you use the **and** or the **or** operators with values that are not booleans, a boolean won't be returned. Instead, the **or** operator will return the first value if it's equivalent to true (doesn't evaluate the second), or the second if the first one is false. The **and** operator will do the opposite (only evaluate the second when the first is true).

```
console.log(0 || "Hello"); // Prints "Hello"
console.log(45 || "Hello"); // Prints 45
console.log(undefined && 145); // Prints undefined
console.log(null || 145); // Prints 145
console.log("" || "Default"); // Prints "Default"
```

You can use the **double negation** !! to transform any value into boolean type. The first negation forces type casting into boolean and negates it. The second negation then restores its original (boolean) value.

```
console.log(!null); // Prints false
console.log(!undefined); // Prints false
console.log(!undefined === false); // Prints true
console.log(!""); // Prints false
console.log(!0); // Prints false
console.log(!"Hello"); // Prints true
```

The boolean operator || (or) can be used to simulate default values in a function for example.

```
function sayHello(name) {
  // If name is undefined or empty (""), it will assign "Anonymous" by default
  let sayName = name || "Anonymous";
  console.log("Hello " + sayName);
}
sayHello("Peter"); // Prints "Hello Peter"
sayHello(); // Prints "Hello anonymous"
```

Collections

Arrays

In JavaScript, an array is a type of object, and there are some differences with primitive types when comparing arrays, or passing them to functions as we'll see soon. You can create a new array by instantiating an object of the class **Array**. Arrays have no limited size.

The constructor can receive 0 arguments (empty array), 1 numeric argument (size of the array), or if it receives 2 or more numbers, or 1 or more different type of values, it will create an array with those values. Also, remember that JavaScript has no variable typing, so you can mix numbers, strings, booleans and objects in the array.

```
let a = new Array(); // Creates an empty array
a[0] = 13;
console.log(a.length); // Prints 1
console.log(a[0]); // Prints 13
console.log(a[1]); // Prints undefined
```

Note that when you access a position which has not been defined, you get undefined. The length of the array depends on what positions have been assigned. Lets see what happens with the array's length when you assign a position greater than its length and not consecutive.

```
let a = new Array(12); // Creates an array with 12 reserved positions
console.log(a.length); // Prints 12
a[20] = "Hello";
console.log(a.length); // Now it prints 21 (0-20). Positions 0-19 will have undefined value
```

We can also reduce an array's length directly modifying its **length** property. If we reduce its length, the positions greater than the new length will be deleted.

```
let a = new Array("a", "b", "c", "d", "e"); // Array with 5 values
console.log(a[3]); // Prints "d"
a.length = 2; // Positions 2-4 will be destroyed
console.log(a[3]); // Prints undefined
```

You can also create an array using **square brackets** notation instead of calling new Array(). The elements you put inside separated by a comma, will be the initial elements the array will hold.

```
let a = ["a", "b", "c", "d", "e"]; // Array with 5 values
console.log(typeof a); // Prints object
console.log(a instanceof Array); // Prints true. a is an Array instance
a[a.length] = "f"; // We insert a new element at the end
console.log(a); // Prints ["a", "b", "c", "d", "e", "f"]
```

Iterating through Arrays

To iterate through an array you can use while and for loops, using the index and incrementing it. Another version of the for loop, is the **for..in** loop. With this loop you

iterate through the indexes of an array or the properties of an object.

```
let ar = new Array(4, 21, 33, 24, 8);

let i = 0;
while(i < ar.length) { // Prints 4 21 33 24 8
  console.log(ar[i]);
  i++;
}

for(let i = 0; i < ar.length; i++) { // Prints 4 21 33 24 8
  console.log(ar[i]);
}

for (let i in ar) { // Prints 4 21 33 24 8
  console.log(ar[i]);
}
```

Iterating through object properties (we'll see that in the future):

```
let person = {
  name: "John",
  age: 45,
  phone: "65-453565"
};

/** Will print:
 * name: John
 * age: 45
 * phone: 65-453565
 */
for (let field in person) {
  console.log(field + ": " + person[field]);
}
```

Since ES2015, we can now iterate through the elements of an array or even through the characters of a string without using an index. The **for...of** loop (contrary to the **for...in** loop) is a real **for each** loop

```
let a = ["Item1", "Item2", "Item3", "Item4"];

for(let index in a) {
  console.log(a[index]);
}

for(let item of a) { // Does the same as above
  console.log(item);
}

let str = "abcdefg";

for(let letter of str) {
  if(letter.match(/[aeiou]/)) {
    console.log(letter + " is a vowel");
  } else {
    console.log(letter + " is a consonant");
  }
}
```

Array methods

Lets see how we can insert values at the beginning (**unshift**) and at the end (**push**) of an array. And two different ways of printing array's values in console.

```
let a = [];  
a.push("a"); // Inserts values at the end of the array  
a.push("b", "c", "d");  
console.log(a.valueOf()); // Prints ["a", "b", "c", "d"]. If omit valueOf() here, it will get called automatically  
a.unshift("A", "B", "C"); // Inserts new values at the beginning of the array  
console.log(a.toString()); // Prints A,B,C,a,b,c,d. toString() produces slightly different result than valueOf()
```

Now we'll see the opposite operation. Removing from the start (**shift**) and also from the end (**pop**) of the array. Note that these operations also return the value that's been removed.

```
console.log(a.pop()); // Prints and removes the last position -> "d"  
console.log(a.shift()); // Prints and removes the first position -> "A"  
console.log(a); // Prints ["B", "C", "a", "b", "c"]
```

We can print the elements of an array using **join()** instead of **toString()**. By default, it also returns a string containing all elements separated by comma. However, we can pass a custom separator to **join**.

```
let a = [3, 21, 15, 61, 9];  
console.log(a.join()); // Prints "3,21,15,61,9"  
console.log(a.join(" -#- ")); // Prints "3 -#- 21 -#- 15 -#- 61 -#- 9"
```

How do we concatenate 2 arrays?. Using **concat** (original array is not modified).

```
let a = ["a", "b", "c"];  
let b = ["d", "e", "f"];  
let c = a.concat(b);  
console.log(c); // Prints ["a", "b", "c", "d", "e", "f"]  
console.log(a); // Prints ["a", "b", "c"] . a hasn't been modified
```

The **slice** method returns a new subarray.

```
let a = ["a", "b", "c", "d", "e", "f"];  
let b = a.slice(1, 3); // Param 1 (start position -> included), param 2 (end position -> excluded)  
console.log(b); // Prints ["b", "c"]  
console.log(a); // Prints ["a", "b", "c", "d", "e", "f"]. Original array is not modified  
console.log(a.slice(3)); // One parameter. Returns from position 3 to the end -> ["d", "e", "f"]
```

splice does modify the original array, returning the deleted elements. It can also insert new elements from the defined position.

```
let a = ["a", "b", "c", "d", "e", "f"];  
a.splice(1, 3); // Remove 3 elements from position 1 ("b", "c", "d")  
console.log(a); // Prints ["a", "e", "f"]  
a.splice(1, 1, "g", "h"); // Remove 1 element at position 1 ("e"), and insert "g", "h" in that position  
console.log(a); // Prints ["a", "g", "h", "f"]  
a.splice(3, 0, "i"); // At position 3, we don't remove anything, and we insert "i"  
console.log(a); // Prints ["a", "g", "h", "i", "f"]
```

We can reverse the positions of an array using a method called **reverse**.

```
let a = ["a", "b", "c", "d", "e", "f"];
a.reverse(); // Reverses the original array
console.log(a); // Prints ["f", "e", "d", "c", "b", "a"]
```

Also, we can order the elements in an array using **sort**.

```
let a = ["Peter", "Anne", "Thomas", "Jen", "Rob", "Alison"];
a.sort(); // Orders the original array
console.log(a); // Prints ["Alison", "Anne", "Jen", "Peter", "Rob", "Thomas"]
```

But, what happens if we try to order elements that are not strings?. By default, it will order it by their string value (if they're objects it will try to call **toString()** method on them). To tell **sort** how to order elements of an array that contains elements which are not strings, we must pass a function that compares 2 values of the array, returning a negative number if the first value is lower (goes before), positive if it's greater (goes after), or 0 if they're equal. Let's see how this works with numbers.

```
let a = [20, 6, 100, 51, 28, 9];
a.sort(); // Orders the original array
console.log(a); // Prints [100, 20, 28, 51, 6, 9]
a.sort((n1, n2) => n1 - n2);
console.log(a); // Prints [6, 9, 20, 28, 51, 100]
```

Lets see an example with objects, ordering persons by age. We'll see how objects work in JavaScript soon.

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

  this.toString = function() { // Person toString() method
    return this.name + " (" + this.age + ")";
  }
}

let persons = [];
persons[0] = new Person("Thomas", 24);
persons[1] = new Person("Mary", 15);
persons[2] = new Person("John", 51);
persons[3] = new Person("Philippa", 9);

persons.sort((p1, p2) => p1.age - p2.age); // Age is a number
console.log(persons.toString()); // Prints: "Filippa (9),Mary (15),Thomas (24),John (51)"
```

Using **indexOf**, we can know if a value is present in the array. It will return the first position where it finds it, or -1 if it's not present. Using **lastIndexOf** is similar, but it will start searching from the end and will return the last position.

```
let a = [3, 21, 15, 61, 9, 15];
console.log(a.indexOf(15)); // Prints 2
console.log(a.indexOf(56)); // Prints -1. Not found
console.log(a.lastIndexOf(15)); // Prints 5
```

The method **every** will return a boolean indicating if **all** the elements in an array match a condition. It needs a function that returns a boolean indicating if each item meets a condition.


```
let a = [3, 21, 15, 61, 9, 54];
console.log(a.every(num => num < 100)); // Check if every number is lower than 100. Prints true
console.log(a.every(num => num % 2 == 0)); // Check if every number is even. Prints false
```

On the other hand, the method **some** is similar to **every**, but it returns true the moment it finds **any** member of the array that matches the condition.

```
let a = [3, 21, 15, 61, 9, 54];
console.log(a.some(num => num % 2 == 0)); // Check if any number is even. Prints true
```

We can also iterate through the elements of an array using the method **forEach**. Optionally, we can also track the current index if we add a second parameter, and receive also the entire array (to **modify** it for example) if we add a third parameter.

```
let a = [3, 21, 15, 61, 9, 54];
let sum = 0;
a.forEach(num => sum += num);
console.log(sum); // Prints 163

// index and array are optional
a.forEach((num, index, array) => console.log("Index " + index + " in [" + array + "] is " + num));
// Prints -> Index 0 in [3,21,15,61,9,54] is 3, Index 1 in [3,21,15,61,9,54] is 21, ...
```

To transform all the elements of an array, the method **map** receives a function that returns every element transformed. This method will return at the end a new array of the same size containing all the transformed elements.

```
let a = [4, 21, 33, 12, 9, 54];
console.log(a.map(num => num*2)); // Prints [8, 42, 66, 24, 18, 108]
```

To filter the elements of an array, and getting an array which contains only the elements that match a particular condition, we can use the method **filter**.

```
let a = [4, 21, 33, 12, 9, 54];
console.log(a.filter(num => num % 2 == 0)); // Prints [4, 12, 54]
```

The method **reduce** uses a function that accumulates a value. It receives each element of the array (second parameter) and the accumulated value (first parameter). After this function, you should pass the initial value. If you don't pass an initial value, the first element of the array will be used (and will start from the second).

```
let a = [4, 21, 33, 12, 9, 54];
console.log(a.reduce((total, num) => total + num, 0)); // Sums all elements in the array. Prints 133
console.log(a.reduce((max, num) => num > max ? num : max, 0)); // Maximum number of the array. Prints 54
```

If you prefer to do the same as **reduce** does, but backwards, use **reduceRight**.

```
let a = [4, 21, 33, 12, 9, 154];
// Begins with last number and subtracts all the other numbers
console.log(a.reduceRight((total, num) => total - num)); // Prints 75 (If we don't send an initial value it will start with the last position of the array)
```

New methods in ES2015

Let's take a look at the new methods present in the Array object:

- **Array.of(value)** → If we want to instantiate an array with just one value, and it's a number, using the new Array() constructor, we can't do it. Simply because just

one number means to create an array with that number of positions instead of creating an array with length = 1 holding that number in position 0.

```
let array = new Array(10); // Empty array (length 10)
let array = Array(10); // Same as before: Empty array (length 10)
let array = Array.of(10); // Array with length 1 -> [10]
let array = [10]; // Array with length 1 -> [10]
```

- **Array.from(array, func)** → Similar to the **map** method, creates an array from another array. It applies a transformation operation (arrow or anonymous function) to every item.

```
let array = [4, 5, 12, 21, 33];
let array2 = Array.from(array, n => n * 2);
console.log(array2); // [8, 10, 24, 42, 66]
let array3 = array.map(n => n * 2); // Same as Array.from
console.log(array3); // [8, 10, 24, 42, 66]
```

- **Array.fill(value)** → This method overrides all array values with a new value. It's a good option to initialize a newly created array with N positions.

```
let sums = new Array(6); // Array with 6 positions
sums.fill(0); // All positions set to 0
console.log(sums); // [0, 0, 0, 0, 0, 0]
```

```
let numbers = [2, 4, 6, 9];
numbers.fill(10); // Positions set to 10
console.log(numbers); // [10, 10, 10, 10]
```

- **Array.fill(value, start, end)** → This does the same as above but filling the array from the start index (included) until the end index (excluded is finished). If no end number is given, it will fill until the last position of the array.

```
let numbers = [2, 4, 6, 9, 14, 16];
numbers.fill(10, 2, 5); // Positions 2,3,4 set to 10
console.log(numbers); // [2, 4, 10, 10, 10, 16]
```

```
let numbers2 = [2, 4, 6, 9, 14, 16];
numbers2.fill(10, -2); // Last 2 positions set to 10
console.log(numbers2); // [2, 4, 6, 9, 10, 10]
```

- **Array.find(condition)** → Finds and returns the first value that meets the given condition (function that returns true or false). We can also use **findIndex** if, instead of the value, we want to return the position of that value.

```
let numbers = [2, 4, 6, 9, 14, 16];
console.log(numbers.find(num => num >= 10)); // Prints 14 (first value found >= 10)
console.log(numbers.findIndex(num => num >= 10)); // Prints 4 (numbers[4] -> 14)
```

- **Array.copyWithin(target, startwith)** → Copy the values of the array, starting from **startWith** index, into **target** index. Better seen with an example:

```
let numbers = [2, 4, 6, 9, 14, 16];
numbers.copyWithin(3, 0); // [0] -> [3], [1] -> [4], [2] -> [5]
console.log(numbers); // [2, 4, 6, 2, 4, 6]
```

- **entries(), keys(), values()** → Those methods return iterators to iterate through the array with [key,value] pairs (entries), keys or values. We can iterate using the **next()** method we saw with iterators, or using the **for..of** loop. We can even use the spread (...) operator.

```
let numbers = [2, 4, 6, 9, 14, 16];
for(let [key, val] of numbers.entries()) { // Iterate over pairs [index,value]
  console.log(key + " = " + val); // 0 = 2, 1 = 4, 2 = 6, 3 = 9, 4 = 14, 5 = 16
}
for(let key of numbers.keys()) { // Iterate over indexes
  console.log(key); // 0, 1, 2, 3, 4, 5
}
for(let val of numbers.values()) { // Iterate over values
  console.log(val); // 2, 4, 6, 9, 14, 16
}
console.log(...numbers.entries()); // [0, 2] [1, 4] [2, 6] [3, 9] [4, 14] [5, 16]
```

Rest and spread

Rest is the action to transform a group of parameters into an array, and **spread** is the opposite, taking out the elements of an array (or a string) into variables.

To **rest** parameters in a function, declare a parameter (it **always** has to be **last one**) and put three dots '...' in front of it. This parameter will transform automatically into an array containing all the parameters passed to the function. If, for example, the rest parameter is in the third position, it will have all the parameters sent except the first and second one.

```
function getAverageMark(...marks) {
  console.log(marks); // Prints [5, 7, 8.5, 6.75, 9] (it's an array)
  let total = marks.reduce((total, mark) => total + mark, 0);
  return total / marks.length;
}
console.log(getAverageMark(5, 7, 8.5, 6.75, 9)); // Prints 7.25

function printUserInfo(name, ...languages) {
  console.log(name + " knows " + languages.length +
    " languages: " + languages.join(" - "));
}

// Prints "Peter knows 3 languages: Java - C# - Python"
printUserInfo("Peter", "Java", "C#", "Python");
// Prints "Mary knows 5 languages: JavaScript - Angular - PHP - HTML - CSS"
printUserInfo("Mary", "JavaScript", "Angular", "PHP", "HTML", "CSS");
```

Spread is the opposite of rest. If we have a variable that contains an array, and we put the same three dots '...' in front of it, it will extract all its values. We can use this property for example with a method like **Math.max** that receives an indeterminate number of parameters (numbers) and returns the highest one.

```
let nums = [12, 32, 6, 8, 23];
console.log(Math.max(nums)); // Prints NaN (array is not valid)
console.log(Math.max(...nums)); // Prints 32 -> equivalent to Math.max(12, 32, 6, 8, 23)
```

We can also use this property if we need to copy an array (instead of referencing it and without having to iterate through it).

```
let a = [1, 2, 3, 4];
let b = a; // Reference the same array as 'a'
let c = [...a]; // New array -> equivalent to [1, 2, 3, 4]
```

Destructuring arrays

Destructuring is the action of extracting individual elements from an array (or properties from an object) directly into individual variables. We can also *destructure* a string into characters.

Let's see an example where we assign the first three elements of an array to three different variables using only one assignment.

```
let array = [150, 400, 780, 1500, 200];
let [first, second, third] = array; // Assigns the first three elements of the array
console.log(third); // Prints 780
```

What if we want to skip a value? Leave it empty inside the square brackets, it won't be assigned:

```
let array = [150, 400, 780, 1500, 200];
let [first, , third] = array; // Assigns the first and third element
console.log(third); // Prints 780
```

We can also assign all the remaining values to the last variable we put inside the square brackets by using **rest** (like in the previous section):

```
let array = [150, 400, 780, 1500, 200];
let [first, second, ...rest] = array; // rest -> array
console.log(rest); // Prints [780, 1500, 200]
```

If we want to assign more values than the array has and we don't want to get any undefined value, we can use default values like with function parameters:

```
let array = ["Peter", "John"];
let [first, second = "Mary", third = "Ann"] = array; // rest -> array
console.log(second); // Prints "John"
console.log(third); // Prints "Ann" -> default value
```

We can also destructure **nested arrays**:

```
let salaries = [["Peter", "Mary"], [24000, 35400]];
let [[name1, name2], [salary1, salary2]] = salaries;
console.log(name1 + " earns " + salary1 + "€"); // Prints "Peter earns 24000€"
```

And we can also use destructuring as **function parameters**. If we send an array when we call that function, it will be destructured into individual variables.

```
function printUserData([id, name, email], otherInfo = "None") {
  console.log("ID: " + id);
  console.log("Name: " + name);
  console.log("Email: " + email);
  console.log("Other info: " + otherInfo);
}
let userData = [3, "Peter", "peter@gmail.com"];
printUserData(userData, "He's not too smart");
```

Map

A Map is a collection that holds pairs of [key,value], values are accessed by using the corresponding key. In JavaScript, an object could be considered a type of Map but with some limitations (only strings and integers are allowed as keys).

The new **Map** collection allows us to use also any object as a key. We create that collection using the constructor **new Map()**, and we can use its **set**, **get** and **delete** methods to store, get or remove a value based on a key.

```
let person1 = {name: "Peter", age: 21};
let person2 = {name: "Mary", age: 34};
let person3 = {name: "Louise", age: 17};
```

```
let hobbies = new Map(); // Will store a person with an array of hobbies (string)
hobbies.set(person1, ["Tennis", "Computers", "Movies"]);
console.log(hobbies); // Map {Object {name: "Peter", age: 21} => ["Tennis", "Computers", "Movies"]}
```

```
hobbies.set(person2, ["Music", "Walking"]);
hobbies.set(person3, ["Boxing", "Eating", "Sleeping"]);
console.log(hobbies);
```

```
Map {Object {name: "Peter", age: 21} => ["Tennis", "Computers", "Movies"], Object
▼ {name: "Mary", age: 34} => ["Music", "Walking"], Object {name: "Louise", age: 17}
  => ["Boxing", "Eating", "Sleeping"]} ⓘ
  size: (...)
  ► __proto__: Map
  ▼ [[Entries]]: Array[3]
    ▼ 0: {Object => Array[3]}
      ▼ key: Object
        age: 21
        name: "Peter"
        ► __proto__: Object
      ▼ value: Array[3]
        0: "Tennis"
        1: "Computers"
        2: "Movies"
        length: 3
```

When we use an object as a key, we must know that we store a **reference to that object**. So, we **must use the same reference** to access or remove a value in that map.

```
console.log(hobbies.has(person1)); // true (reference to original object stored)
console.log(hobbies.has({name: "Peter", age: 21})); // false (different object!)
```

We can use the properties **size** to access the length of a Map and we can iterate through it using the **for..of** loop. On each iteration, an array with 2 positions **0 → key** and **1 → value** is returned.

```
console.log(hobbies.size); // Prints 3
hobbies.delete(person2); // Removes person2 from the Map
console.log(hobbies.size); // Prints 2
console.log(hobbies.get(person3)[2]); // Prints "Sleeping"
```

```
/** All of these print:
```

```

* Peter: Tennis, Computers, Movies
* Louise: Boxing, Eating, Sleeping */
for(let entry of hobbies) {
  console.log(entry[0].name + ": " + entry[1].join(", "));
}
for(let [person, hobArray] of hobbies) { // Better
  console.log(person.name + ": " + hobArray.join(", "));
}

hobbies.forEach((hobArray, person) => { // Also better
  console.log(person.name + ": " + hobArray.join(", "));
});

```

If we have an array that contains other arrays with 2 positions (key, value), we can create a Map from it directly.

```

let prods = [
  ["Computer", 345],
  ["Television", 299],
  ["Table", 65]
];

let prodMap = new Map(prods);
console.log(prodMap); // Map {"Computer" => 345, "Television" => 299, "Table" => 65}

```

Set

Set is like **Map**, but it doesn't store values (only keys). It can be viewed as a collection that ensures that there are **no duplicate values** (in arrays there can be duplicates). It uses **add**, **delete** and **has** → boolean methods to store, delete and see if a value exists.

```

let set = new Set();
set.add("John");
set.add("Mary");
set.add("Peter");
set.delete("Peter");
console.log(set.size); // Prints 2

set.add("Mary"); // Mary already exists
console.log(set.size); // Prints 2

// Iterate through the values
set.forEach(value => {
  console.log(value);
})

```

We can construct a Set from an array just to remove the duplicates from it.

```

let names = ["Jennifer", "Alex", "Tony", "Johnny", "Alex", "Tony", "Alex"];
let nameSet = new Set(names);
console.log(nameSet); // Set {"Jennifer", "Alex", "Tony", "Johnny"}

```