Introduction

JavaScript is one of the most widely used programming languages worldwide today. One of the main reasons for this is that most applications developed today are web applications, meaning they run within a browser environment. Web browsers are only capable of executing JavaScript code (and even programs compiled in WebAssembly).

The popularity of the **NodeJS** environment for running server-side applications using JavaScript has also helped, as has the ability to develop desktop applications with the **Electron** environment , or mobile web applications with access to native features with **Ionic** . For example, one of the most well-known and popular code editors, **Visual Studio Code** , is a great example of a web application made with Electron, but at first glance it is indistinguishable from a native desktop application.

A little history

To summarize the history of this language, it all started in 1995, when the now defunct NetScape, which at that time dominated the web browser market in the early days of the Internet, wanted to implement a Java virtual machine in its browser to add some dynamism to the primitive HTML documents, which were little more than a collection of text, links and some images (slow to load with those Jurassic modems).

However, they ultimately decided to create a language from scratch, one that was simple and executed by an interpreter built into the browser. Initially called LiveScript, it was later called LiveScript, but a few months later, Sun Microsystems, the owner of Java, agreed to allow the word Java to be used as part of the new language's name, and it was renamed JavaScript (although it has virtually nothing to do with Java).

Everything was going well until Internet Explorer began to gain a majority share of web browsers, as it was the default browser for the Windows operating system. To protect the language, the **ECMAScript** standard was created , based on which JavaScript implementations in different browsers should be created. However, Internet Explorer, in its attempt to establish itself as the de facto standard by deviating from the ECMAScript standard, made many web applications created to work in that browser incompatible with other browsers, further strengthening its monopoly and frustrating web developers.

Fortunately, thanks to the emergence of new browsers that gained significant market share (Opera, Firefox, Chrome, etc.), Microsoft was forced to back down and adapt to the standard. Fortunately, Internet Explorer is now history (its version 6 was particularly damaging).

JavaScript underwent a major overhaul in 2015 with the creation of the ES2015 standard (formerly known as ES6), which greatly evolved and modernized the language. Currently, the standard is updated annually (ES2016, ES2017, ES2018, etc.) with minor improvements to the language.

Introduction

Code editors

To start programming with JavaScript, the first thing you'll need is a code editor. You can download one or use an online version.

To open the console, we will first have to deploy the developer tools (F12, or right click -> inspect) and go to the corresponding tab:

To run small tests, we can use the browser's built-in JavaScript console. Here we can view the messages the application prints (information, debug messages, errors) and execute JavaScript code by simply typing it and executing it with the "Enter" key. It's worth noting that the code executed there can interact with the HTML currently being displayed, as well as call JavaScript functions in the current application.

Another option is the online code editors available. Although they aren't as customizable and are generally slower, they have the great advantage of allowing you to continue editing code from any other machine, as well as sharing the code with others who can then edit it.

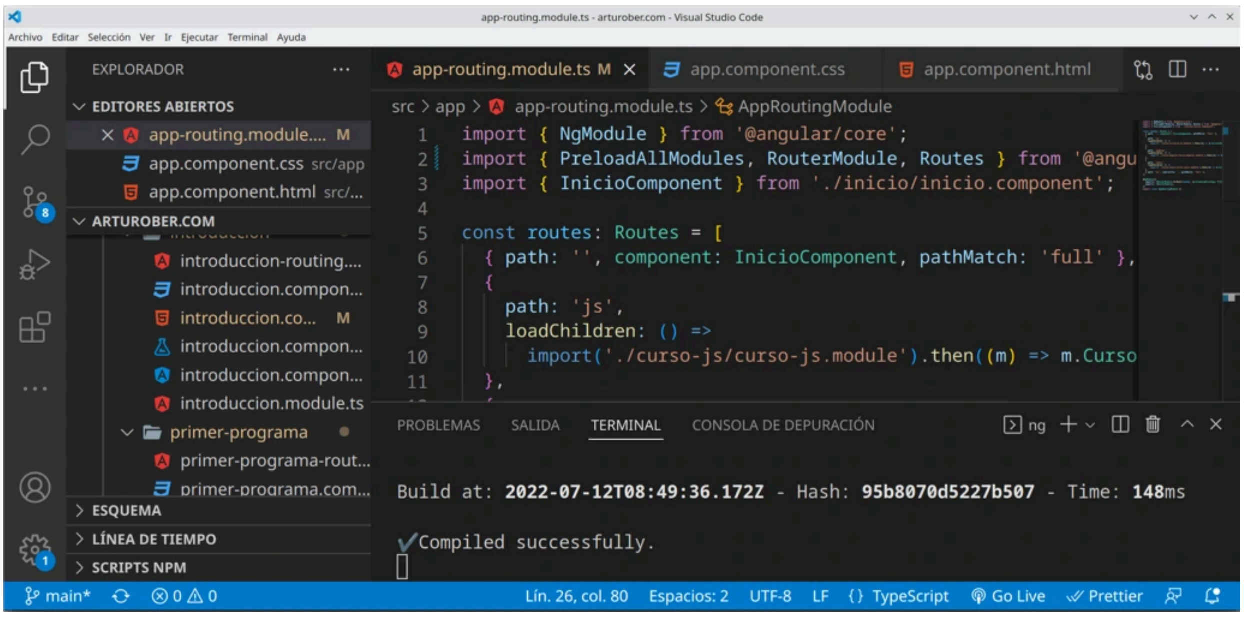
Examples of online editors would be:

- Plunker
- JSFiddle
- PlayCode
- StackBlitz

Finally, we can download an editor or IDE that has the necessary tools to work with HTML, CSS, and JavaScript code. We can download IDEs or Integrated Development Environments such as Visual Studio , WebStorm , Eclipse , or NetBeans , for example. These programs come with tools for development, project creation and management, and integrated debugging. In other words, they are very complete solutions. However, they are also heavier, consuming many more resources, than a standard code editor.

The most popular editor for developing applications with JavaScript (and many other languages) today is Visual Studio Code . Although they share a name, it's a different editor than Visual Studio. In this case, we're talking about an open-source, cross-platform editor that's much lighter than the aforementioned editors, but its capabilities can be incredibly extended through a plugin system. It could be considered a lightweight IDE rather than a simple code editor, but it's no less powerful.

Download Visual Studio Code



```
import { NgModule } from '@angular/core';
import { PreloadAllModules, RouterModule, Routes } from '@angu
import { InicioComponent } from './inicio/inicio.component';

const routes: Routes = [
  { path: '', component: InicioComponent, pathMatch: 'full' },
  {
    path: 'js',
    loadChildren: () =>
      import('./curso-js/curso-js.module').then((m) => m.Curso
  },
```

```
Build at: 2022-07-12T08:49:36.172Z - Hash: 95b8070d5227b507 - Time: 148ms

✓Compiled successfully.
```

Creating my first program

To run JavaScript code in a browser, we need to open an HTML document containing the JavaScript code. We can do this in two ways.

JavaScript inside an element

It is not the most recommended way, since we are mixing 2 languages in a single file, which complicates development and maintainability, especially in large programs, or with teams where each person is in charge of a development area (front, back, design, layout, ...).

It would be enough to include the JavaScript code we want to execute between two <script></script> tags. When the browser loads the HTML document and reaches the element in question, it will execute the JavaScript code.)

# index.html

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

To test this example, simply open the **index.html** file in a web browser (directly, we don't need a web server).

console.log() is used to write to the browser console what we pass to it (F12 to open the developer tools and see the result).

JavaScript in a separate file

If we separate the JavaScript code from the HTML, everything becomes much cleaner, especially as the program becomes more complex. To execute code in a separate JavaScript file, simply reference the file (relative path) from the HTML using the src attribute on an element.

# index.html

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

# index.js

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

Where to place the scripts

Generally, we want JavaScript code to run after processing (and rendering the HTML), i.e., at the very end. If we don't do this, it won't run until the code referenced by the element is processed.

By processing it at the end, we achieve two things. All HTML (DOM) elements will have been loaded into memory, so our JavaScript code will have access to them at any time during execution. The second advantage is that the HTML will load faster, giving the user the impression of better performance.

To achieve this we can place the scripts at the end of the HTML document (before closing the <body>), or we can use the **defer** attribute which achieves the same effect even if we place the element in the <header> of the page.

# index.html

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

# index.js

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

Browsers with JavaScript disabled

The <noscript> tag is used to place HTML code that will be rendered only when the browser doesn't support JavaScript or when it has been disabled. This tag is useful for informing the user that a website requires JavaScript to be enabled to function properly, for example.

# index.html

```
<!DOCTYPE>
<html>
    <head>
        <title>JS Example</title>
        <script src="index.js" defer></script>
    </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>
```

Comments

Comments are used to document code; that is, they are completely ignored when the program is run. When you insert a double slash '//', everything to the right of it up to the end of that line is considered a comment.

You can create a multi-line comment using the slash and asterisk, and end it with the same sequence in reverse. Example: /* Comment */.

# index.js

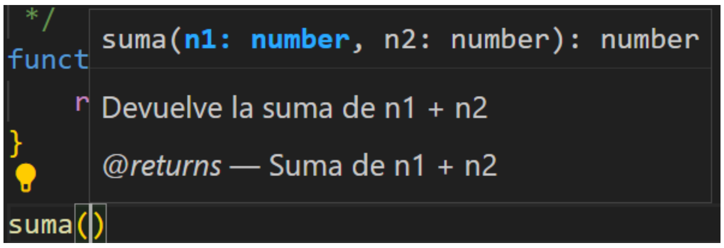
```
let nombre = "Juan"; // Almacenamos el nombre del usuario

/**
 * Vamos a calcular cuantas letras tiene el nombre del usuario
 * Aquí sigue el comentario...
 */
console.log(nombre.length);
```

If we begin a comment with a slash and two asterisks (/**), we can create a comment in **JSDoc** format , a **JavaDoc** -like tool for documenting our code more precisely, especially functions/methods and classes. This tool generates code documentation in HTML and other formats from these comments.

# index.js

```
/**
 * Devuelve la suma de n1 + n2
 * @param {number} n1
 * @param {number} n2
 * @returns {number} Suma de n1 + n2
 */
function suma(n1, n2) {
    return n1 + n2;
}
```

Editors like Visual Studio Code integrate with JSDoc comments and display the associated documentation when we type a function name, for example. They also show the parameter types and return values.



```
suma(n1: number, n2: number): number
Devuelve la suma de n1 + n2
@returns — Suma de n1 + n2
suma()
```

Variables

Variables are identifiers that store a value and can change during the execution of the program (hence their name) **.**

Declaring a variable

You can declare a variable using the keyword **let** (you can also use var, but it's not recommended for reasons we'll see later). The variable name should be in camelcase and can contain letters, numbers, $, and _ (but not begin with a number). In JavaScript, variables aren't declared with an explicit data type. The type can change internally depending on the value assigned to it. This means you could assign a string, and then a number. These types of languages are known as **weakly typed** .

The basic data types that we can assign to a variable are:

- **number** → Stores a number that can be either integer or decimal
- **string** → Stores a text string. It is represented in single, double, or backticks.
- **boolean**→ Stores a binary value that can be true or false
- **undefined**→ Special type that a variable has when it has not been assigned a value yet

# index.js

```
let v1 = "Hola Mundo!";
console.log(typeof v1); // Imprime -> string

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

v1 = true;
console.log(typeof v1); // Imprime -> boolean

let v2;
console.log(typeof v2); // Imprime -> undefined
```

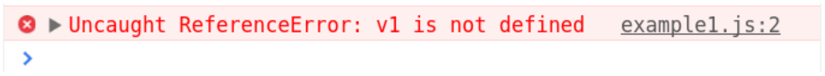
The typeof statement returns the type of the variable (depends on the assigned value)

What happens if we forget to use **let** ? JavaScript will automatically create that variable as a **global variable** , which will be accessible throughout the code. This is NOT recommended because global variables are dangerous. We'll explore the scope of variables (global and local) later.

To prevent us from forgetting to declare the variable, we can use a special declaration: **'use strict'** at the beginning of our JS file. This way, we won't be able to assign a value or access a variable that hasn't been previously declared with let.

# index.js

```
'use strict';
v1 = "Hola Mundo";
```



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

Constants

When a variable is not going to change its value throughout a function or block, or when we want to define an immutable global value (for example the number PI), it is recommended to declare it as a constant with the reserved word const instead of using let.

Global constants are often named in all capital letters using *SNAKE_CASE* notation . If we create a constant local to a piece of code, the same rule applies to naming it as with variables. It's worth noting that it's often recommended to store all values that have an initial assignment and that we know won't change over time as constants. This way, we prevent accidental modifications.

# index.js

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

Number

In JS, there's no distinction between integers and decimals. The data type for any number is **number** . Additionally, numbers can be represented in exponential notation, in addition to classic decimal notation.

# index.js

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

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

Numbers are objects

In JavaScript, everything is an object, even values of basic or primitive types, which isn't the case in other languages (Java, C++, etc.). This can be verified by adding a period after the number (if it's an integer, we must wrap it in parentheses), which allows us to access some methods or properties.

# index.js

```
console.log(3.32924325.toFixed(2)); // Imprime 3.33
console.log(5435.45.toExponential()); // Imprime 5.43545e+3
console.log((3).toFixed(2)); // Imprime 3.00
```

There is also the **Number** class , from which we can access quite useful static properties and methods.

# index.js

```
console.log(Number.MIN_VALUE); // 5e-324 (número más pequeño)
console.log(Number.MAX_VALUE); // 1.7976931348623157e+308 (Número más grande)
```

There are also special values for numbers outside the range (Infinity and -Infinity). To determine whether a number is finite, we can compare it to the value **Infinity** or **-Infinity** . There is also a function called **isFinite(value)** that returns false for values that are Infinity or -Infinity.

# index.js

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

let number = Number.POSITIVE_INFINITY / 2; // Todavía Infinity!!
if(Number.isFinite(number)) {
    console.log("El número " + number + " es finito");
} else {
    console.log("El número es infinito!");
}
```

We should not use Number as a constructor, e.g. **new Number(34)** , to create numbers, since it is slower, and also generates erroneous comparisons between numbers by not using the primitive value, but the memory reference, to compare them.

# index.js

```
let x = new Number(500);
let y = new Number(500);
console.log(x == y); // false
console.log(x === y); // false

let a = 500;
console.log(x == a); // true
console.log(x === a); // false

console.log(typeof a); // number
console.log(typeof x); // object

// Sin embargo, ambos tienen las mismas propiedades y métodos
console.log(a.__proto__) // Number {...}
console.log(x.__proto__) // Number {...}
```

Conversion to type 'number'

You can convert a data type from another type to a number using the global Number(value) function . You can also add a **'+'** sign before the variable to force the conversion, as the '+' sign can only be applied to a value of this type.

# index.js

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

console.log(s1 + s2); // 3214 (concatena cadenas)
console.log(Number(s1) + Number(s2)); // 46
console.log(+s1 + +s2); // 46

console.log(+true); // true equivale a 1
console.log(+false); // false equivale a 0
```

Special value NaN

When we try to convert a piece of data to a number and it's not possible (for example, a string that doesn't represent a number), the special value **NaN** (Not a Number) is returned. Any arithmetic operation on this value also produces another NaN value.

To check if the result of a conversion or operation is NaN, we have the Number.isNaN(value) function, which will return a Boolean value indicating whether it is a valid number or not.

# index.js

```
let nombre = "Juan";
let num = +nombre;
console.log(num); // NaN
console.log(Number.isNaN(num)); // true
console.log(num + 23); // NaN (no se puede operar)
```

Boolean

As in other programming languages, the boolean data type can store the logical values **true** or **false** .

# index.js

```
let a = true;
console.log(typeof a); // boolean
```

Like the number type, **boolean** data types are also objects. However, they have no methods or properties beyond those common to all objects in the language (inherited from **object**), as they are such a simple data type.

Conversion to 'boolean' type

Converting a data item to a Boolean is done using the **Boolean(value)** function . You can also use !! (double negation) before the value to force the conversion, since that operator only works with Booleans. It first converts and then negates the value, so we must use another negation to obtain the equivalent original value.

These values are equivalent to **false** : empty string (""), null, undefined, NaN, and 0. Any other value should return **true** .

# index.js

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

console.log(Boolean(v)); // false
console.log(!!s); // true
console.log(!!0); // false
console.log(!!24); // true
console.log(!!NaN); // false
console.log(!!null); // false
console.log(!!undefined); // false
console.log(!![]); // true (array vacío)
console.log(!!{}); // true (objeto vacío)
```

String

String values are text strings that are usually enclosed in single quotes or double quotes. The + operator can be used to generate a new string from the concatenation of strings.

# index.js

```
let s1 = "This is a string";
let s2 = 'This is another string';

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

When a string is enclosed in double quotes, we can use single quotes inside, and vice versa. However, if you want to put double quotes inside a string declared within double quotes, you need to escape them (\") to avoid closing the string. The same applies to combining single quotes.

# index.js

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

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

Template literals

Since ES2015, JavaScript supports **multiline** strings with **variable substitution** . To do this, we wrap the text in `` ` `` (backquote) characters instead of single or double quotes.

Line breaks or spaces, tabs, etc., are printed as represented in the string (without the need for special characters like \ or \t). Any variable (or any expression that returns a value) would go inside **${}** if you want to replace it with its value.

# index.js

```
let num = 13;
console.log(`Ejemplo de cadena multi-línea,
el valor de num es: ${num}`);
```

Operations with strings

Like other data types, strings are objects and have some useful methods we can use. None of these methods change the value of the original variable unless you reassign it.

# index.js

```
let s1 = "This is a string";
// Longitud de una cadena
console.log(s1.length); // 16

// Obtener un carácter de la cadena
console.log(s1.charAt(0)); //  "T" -> equivale a s1[0]

// Obtiene el índice donde aparece por primera vez la subcadena a buscar (-1 si no lo encuentra)
console.log(s1.indexOf("s")); // 3

// Índice donde la subcadena aparece por última vez (o -1)
console.log(s1.lastIndexOf("s")); // 10

// Devuelve un array con las coincidencias encontradas en base a una expresión regular
console.log(s1.match(/.s/g)); // ["is", "is", " s"]

// Índice de la primera coincidencia con la expresión regular
console.log(s1.search(/[aeiou]/)); // 2 (posición de la primera vocal)

// Reemplaza lo que coincida con una expresión regular (o string) por otra cosa
// En una expresión regular, la opción final g implica todas las coincidencias (si no, solo la primera)
console.log(s1.replace(/i/g, "e")); // "Thes es a streng"

// Devuelve una subcadena (posición inicial: incluida, posición final: excluida)
// A diferencia de substring, aquí podemos usar índices negativos para contar desde el final
console.log(s1.slice(5, 7)); // "is"
```

```
// Igual que slice
console.log(s1.substring(5, 7)); // "is"

// Transforma la cadena a minúsculas
console.log(s1.toLocaleLowerCase()); // "this is a string"

// Transforma la cadena a mayúsculas
console.log(s1.toLocaleUpperCase()); // "THIS IS A STRING"

// Borra los espacios, tabuladores y saltos de línea a principio y final
// También tenemos trimEnd (solo final) y trimStart (solo principio)
console.log("   String with spaces   ".trim()); // "String with spaces"
```

Converting to 'string' type

You can convert a data item to a string using the global String(value) function . Alternatively, you can concatenate it with an empty string, which will force the conversion if it's not a string.

# index.js

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

// Si se utiliza el operador '+' con un operando de tipo string, el otro se convierte automáticamente y se concatena
console.log(String(32) + 14); // 3214
console.log("" + 32 + 14); // 3214
```

Undefined

The special value **undefined** , whose type is also undefined, is assigned to variables, function parameters, array positions, etc. when they have been defined but have not yet been given a value.

When accessing nonexistent array positions, JavaScript won't throw an error. Instead, it returns the value undefined.

# index.js

```
let a;
console.log(a); // undefined

let array = [1,2,3];
console.log(array[6]); // undefined

function f(nombre) {
    console.log(`Hola ${nombre}`);
}

f("Juan"); // Hola Juan
f(); // Hola undefined
```

The way to check whether a variable has a value is to compare it directly with the undefined value (using strict comparison ===). You can also check the type with the **typeof** operator . With the second method, if the variable doesn't exist, it won't generate an error (with the first, it will).

# index.js

```
function f(nombre) {
    if(nombre === undefined) {
        console.error("No has introducido un nombre!");
    } else {
        console.log(`Hola ${nombre}`);
    }
}

f("Juan"); // Hola Juan
f(); // // ERROR: No has introducido un nombre!
```

undefined vs null

It's important not to confuse **undefined** with **null** . The latter is a value that must be assigned intentionally and refers to an object-type value but doesn't currently point to any memory location. They're similar, but not the same.

# index.js

```
let a; // undefined
let b = null;

console.log(a == b); // true (son equivalentes)
console.log(a === b); // false (diferente tipo de datos)
```

Operadores

Addition and concatenation '+'

This operator can be used to add numbers or concatenate strings. But what happens if we try to add a number to a string, or something other than a number or string? Let's see what happens:

# index.js

```
console.log(4 + 6); // 10
console.log("Hello " + "world!"); // "Hello world!"
console.log("23" + 12); // "2312"
console.log("42" + true); // "42true"
console.log("42" + undefined); // "42undefined"
console.log("42" + null); // "42null"
console.log(42 + "hello"); // "42hello"
console.log(42 + true); // 43 (true => 1)
console.log(42 + false); // 42 (false => 0)
console.log(42 + undefined); // NaN (undefined no se puede convertir a número)
console.log(42 + null); // 42 (null => 0)
console.log(13 + 10 + "12"); // "2312" (13 + 10 = 23, 23 + "12" = "2312")
```

When an operand is a string, concatenation will always be performed, so an attempt will be made to transform the other value into a string (if it isn't). Otherwise, an addition will be attempted (converting non-numeric values to numbers). If the conversion to a number fails, NaN (Not a Number) will be returned.

To check if a number is NaN, you can use the Number.isNaN(value) method, which returns a boolean (true if it is NaN, that is, one of the operands could not be converted to a number).

Arithmetic operators

Other arithmetic operators are: **subtraction (-)** , **multiplication (*)** , **division (/)** , **remainder (%)** , and **power (**)** . These operators always operate on numbers, so every operand that is not of type *number* must be converted to a number.

# index.js

```
console.log(4 * 6); // 24
console.log(4 ** 2); // 16
console.log("Hello " * "world!"); // NaN
console.log("24" / 12); // 2 (24 / 12)
console.log("42" * true); // 42 (42 * 1)
console.log("42" * false); // 0 (42 * 0)
console.log("42" * undefined); // NaN
console.log("42" - null); // 42 (42 - 0)
console.log(12 * "hello"); // NaN ("hello" no se puede convertir a number)
console.log(13 * 10 - "12"); // 118 ((13 * 10) - 12)
```

Unary operators

In JavaScript we have the preincrement (++variable), **postincrement** (variable++), **predecrement** (--variable) and **postdecrement** (variable--) operators .

# index.js

```
let a = 1;
let b = 5;
console.log(a++); // Imprime 1 e incrementa a = 2
console.log(++a); // Incrementa a = 3, e imprime 3
console.log(++a + ++b); // Incrementa a = 4 y b = 6. Suma (4+6), e imprime 10
console.log(a-- + --b); // Decrementa b = 5. Suma (4+5). Imprime 9. Decrementa a = 3
```

Relational operators

The comparison operator compares two values and returns a boolean (true or false). These operators are practically the same as in most programming languages, except for a few, which we will see below.

We can use **==** or **===** to compare for equality (or the opposite **: !=** , **!==**). The main difference is that the former (==), regardless of the data type being compared, compares whether the values are **equal** . Internally, it converts one of the values to the type of the other to compare them.

When using === (or !==), the values must also be of the same type. If the value type is different (or if it's the same data type but a different value), it returns false. It returns true when both values are **identical** .

# index.js

```
console.log(3 == "3"); // true
console.log(3 === "3"); // false
console.log(3 != "3"); // false
console.log(3 !== "3"); // true

console.log("" == false); // true
console.log(false == null); // false (null no equivale a ningún booleano).
console.log(false == undefined); // false (undefined no equivale a ningún booleano).
console.log(null == undefined); // true (null y undefined si equivalen entre ellos)
console.log(0 == false); // true
console.log({} == false); // Empty object -> false
console.log([] == false); // Empty array -> true
```

Because of this language feature, we can simply evaluate any value in a conditional structure. JavaScript will retrieve its Boolean equivalent (true or false) and evaluate the condition based on that.

# index.js

```
function sayHello(name) {
  if(name) { // name no está vacío, ni es undefined, ni null
    console.log(`Hello ${name}`);
  } else {
    console.error('Name cannot be empty');
  }
}
```

Other relational operators for numbers or strings are: less than (**<**), greater than (**>**), less than or equal to (**<=**), and greater than or equal to (**>=**). When comparing a string with these operators, each character is compared, and their position in the Unicode encoding is compared to determine whether it is less than (placed before) or greater than (placed after). Unlike the addition operator (+), when one of the two operands is a number, the other will be transformed into a number for comparison. In order to compare as a string, both operands must be strings.

# index.js

```
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. Las letras mayúsculas van antes
console.log("ad" < "adios"); // true
```

Logical operators

The logical operators are negation (**!**), and (**&&**), or (**||**). These operators are usually used in combination with relational operators to form a more complex condition. The final result is always **true** or **false** .

# index.js

```
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)
```

You can use either the && or || operator with non-Boolean values. The expression will return one of the two values. The || operator returns the first value if it is true, or the second otherwise. The && operator does the opposite (it will return the second value only if the first is true).

# index.js

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

Zero Coalescence Operator (??)

The null coalescing operator (??), introduced in ECMAScript 2020, provides a precise and safe way to assign default values. Its main function is to return the right-hand operand only when the left-hand operand is **null** or **undefined** . Otherwise, it returns the left-hand operand.

The main difference with the || operator is that the ?? operator interprets "", 0, and false as valid values when they are on the left. That is, it only considers undefined and null values, as mentioned above.

# index.js

```
let cantidad = 0;

// Usando el operador OR (||)
const cantidadConOR = cantidad || 50;
console.log(cantidadConOR); // 50 (porque 0 equivale a false)

// Usando el operador de coalescencia nula (??)
const cantidadConNullish = cantidad ?? 50;
console.log(cantidadConNullish); // 0 (porque 0 no es null ni undefined)

let nombre = null;
console.log(nombre ?? "Anónimo"); // "Anónimo" (nombre es null)
```

Control structures

Conditional structures

if structure

The if statement behaves like most programming languages. It evaluates a logical condition, returning a Boolean value as a result, and if true, executes the code inside the if statement. Optionally, we can add an else if statement and an else statement.

index.js

```
let price = 65;

if(price < 50) {
    console.log("This is cheap!");
} else if (price < 100) {
    console.log("Not so cheap...");
} else {
    console.log("Too expensive!");
}
```

Ternary operator

There is also the ternary operator **'?'** , which returns one value or another depending on a condition. If the condition is true, it returns the first value, and if it's false, it returns the second.

index.js

```
const socio = true;
const cuota = socio ? 50 : 100;
console.log(`Vas a pagar ${cuota}€`); // Vas a pagar 50€
```

Switch structure

The switch structure behaves similarly to other programming languages. As we know, a variable is evaluated and the block corresponding to its value is executed (it can be a number, string, etc.). Typically, a break statement must be added at the end of each block, since otherwise, the instructions in the next block would continue to be executed. An example where two values would execute the same block of code:

index.js

```
let userType = 1;

switch(userType) {
    case 1:
    case 2: // Tipos 1 y 2 entran aquí
        console.log("You can enter");
        break;
    case 3:
        console.log("You don't have enough permissions");
        break;
    default: // Ninguno de los anteriores
        console.error("Unknown user type!");
}
```

Iterative structures (loops)

While loop

We have the while loop that evaluates a condition and repeats itself over and over again until the condition is false (or if the condition is false from the start, it does not enter to execute the block of instructions it contains).

index.js

```
let value = 1;

while (value <= 5) { // prints 1 2 3 4 5
    console.log(value++);
}
```

For loop

In a **for** loop you can initialize one or more variables and also execute multiple update statements (increment, decrement) in each iteration by separating them with commas.

index.js

```
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
*/
```

Break and continue

Within a loop, we can use the **break** and **continue** statements . The first will exit the loop immediately after execution, and the second will go to the next iteration, skipping the rest of the statements in the current iteration (it executes the corresponding increment if we are inside a for loop).

Functions

In JavaScript, we declare functions using the keyword **function** before the function name. The parameters we pass to the function go inside the parentheses after the function name. Once the function is defined, we declare its body between the braces. Function names (like variable names) are usually written in **camelcase** with the first letter lowercase.

To execute the function, simply add its name to the code and pass the values for each parameter in parentheses. Even if it doesn't receive any values, the parentheses are required during the call.

# index.js

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

sayHello("Tom"); // "Hello Tom"
```

You don't need to have the function declared before the call; this is because the JavaScript execution engine makes two passes. First, it processes the variable and function declarations, and then it executes the program.

We can call a function by passing more or fewer parameters than specified in the declaration. If we pass more parameters, the excess will be ignored; if we pass fewer parameters, the unused ones will be assigned the value **undefined** .

# index.js

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

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

Return of values

We can use the return keyword to return a value in a function. If we try to access the value of a function that doesn't return anything, we'll get **undefined** .

# index.js

```
function totalPrice(priceUnit, units) {
    return priceUnit * units;
}

let total = totalPrice(5.95, 6);
console.log(total);  // 35.7
```

It's worth remembering that as soon as the return statement is executed , the function immediately terminates by returning the corresponding value. It doesn't make sense to have code statements after the return statement unless there's a conditional structure that discriminates between several options when returning a value.

Anonymous functions

The way to declare an anonymous function is to not assign it a name. We can assign this function as a value to a variable, since it's a value type (such as a string or number), so it can be assigned to (or referenced from) multiple variables. It's used just like a classic function.

# index.js

```
let totalPrice = function(priceUnit, units) {
    return priceUnit * units;
}

console.log(typeof totalPrice); // "function" (tipo de datos del identificador totalPrice)

console.log(totalPrice(5.95, 6)); // 35.7
let getTotal = totalPrice; // Podemos crear varios identificadores que referencien a la misma función
console.log(getTotal(5.95, 6));  // 35.7
```

Arrow (or lambda) functions

One of the most important features added in ES2015 was the ability to use lambda (or arrow) functions. Other languages like C#, Java, and others also support them. These expressions offer the ability to create anonymous functions, but with some advantages.

Let's see the differences between creating two equivalent functions (one anonymous and one lambda that do the same thing):

# index.js

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

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

When declaring an arrow or lambda function, the keyword **function** is not used. If only one parameter is passed, the parentheses can be omitted. The parameters must be followed by an arrow (=>) and the function's contents.

# index.js

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

If there's only one statement inside the arrow function, we can omit the braces '{}'. In this case, we must omit the keyword **return** , since it does so implicitly (it returns the result of that statement). If there's more than one statement, we use the braces, and it behaves like a normal function. Therefore, if it returns something, we must use the keyword return.

# index.js

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

The most important difference between the two types of functions is the behavior of the this keyword . But we'll cover that in the Object-Oriented Programming section.

Default parameters

If a parameter is declared in a function and no value is passed to it when we call it, its value is set to **undefined** .

# index.js

```
function sayHello(name) {
    console.log("Hello! I'm " + name);
}

sayHello(); // Prints "Hello! I'm undefined"
```

One solution used to set a default value was to use the '||' (or) operator, so that if it evaluates to undefined (equivalent to false), another default value is assigned.

# index.js

```
function sayHello(name) {
    name = name || "Anonymous";
    console.log("Hello! I'm " + name);
}
```

Since ES2015 we have the option to set a **default value** directly.

# index.js

```
function sayHello(name = "Anonymous") {
    console.log("Hello! I'm " + name);
}

sayHello(); // "Hello! I'm Anonymous"
```

We can also assign a default value based on an expression.

# index.js

```
function getTotalPrice(price, tax = price * 0.07) {
    return price + tax;
}

console.log(getTotalPrice(100)); // 107
```

Scope of variables

Global variables

When a variable is declared in the main block (outside of any function), it is created as global. Variables not declared with the **let** keyword are also global (unless we're using **strict mode** , which doesn't allow this).

A global variable can be accessed from anywhere in the code. This is dangerous from the perspective of preventing side effects when its value is modified, so it is recommended to avoid this type of variable unless strictly necessary.

# index.js

```
let global = "Hello";

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

changeGlobal();
console.log(global); // "GoodBye"
```

If we forget to include the word **let** when declaring a variable inside a function, if we don't use **strict mode** (it's recommended that you do), a global variable will be created. Strict mode won't let you do that.

# index.js

```
'use strict';

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

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

Local variables

All variables declared within a function or other type of block (if, while, etc.) are local to that block. That is, they can only be accessed from the block where they are declared (including other blocks within it), but not from outside.

# index.js

```
function setPerson() {
    let  person = "Peter"; // Variable local
}

setPerson();
console.log(person); // Error → Uncaught ReferenceError: person is not defined
```

If a global variable with the same name exists, the local variable will not update the value of the global variable.

# index.js

```
function setPerson() {
    let person = "Peter";
}

let person = "John";
setPerson();
console.log(person); // Imprime John
```

Arrays

In JavaScript, arrays are a type of object and are therefore instances of the Array class. These arrays are dynamic, so they don't have a fixed size. That is, we can create them with an initial size and then add more elements.

The constructor can take 0 parameters (an empty array), 1 number (the size of the array), or otherwise, an array would be created with the elements received. We must keep in mind that in JavaScript, an array can contain different data types at the same time: number, string, boolean, object, etc.

# index.js

```
let a = new Array(); // Crea un array vacío
a[0] = 13; // Asigna la primera posición del array
console.log(a.length); // Imprime 1
console.log(a[0]); // Imprime 13
console.log(a[1]); // Imprime undefined (posición no asignada aún)
```

Note that when you access an array position that hasn't been defined, it returns **undefined** . The length of an array depends on the last position assigned. Let's look at an example of what happens when you assign a position greater than the length and that isn't consecutive to the last assigned value.

# index.js

```
let  a = new Array(12); // Crea un array de tamaño 12
console.log(a.length); // Imprime 12
a[20] = "Hello";
console.log(a.length); // Ahora imprime 21 (0-20). Las posiciones 0-19 tendrán el valor undefined
```

We can reduce the length of an array by directly modifying the array's length property (**length**). If we reduce the length of an array, positions greater than the new length will be considered undefined (deleted).

# index.js

```
let  a = new Array("a", "b", "c", "d", "e"); // Array con 5 valores
console.log(a[3]); // Imprime "d"
a.length = 2; // Posiciones 2-4 serán destruidas
console.log(a[3]); // Imprime undefined
```

You can create an array directly using bracket notation instead of using new Array(). The values you put inside (optional), separated by commas, will be the initial elements.

# index.js

```
let  a = ["a", "b", "c", "d", "e"]; // Array de tamaño 5, con 5 valores inicialmente
console.log(typeof a); // Imprime object
console.log(a instanceof Array); // Imprime true. a es una instancia de array
a[a.length] = "f"; // Insertamos un nuevo elemento al final
console.log(a); // Imprime ["a", "b", "c", "d", "e", "f"]
```

Traversing arrays

We can iterate through an array using the classic **while** and **for** loops , creating a counter for the index that we'll increment with each iteration. Another version of the for loop is the **for..in** loop . With this loop, we can iterate through the indices of an array without needing to create a counter variable.

# index.js

```
let  ar = [4, 21, 33, 24, 8];

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

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

for (let i in ar) { // Imprime 4 21 33 24 8
```

```
    console.log(ar[i]);
}
```

In addition to the previous loops that traverse array indices, there is another type of loop (**for..of**) that allows you to traverse array values directly without needing the index. This would be equivalent to the foreach loop in other programming languages. This type of loop can also be used to traverse characters in a string.

# index.js

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

for(let item of a) {
    console.log(item);
}

let str = "abcdefg";

for(let letter of str) {
    if(letter.match(/^[aeiou]$/)) {
        console.log(letter + " es una vocal");
    } else {
        console.log(letter + " es una consonante");
    }
}
```

Array methods

As objects of the Array class, arrays have a series of methods that we can use to perform basic operations. It's important to differentiate between methods that modify the original array and those that generate a new array without modifying the original.

Insert and delete at the beginning and end

To insert and delete values at the beginning and end of the array we have the following methods (All these methods modify the original array):

- **unshift** : Adds one or more values to the beginning of the array
- **push** : Adds one or more values to the end of the array
- **shift** : Removes and returns the first value from the array
- **pop** : Removes and returns the last value from the array

# index.js

```
let  a = ["a"];
a.push("b", "c", "d"); // Inserta nuevos valores al final del array
console.log(a); // Imprime ["a", "b", "c", "d"]
a.unshift("A", "B", "C"); // Inserta nuevos valores al principio del array
console.log(a); // Imprime ["A", "B", "C", "a", "b", "c", "d"]

console.log(a.pop()); // Imprime y elimina la última posición → "d"
console.log(a.shift()); // Imprime y elimina la primera posición → "A"
console.log(a); // Imprime ["B", "C", "a", "b", "c"]
```

Convert an array to a string

We can convert an array to a string using **join()** in addition to **toString()** . By default, it returns a string with all elements separated by commas. However, we can specify the separator to print by passing it as a parameter.

# index.js

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

Convert a string to an array

To perform the reverse operation, we can use **Array.from()** , which returns a new array containing the elements of any iterable collection, including a **string** . In the latter case, it would generate an array that would store a letter of the string in each position.

# index.js

```
let s = "tenedor";
```

```
console.log(Array.from(s)); // ['t', 'e', 'n', 'e', 'd', 'o', 'r']
```

Concatenate arrays

To concatenate the contents of two or more arrays, we use the **concat()** method . You can pass one or more arrays as a parameter, and their contents will be concatenated with the contents of the main array. It's important to note that the original array isn't modified; instead, a new array is generated and returned with all the elements concatenated.

# index.js

```
let  a = ["a", "b", "c"];
let  b = ["d", "e", "f"];
let  c = a.concat(b);
console.log(c); // Imprime ["a", "b", "c", "d", "e", "f"]
console.log(a); // Imprime ["a", "b", "c"] . El array original no ha sido modificado
```

Get subarrays

The slice method returns a subarray. We typically pass two parameters: the start position (included) and the end position (excluded). These positions can be negative (counting from the end). You can also omit the end position if you want to get to the end of the array. It's important to note that this method doesn't modify the original array.

# index.js

```
let a = ["a", "b", "c", "d", "e", "f"];
let b = a.slice(1, 3); // (posición de inicio → incluida, posición final → excluida)
console.log(b); // Imprime ["b", "c"]
console.log(a); // Imprime ["a", "b", "c", "d", "e", "f"]. El array original se modifica
console.log(a.slice(3)); // Un parámetro. Devuelve desde la posición 3 al final → ["d", "e", "f"]
```

Insert and delete at intermediate positions

To add and remove elements at intermediate positions in the array, we have the **splice** method . We pass this method as a parameter the position we want to work on, the number of elements to delete from there (it can be 0), and the number of elements to insert in their place (optional). It's important to know that this method modifies the original array. If we want the result to be returned as a new array without modifying the original, we can use the equivalent method **toSpliced** (standard as of version ES2023).

# index.js

```
let  a = ["a", "b", "c", "d", "e", "f"];
a.splice(1, 3); // Elimina 3 elementos desde la posición 1 ("b", "c", "d")
console.log(a); // Imprime ["a", "e", "f"]
a.splice(1,1, "g", "h"); // Elimina 1 elemento en la posición 1 ("e"), e inserta "g", "h" en esa posición
console.log(a); // Imprime ["a", "g", "h", "f"]
a.splice(3, 0, "i"); // En la posición 3, no elimina nada, e inserta "i"
console.log(a); // Imprime ["a", "g", "h", "i", "f"]

let a2 = a.toSpliced(2, 1, "H");
console.log(a); // ["a", "g", "h", "i", "f"] -> No modificado
console.log(a2); // ["a", "g", "H", "i", "f"]
```

Invert array positions

We can reverse the order of the array using the **reverse** method . This modifies the original array. If, on the other hand, we want to return a reversed version of the array without modifying the original, we have the new (ES2023) **toReversed** method .

# index.js

```
let  a = ["a", "b", "c", "d", "e", "f"];
console.log(a.toReversed()); // ["f", "e", "d", "c", "b", "a"]
console.log(a); // ["a", "b", "c", "d", "e", "f"] -> Array original no modificado

a.reverse(); // Invierte el orden del array original
console.log(a); // ["f", "e", "d", "c", "b", "a"]
```

Sort an array

También, podemos ordenar los elementos de un array usando el método **sort**. Al igual que ocurría con reverse, este método modifica el array original. En la versión ES2023 se ha introducido el método **toSorted**, que devuelve un nuevo array ordenado sin modificar el original.

# index.js

```
let  a = ["Peter", "Anne", "Thomas", "Jen", "Rob", "Alison"];
console.log(a.toSorted()); // ["Alison", "Anne", "Jen", "Peter", "Rob", "Thomas"]
console.log(a); // ["Peter", "Anne", "Thomas", "Jen", "Rob", "Alison"] -> Original no modificado

a.sort(); // Ordena el array original
console.log(a); // ["Alison", "Anne", "Jen", "Peter", "Rob", "Thomas"]
```

Importante: Por defecto todos los valores de un array se ordenan alfabéticamente. Es decir, si un valor no es de tipo string, se convierte a dicho tipo internamente de cara a saber qué posición ocupará.

Si queremos ordenar otro tipo de datos, o de una manera diferente al orden alfabético de la 'a' a la 'z', debemos pasarle al método sort (o sorted) una función de ordenación. Esta función comparará 2 valores del array y devolverá un valor numérico indicando cual es menor (negativo si el primero es menor, 0 si son iguales y positivo si el primero es mayor).

# index.js

```
let a = [20, 6, 100, 51, 28, 9];
a.sort(); // Ordena el array alfabéticamente
console.log(a); // Imprime [100, 20, 28, 51, 6, 9]
a.sort((n1, n2) => n1 - n2); // Trabajando con números, podemos devolver la resta
console.log(a); // Imprime [6, 9, 20, 28, 51, 100]

/**** Ejemplo con objetos, que veremos más adelante ****/
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  toString() {
    return this.name + " (" + this.age + ")";
  }
}

let people = [
  new Person("Thomas", 24),
  new Person("Mary", 15),
  new Person("John", 51),
  new Person("Phillip", 9)
];

people.sort((p1, p2) => p1.age - p2.age); // Ordenamos por edad número
console.log(people.toString()); // "Phillip (9),Mary (15),Thomas (24),John (51)"

people.sort((p1, p2) => p1.name.localeCompare(p2.name)); // Ordenamos por nombre (string)
console.log(people.toString()); // "John (51),Mary (15),Phillip (9),Thomas (24)"
```

Cambiar el valor de una posición generando un nuevo array

Otro método nuevo de la versión ES2023 es **with**. Este nos devuelve un nuevo array con un valor cualquiera modificado. Al contrario que asignar un nuevo valor a una posición del array usando la notación de corchetes, este método preserva el array original intacto.

# index.js

```
let  a = [1, 2, 3, 4];
let a2 = a.with(2, 99); // En lugar de hacer a[2] = 99
console.log(a); // [1, 2, 3, 4] -> Original
console.log(a2); // [1, 2, 99, 4] -> Nuevo array con el cambio
```

Recorrer array de manera funcional

Podemos iterar por los elementos de un array usando el método **forEach**. De forma opcional, podemos llevar un seguimiento del índice al que está accediendo en cada momento, e incluso recibir el array como tercer parámetro.

# index.js

```
let a = [3, 21, 15, 61, 9, 54];
```

```
a.forEach((num, indice, array) => { // índice y array son parámetros opcionales
    console.log(indice + " -> " + num);
});
```

Comprobar una condición con todos los elementos

El método **every** devolverá un booleano indicando si todos los elementos del array cumplen cierta condición. Esta función recibirá un elemento, lo testeará, y devolverá cierto o falso dependiendo de si cumple la condición o no. A este tipo de funciones se les llama **predicados**.

Por otro lado, el método **some** es similar, pero devuelve true en cuanto uno de los elementos del array cumpla la condición.

# index.js

```
let a = [3, 21, 15, 61, 9, 54];
console.log(a.every(num =>  num < 100));  // Comprueba si cada número es menor a 100. Imprime true
console.log(a.every(num => num % 2 == 0)); // Comprueba si cada número es par. Imprime false

console.log(a.some(num => num % 2 == 0));  // Comprueba si algún elemento del array es par. Imprime true
```

Modificar todos los elementos del array (map)

Para modificar todos los elementos de un array, el método **map** recibe una función que devuelve un nuevo valor a partir de un elemento del array. El método map devolverá al final un nuevo array del mismo tamaño conteniendo todos los valores generados.

# index.js

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

Filtrar los elementos del array (filter)

Para filtrar los elementos de un array, y obtener como resultado un array que contenga sólo los elementos que cumplan cierta condición, usamos el método **filter**. Este método recibe una función (predicado), que devuelve un valor booleano, indicando los elementos que se incluirán en el nuevo array generado.

# index.js

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

Calcular valor a partir de array (reduce)

The **reduce** method uses a function that accumulates a value, processing each element (the second parameter) with the accumulated value (the first parameter). As the second parameter to reduce, you should pass an initial value. If you don't pass an initial value, the first element of an array will be used as such (if the array is empty, undefined will be returned).

To do the same thing as reduce but in reverse (starting at the end of the array), we will use **reduceRight** .

# index.js

```
let  a = [4, 21, 33, 12, 9, 54];
console.log(a.reduce((total, num) => total + num, 0));  // Suma todos los elementos del array. Imprime 133
console.log(a.reduce((max, num) => Math.max(max, num), 0));  // Número máximo del array. Imprime 54

console.log(a.reduceRight((total, num)  => total - num)); // Imprime -25 (Toma la última posición como valor inicial
al no proporcionarlo)
```

Search for elements in an array

Using **indexOf** , we can determine whether the value we pass is in the array or not. If it is found, it returns the first position where it is, and if not, it returns -1. If we use the **lastIndexOf** method , it returns the first occurrence found, starting from the end.

Optionally, we can pass as a second parameter the position of the array from which it will search.

# index.js

```
let a = [3, 21, 15, 61, 9, 15];
console.log(a.indexOf(15)); // Imprime 2
console.log(a.indexOf(15, 3)); // Imprime 5
console.log(a.indexOf(56)); // Imprime -1. No encontrado
console.log(a.lastIndexOf(15)); // Imprime 5
```

Search for elements with functional methods

find Finds and returns the first value it finds that meets the specified condition. With **findIndex** , we return the position of that value in the array.

Equivalent methods that start searching from the last position in the array have recently been standardized. These methods are **findLast** and **findLastIndex** .

# index.js

```
let numbers = [2, 4, 6, 9, 14, 16];
console.log(numbers.find(num => num >= 10)); // Imprime 14 (primer valor encontrado >= 10)
console.log(numbers.findIndex(num => num >= 10)); // Imprime 4 (numbers[4] -> 14)
console.log(numbers.findLast(num => num >= 10)); // Imprime 16 (último valor encontrado >= 10)
console.log(numbers.findLastIndex(num => num >= 10)); // Imprime 5 (numbers[5] -> 16)
```

Extract internal arrays

From an array containing other arrays. The **flat** method extracts the elements from the internal arrays and concatenates them into the resulting array. You can specify the level of internal arrays to extract as a parameter (the default is 1).

# index.js

```
let nums = [[1, 2, 3], [4, 5, 6]];
console.log(nums.flat()); // [1, 2, 3, 4, 5, 6]

let nums2 = [[[1, 2],[3,4]], [5,6], [[7, 8], 9]];
console.log(nums2.flat()); // [[1, 2], [3, 4], 5, 6, [7, 8], 9]
console.log(nums2.flat(2)); // [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The flatmap method is a combination of map and flat. If the function returns an array, the values are extracted and included as part of the resulting final array.

# index.js

```
let words = ["house", "tree", "dog"];
console.log(words.map(w => Array.from(w))); // [["h", "o", "u", "s", "e"], ["t", "r", "e", "e"], ["d", "o", "g"]]
console.log(words.flatMap(w => Array.from(w))); // ["h", "o", "u", "s", "e", "t", "r", "e", "e", "d", "o", "g"]
```

Rest/Spread

Rest (grouping of values)

Rest is the action of grouping a series of values passed to a function/method under a single parameter (of array type).

To use **rest** in a function's parameters, it is always declared as the last parameter (there can't be more than one) and preceded by three colons ('...'). This parameter will automatically be transformed into an array containing all the remaining values passed to the function. If, for example, the rest parameter is in the third position, it will contain all the parameters passed to it except the first and second (starting with the third).

# index.js

```
function getMedia(...notas) {
    console.log(notas); // Imprime [5, 7, 8.5, 6.75, 9] (está en un array)
    let total = notas.reduce((total,notas) => total + notas, 0);
    return total / notas.length;
}

console.log(getMedia(5, 7, 8.5, 6.75, 9)); // Imprime 7.25

function imprimirUsuario(nombre, ...lenguajes) {
    console.log(nombre + " sabe " + lenguajes.length + " lenguajes: " + lenguajes.join(" - "));
}

// Imprime "Pedro sabe 3 lenguajes: Java - C# - Python"
imprimirUsuario("Pedro", "Java", "C#", "Python");
// Imprime "María sabe 5 lenguajes: JavaScript - Angular - PHP - HTML - CSS"
imprimirUsuario("María", "JavaScript", "Angular", "PHP", "HTML", "CSS");
```

Spread (unbundling of securities)

Spread is the opposite of rest. If we have a variable that contains a collection (array, for example), and we put three dots ' **...** ' in front of it, it will extract all of its values. We could use the property, for example, to calculate the maximum numerical value in an array using the Math.max method, which takes an indeterminate number of separate parameters and returns the largest of them all.

# index.js

```
let nums = [12, 32, 6, 8, 23];
console.log(Math.max(nums)); // Imprime NaN (array no es válido), deben ser valores independientes
console.log(Math.max(...nums)); // Imprime 32 -> equivale a Math.max(12, 32, 6 ,8 ,23)
```

Another possibility of the **spread** operator is to clone arrays easily.

# index.js

```
let a = [1, 2, 3, 4];
let b = a; // 'b' referencia el mismo array que 'a' (las modificaciones afectan a ambas variables).
let c = [...a]; // Nuevo array (copia de a) -> contiene [1, 2, 3, 4].
```

It also allows us to create an array with the contents of other arrays, that is, to concatenate them. Individual values can be inserted at any position. In practice, it would be like placing all the values of the array individually in that position.

# index.js

```
let a = [1, 2, 3, 4];
let b = [5, 6, 7, 8];
let c = [...a,...b, 9, 10]; //  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Destructuring

Destructuring an array is the action of extracting individual elements from an array 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 a single assignment.

# index.js

```
let array = [150, 400, 780, 1500, 200];
let [v1, v2, v3] = array; // Asigna los tres primeros elementos del array
console.log(v3); // Imprime 780
```

What happens if we want to skip a value? The space inside the brackets is left empty (without a variable), and it won't be assigned.

# index.js

```
let array = [150, 400, 780, 1500, 200];
let [v1, , v3] = array; // Asigna el primer y tercer elemento
console.log(v3); // Imprime 780
```

We can assign the remainder of the array to the last variable we put in brackets using the **rest** operator .

# index.js

```
let array = [150, 400, 780, 1500, 200];
let [v1, v2, ...resto] = array; // resto será un array
console.log(resto); // Imprime [780, 1500 ,200]
```

If we want to assign more values than the array can hold and we don't want to assign undefined in case that position has no value, we can assign **default values** in that case.

# index.js

```
let array = ["Peter", "John"];
let [v1, v2 = "Mary", v3 = "Anne"] = array;
console.log(v2); // Imprime "John"
console.log(v3); // Imprime "Anne" -> valor por defecto
```

We can also destructure **nested arrays** . Just mimic the nested structure of the array in the structure on the left (variables).

# index.js

```
let sueldos = [["Pedro", "Maria"], [24000, 35400]];
let [[nombre1, nombre2], [sueldo1, sueldo2]] = sueldos;
console.log(nombre1 + " gana " + sueldo1 + "€"); // Imprime "Pedro gana 24000€"
```

You can also destructure an array passed as **a parameter to a function** . This improves the readability of the code within the function, since we're accessing a named variable, not an array position.

# index.js

```
function imprimirUsuario([id, nombre, email], otraInfo = "Nada") {
    console.log("ID: " + id);
    console.log("Nombre: " + nombre);
    console.log("Email: " + email );
    console.log("Otra info: " + otraInfo );
}

let infoUsu = [3, "Pedro", "peter@gmail.com"];
imprimirUsuario(infoUsu, "No es muy listo");
```

Map y Set

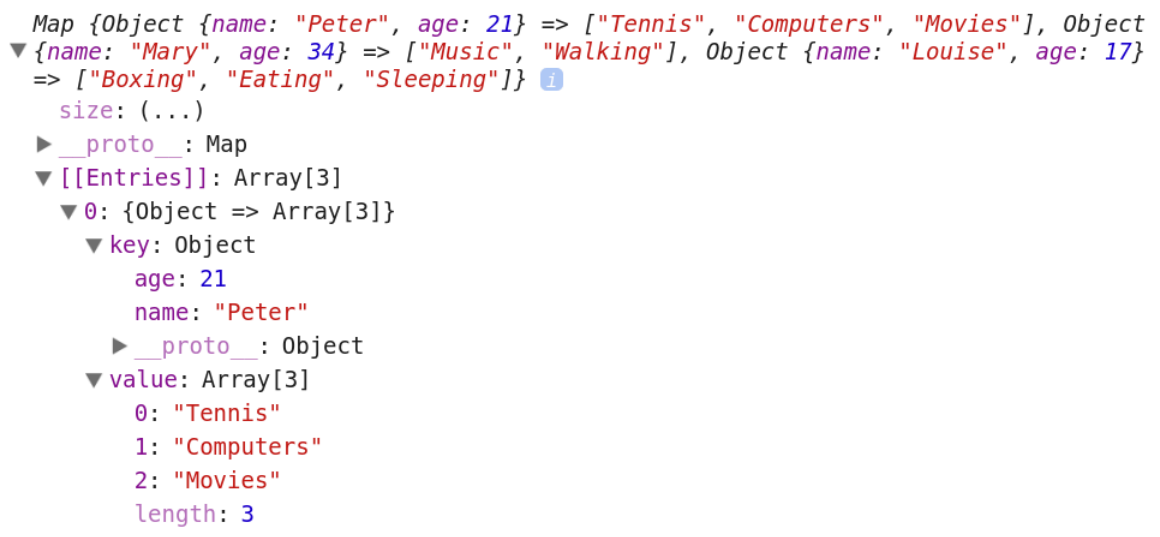
Dictionary (Map)

A dictionary (**Map**) is a collection that stores key-value pairs; the values are accessed using the corresponding key. In JavaScript, an object could be considered a type of Map, but with some limitations (only strings and integers as keys).

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

# index.js

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

let hobbies = new Map(); // Almacenará una persona con un array de 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
```

Just as when assigning an object to a variable, array, function parameter, etc., when we use an object as a key, we must know that we are storing a reference to that object. Therefore, the same reference must be used to access its value, modify it, or delete it.

# index.js

```
console.log(hobbies.has(person1)); // true (referencia al objeto original almacenado)
console.log(hobbies.has({name: "Peter", age: 21})); // false (mismas propiedades pero objeto diferente!)
```

The size property returns the length of the map, and we can iterate through it using the **for..of** loop or the **forEach** method . In the first case, for each iteration, an array with two positions is returned: 0 → key, and 1 → value.

# index.js

```
 // Continuamos con el código anterior
console.log(hobbies.size); // Imprime 3
hobbies.delete(person2); // Elimina person2 del Map
console.log(hobbies.size); // Imprime 2
console.log(hobbies.get(person3)[2]); // Imprime "Sleeping"

/** Recorremos Map y lo imprimimos:
 * 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) { // Mejor
    console.log(person.name + ": " + hobArray.join(", "));
}

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

Set

The Set collection is internally similar to a Map, but it doesn't store the value portion (only the key). It can be thought of as a collection that doesn't allow duplicate values (an array can have duplicate values). It's a fairly simple collection that only has methods like **add** , **delete ,** and **has** , which return a boolean and are used to store, delete, and check if a value exists.

Internally, a set doesn't store values in positions marked by an index like an Array. This means, as with a Map collection, that values don't have to be traversed in the order they were inserted.

# index.js

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

set.add("Mary"); // Mary ya existe
console.log(set.size); // Imprime 2

// Itera a través de los valores
set.forEach(value => {
    console.log(value);
})
```

We can create a Set from an array (which eliminates duplicate values).

# index.js

```
let names = ["Jennifer", "Alex", "Tony", "Johny", "Alex", "Tony", "Alex"];
let nameSet = new Set(names);
console.log(nameSet); // Set {"Jennifer", "Alex", "Tony", "Johny"}
names = [...nameSet]; // Reasignamos el array con los duplicados eliminados
```

Operations between sets

In version ES2025, new methods have been introduced to perform operations between different sets:

- **intersection** : Elements that are repeated in both sets
- **union** : All elements of both sets
- **difference** : Elements of the first set that are not in the second
- **symmetricDifference** : Elements that are not repeated in both sets
- **isSubsetOf** : Set 2 contains all elements present in set 1 (boolean)
- **isSupersetOf** : Set 1 contains all elements present in set 2 (boolean)
- **isDisjointFrom** : There are no elements that are repeated in both sets (boolean)

# index.js

```
const setOne = new Set([1, 2, 3]);
const setTwo = new Set([3, 4, 5]);

setOne.intersection(setTwo);
// Set(1) { 3 }
setOne.union(setTwo);
// Set(5) { 1, 2, 3, 4, 5 }
setOne.difference(setTwo);
// Set(2) { 1, 2 }
setOne.symmetricDifference(setTwo);
// Set(4) { 1, 2, 4, 5 }

setOne.isSubsetOf(setTwo);
// false
setOne.isSupersetOf(setTwo);
// false
```

```
setOne.isDisjointFrom(setTwo);
// false
```

Iterator

An iterator is an object that allows you to sequentially traverse a collection of values by calling the **next()** method . This method returns an object with the following properties: **value** (current value) and **done** (a boolean indicating whether it has finished traversing the sequence).

An iterator can be created from an array using the **values()** method .

# index.js

```
const a = [23, 45, 67, 89, 12];
const iterador = a.values();

console.log(iterador.next()); // { value: 23, done: false }
console.log(iterador.next()); // { value: 45, done: false }
console.log(iterador.next()); // { value: 67, done: false }
console.log(iterador.next()); // { value: 89, done: false }
console.log(iterador.next()); // { value: 12, done: false }
console.log(iterador.next()); // { value: undefined, done: true }

// Otra forma de crear un iterador

const iterador2 = a[Symbol.iterator]();
console.log(iterador2.next()); // { value: 23, done: false }
// ... etc
```

You can also use the **Iterator.from(collection)** method to create an iterator that iterates through any type of collection, including text strings.

# index.js

```
const a = [23, 45, 67, 89, 12];
const iterador = Iterator.from(a);

console.log(iterador.next()); // { value: 23, done: false }
// ... hasta que se acaben los elementos

const cadena = "Hola";
const iteradorCadena = Iterator.from(cadena);

console.log(iteradorCadena.next()); // { value: 'H', done: false }
// ... hasta que se acaben los caracteres
```

You can iterate through the elements of an iterator using a **for..of** loop .

# index.js

```
const a = [23, 45, 67, 89, 12];
const iterador = Iterator.from(a);

for(const n of iterador){
    console.log(n);
}
// Imprime los valores: 23 45 67 89 12
```

Operators such as spread (...) can also be applied to iterators.

# index.js

```
const a = [23, 45, 67, 89, 12];
const iterador = Iterator.from(a);

console.log([...iterador, 100, 200, 300]); // [23, 45, 67, 89, 12, 100, 200, 300]
```

Important : Iterators can only be traversed forward, so values already read cannot be accessed again by the iterator. For example, if we traverse all the values in the iterator and then transform it into an array, it will return an empty array.

# index.js

```
const a = [23, 45, 67, 89, 12];
const iterador = Iterator.from(a);
```

```
console.log(iterador.next()); // {value: 23, done: false}
console.log(iterador.next()); // {value: 45, done: false}

// El operador spread internamente recorre el iterador hasta el final (Desde la última posición)
console.log([...iterador]); // [67, 89, 12] -> El resto de valores
console.log([...iterador]); // [] -> Ya no quedan valores
```

Operations with iterators

In the ES2025 version of JavaScript, new methods have been introduced for iterators (Iterator Helpers) that allow intermediate operations with data such as filter, map, reduce, etc.

Unlike equivalent methods on JavaScript arrays, iterators behave more efficiently (similar to Java's Stream collection) since intermediate arrays are not generated with each operation, but rather each operation is applied individually to each element when its value is accessed.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Iterator/drop

Next we will see the new methods for iterators with examples of how they work.

toArray -> Returns the remaining elements of the iterator in an array.

# index.js

```
const a = [23, 45, 67, 89, 12];
const iterador = Iterator.from(a);

console.log(iterador.next()); // {value: 23, done: false}
console.log(iterador.next()); // {value: 45, done: false}

const restantes = iterador.toArray();
console.log(restantes); // [67, 89, 12]
```

forEach -> Iterates through the remaining elements applying the corresponding function to them.

# index.js

```
iterador.forEach(e => console.log(e));
```

find -> Returns the first element, among the remaining ones, that satisfies the predicate function it receives, or undefined.

# index.js

```
console.log(iterador.find(e => e % 2 === 0)); // 12
```

every and **some** -> Return a boolean value indicating whether all or some (respectively) of the remaining elements meet the predicate condition.

# index.js

```
console.log(iterador.every(e => e % 2 === 0)); // false
console.log(iterador.some(e => e % 2 === 0)); // true
```

drop -> Skips (deletes) the first n elements

# index.js

```
const a = [23, 45, 67, 89, 12];
const iterador = Iterator.from(a);

console.log(iterador.drop(3).toArray()); // [89, 12]
```

take -> Keeps the first n elements, discarding the rest

# index.js

```
const a = [23, 45, 67, 89, 12];
const iterador = Iterator.from(a);

console.log(iterador.take(3).toArray()); // [23, 45, 67]
```

filter -> returns another iterator that iterates through the elements that meet the predicate condition.

# index.js

```
console.log(iterador.filter((n) => n % 2 === 0).toArray()); // [12]
```

map -> Returns another iterator that transforms the values from a function.

# index.js

```
console.log(iterador.map((n) => n * 2).toArray()); // [46, 90, 134, 178, 24]
```

reduce -> Returns a final value calculated based on the elements of the iterator.

# index.js

```
console.log(iterador.reduce((total, n) => total + n)); // 236
```

flatMap -> Similar to map, but if we return an iterable collection in the function, it extracts its elements.

# index.js

```
const a = ["hola", "adiós", "casa", "coche"];
const iterador = Iterator.from(a);

console.log(iterador.flatMap((s) => [...s]).toArray());
// ['h', 'o', 'l', 'a', 'a', 'd', 'i', 'ó', 's', 'c', 'a', 's', 'a', 'c', 'o', 'c', 'h', 'e']
```

Obviously, several of these methods can be combined, keeping in mind that some of them return another iterator. Transformation, filtering, and other functions are not applied until the elements are traversed. This is unlike similar array methods, where each call generates a new intermediate array in memory.

# index.js

```
const a = ["hola", "adiós", "casa", "coche"];
const iterador = Iterator.from(a);

console.log(
  iterador
    .flatMap((s) => [...s])
    .filter((l) => /[aeiouáéíóú]/.test(l))
    .map((l) => l.toUpperCase())
    .take(5)
    .toArray()
);
// ['O', 'A', 'A', 'I', 'Ó']
```