JSON

JSON, or **JavaScript Object Notation** , is a special notation used to create generic objects in JavaScript. It's a popular format for exchanging data between applications or for storing information in non-SQL databases (MongoDB, for example). For more information, see json.org.

First, let's see how to create a generic object in JavaScript without using JSON, and we'll add some properties and methods to it (yes, in JavaScript, we can add properties and methods to an object at any time). To create a generic (and therefore empty) object, we use the **Object** class .

As we can see, we add (or access) properties to the object using either the dot (**object.property**) or the associative array notation (**object[property]**).

# index.js

```
let obj = new Object();
obj.nombre = "Peter"; // Añadimos la propiedad 'nombre' usando la notación del punto
obj["edad"] = 41; // Añadimos la propiedad 'edad' usando la notación de un array asociativo
obj.getInfo = function() { // Creamos un nuevo método → getInfo()
    return "Mi nombre es " + this.nombre + " y tengo " + this.edad
}

console.log(obj.getInfo()); // Imprime "Mi nombre es Peter y tengo 41"
console.log(obj.nombre); // Imprime "Peter". Accedemos al nombre usando la notación del punto
console.log(obj["nombre"]); // Imprime "Peter". Ahora accedemos con la notación del array asociativo
let prop = "nombre";
console.log(obj[prop]); // Imprime "Peter". Podemos acceder a la propiedad “nombre” a partir de una variable que
almacena el nombre de dicha propiedad (sólo para la notación de array)
```

Now, we'll do the same thing using JSON notation. It's equivalent to what we did before, but we'll assign properties using a colon ':' instead of the equal sign '='. The initial properties will be declared between braces, but we can add more later, as we did in the previous example. The equivalent of using new Object() would be to leave the braces **{ }** (empty object).

# index.js

```
let obj = {
    nombre: "Peter",
    edad: 41,
    getInfo () { // Método
        return "Mi nombre es " + this.nombre + " y tengo " + this.edad
    }
}
```

In JSON, as in JavaScript, brackets are used to create arrays. Within an array, we can store other objects (always in JSON format), primitive values, or other arrays.

# index.js

```
let persona = {
  nombre: "Peter",
  edad: 41,
  trabajos: [ // trabajos es un array de objetos JSON
      {
          descripcion: "Malabarista",
          duracion: "2003-2005"
      },
      {
          descripcion: "Conductor de autobús",
          duracion: "2005-2015"
      }
  ]
}

console.log(persona.trabajos[1].descripcion); // Imprime "Conductor de autobús”
```

Classes

Since version ES2015, a more modern syntax for creating classes in JavaScript has been standardized, similar to other object-oriented languages, avoiding the concept of constructor functions and using prototype explicitly (although internally it still works that way).

# index.js

```
class Product { }

console.log(typeof Product); // Imprime "function". Internamente sigue siendo una función como en versiones antiguas
```

Builder

To create a constructor and assign attributes to objects instantiated from the class, we have to implement a method called **constructor()** .

Within the constructor and other methods of the class, we must use the keyword **this** to access the properties of the current object, either to assign new values or to read them. Unlike in other strongly typed languages, we don't need to declare the object's attributes outside of the constructor (although it is possible to do so);

# index.js

```
class Product {
    constructor(nombre, precio) {
        this.nombre = nombre;
        this.precio = precio;
    }
}

let p = new Product("Producto", 50);
console.log(p); // Product {nombre: "Producto", precio: 50}
```

Methods

This is how we declare methods in a class. These methods access the object from which they are called using the **this** reference .

# index.js

```
class Product {
    ///...
    getDescuento(descuento) {
        let totalDesc = this.precio * descuento / 100;
        return this.precio - totalDesc;
    }
  }

let p = new Product("Producto", 50);
console.log(p.getDescuento(20)); // Imprime 40
```

Public/private sphere

Starting with ES2022, **private** attributes and methods can be declared in a class. To declare them as private, you must precede the attribute or method name with a hash mark (#). This way, they cannot be accessed from outside the class.

This means that if we want to access an object's private attributes, we must do so through methods (getters/setters) or public properties. We'll look at both options below. This allows us to control access, and for example, control the value assigned to an attribute, or simply disallow changes to its value (read-only).

# index.js

```
class User {
    #name;

    constructor(name) {
        this.#name = name;
    }
}

let u = User("Pepe");
console.log(u.#name); // ERROR: Uncaught SyntaxError: Private field '#name' must be declared in an enclosing class
```

Java-style getters/setters

It's based on declaring methods with the same name as the attributes and prefixed with get or set. We call the method in question to access or modify the attribute.

# index.js

```
class User {
    #name;

    constructor(name) {
        this.setName(name);
    }

    getName() { // Getter
        return this.#name;
    }

    setName(name) { // Setter
        this.#name = name;
    }
}

let user = new User("john");
user.setName("Alex");
console.log(user.getName()); // Alex
```

C#/Python style getters/setters

Also known as public properties. Internally, it works the same, but in this case, the prefix (get/set) is separated by a space. Externally, it's used as if we were accessing a public attribute, but when we read the value or overwrite it, the corresponding getter/setter is called underneath. You could say it's a more natural way to access an object's attributes.

# index.js

```
class User {
    #name;

    constructor(name) {
        this.name = name; // Llamada implícita al setter
    }

    get name() { // Getter
        return this.#name;
    }

    set name(name) { // Setter
        this.#name = name;
    }
  }

let user = new User("john");
user.name = "Alex"; // Llamada implícita al setter
console.log(user.name); // Llamada implícita al getter
```

Static properties

In modern JavaScript, you can declare both class properties and methods, that is, methods independent of any instantiated object. To do this, you prefix them with the keyword **static** . You must access these properties and methods using the class name as a prefix, followed by a period.

# index.js

```
class Empleado {
    static #sueldoMinimo = 15000;

    constructor(nombre, sueldo) {
        this.nombre = nombre;
        this.sueldo = sueldo;
    }

    static creaBecario(nombre) {
        return new Empleado(nombre, Empleado.#sueldoMinimo);
    }
}

let e = Empleado.creaBecario("Elena");
console.log(e); // Empleado {nombre: 'Elena', sueldo: '15000'}
```

Inheritance

A class can inherit from another class using the extends keyword . It will inherit all the properties and methods of the parent class. Of course, we can override them in the child class, but we can still call the parent class's methods using the **super** keyword . In fact, if we create a constructor in the child class, we must call the parent's constructor using super.

# index.js

```
class User {
    constructor(name) {
        this.name = name;
    }

    sayHello() {
        console.log(`Hola, soy ${this.name}`);
    }

    sayType() {
        console.log("Soy un usuario");
    }
}

class Admin extends User {
    constructor(name) {
        super(name); // Llamamos al constructor de User
    }

    sayType() { // Método sobrescrito
        super.sayType(); // Llamamos a User.sayType (método del padre)
        console.log("Pero también un admin");
    }
}

let admin = new Admin("Anthony");
admin.sayHello(); // Imprime "Hola, soy Anthony"
admin.sayType(); // Imprime "Soy un usuario" y "Pero también un admin"
```

Primitive value and string

When an object is converted to a string, the **toString** method (inherited from Object) is automatically called. By default, it will print the text "[object Object]", but we can override this method.

# index.js

```
class Persona {
    constructor(nombre, edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    toString() {
        return `${this.nombre} (${this.edad})`
    }
}

let p = new Persona("Ana", 34);
console.log("Persona: " + p); // # Persona: Ana (34)
```

When comparing objects using relational operators (>, <, >=, <=), we get the default primitive value (toString() by default). If we override the **valueOf()** method (inherited from Object), it returns another primitive value, which will be used for this type of comparison.

# index.js

```
class Persona {
    constructor(nombre, edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    // Otros métodos
    valueOf() {
        return this.edad; // Los objetos se compararán por edad
    }
}

let p = new Persona("Ana", 34);
let p2 = new Persona("Juan", 25);

console.log(p > p2); // true
```

Operations with objects

Object destructuring

Just like with arrays, it's also possible to destructure objects (extract property:value pairs). The operation is similar to destructuring an array, but we use curly braces ' **{ }** ' instead of brackets.

# index.js

```
let usuario = {
    id: 3,
    nombre: "Pedro",
    email: "peter@gmail.com"
}

let {id, nombre, email} = usuario;
console.log(nombre); // Imprime "Pedro"

// Se pueden asignar variables con nombres diferentes a los atributos
let {nombre: nombreUsuario, email: emailUsuario} = usuario;
console.log(nombreUsuario); // Imprime "Pedro"

// Esta función recibirá un objeto como primer parámetro y lo desestructurará en variables
function imprimirUsuario({id, nombre, email}, otraInfo = "Nada") {
    // Cuerpo de la función
}

otraInfo(usuario, "Es muy listo");
```

Object Spread

We can use the spread operator **'...'** with objects to clone objects, or create new objects by combining the properties of two or more objects. When there are duplicate properties in both objects, the value of the last object prevails.

# index.js

```
function configGame(options) {
    let defaults = {
        name: "Player 1",
        level: 1,
        difficulty: "normal",
        gender: "female"
    };

    let config = {...defaults, ...options}; // Combinamos el objeto defaults con options
    console.log(config);
}

let options = {
    name: "Super Master",
    gender: "male"
};
configGame(options); // {name: "Super Master", level: 1, difficulty: "normal", gender: "male"}
```

Optional concatenation (?.)

The ? operator works similarly to the property chaining operator (.), but with one key difference: if a **null** or **undefined** reference is encountered at any point in the sequence , the expression short-circuits and returns that value (null or undefined) immediately, instead of raising an error that would stop the program from executing.

Optional concatenation syntax can be applied in three ways:

- Accessing object properties: **object?.property**
- Accessing array elements: **array?.[index]**
- Calls to functions or methods: **function?.(parameters)**

# Properties

```
const coche = {
  marca: 'Ford',
  modelo: 'Mustang',
  // motor: {
  //   cilindros: 8
  // }
};

const cilindros = coche?.motor?.cilindros;
console.log(cilindros); // undefined (no hay error)
```

# Arrays

```
const coche = {
  marca: 'Ford',
  modelo: 'Mustang',
  // motor: {
  //   cilindros: 8
  // }
};

const cilindros = coche?.motor?.cilindros;
console.log(cilindros); // undefined (no hay error)
```

# Functions

```
const coche = {
  marca: 'Ford',
  modelo: 'Mustang',
  // motor: {
  //   cilindros: 8
  // }
};

const cilindros = coche?.motor?.cilindros;
console.log(cilindros); // undefined (no hay error)
```

Combine with the Null Coalescence Operator (??)

Optional chaining is perfectly complemented by the null coalescing operator (??). The latter allows you to provide a default value if an expression evaluates to null or undefined.

# index.js

```
const configuracion = {
  // tema: {
  //   color: 'oscuro'
  // }
};

const temaActual = configuracion.tema?.color ?? 'claro'; // configuracion.tema devuelve "undefined"

console.log(temaActual); // "claro"
```

Modules

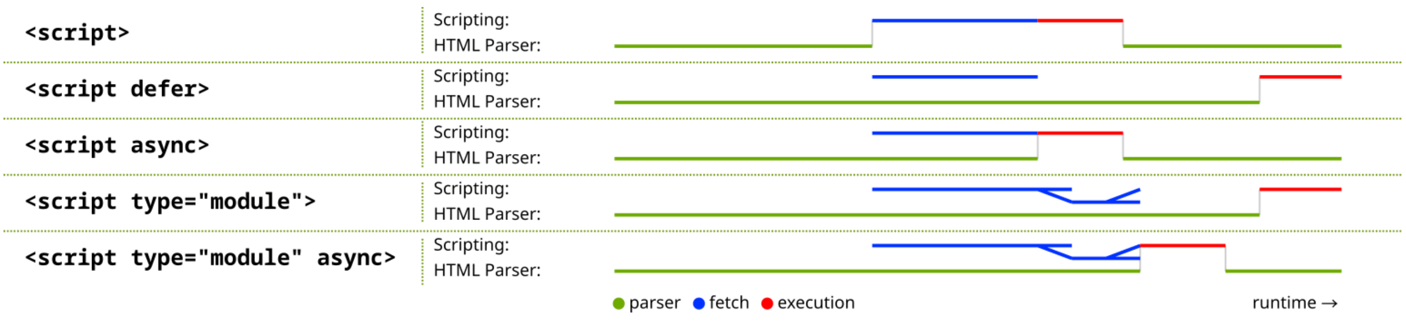
Why use modules?

Manually adding all the JavaScript files we're going to use to our HTML (classes, functions, constants, libraries, etc.) can be problematic in a large and complex project. It could lead to problems with overriding global variables or functions, including much more code than necessary in the application, cluttering the HTML, etc.

Dividing our application into modules (each file is a module) has been supported by JavaScript since ES2015. This allows us to decide which variables, functions, classes, etc. we will export in each module or file to use from other modules (files). This way, we only need to include one (or a few) main files in the HTML, and from there we can import the rest of the things we need. Variables will always be local to each module, reducing the risk of code overlap and the side effects that global variables and functions can have on the entire application.

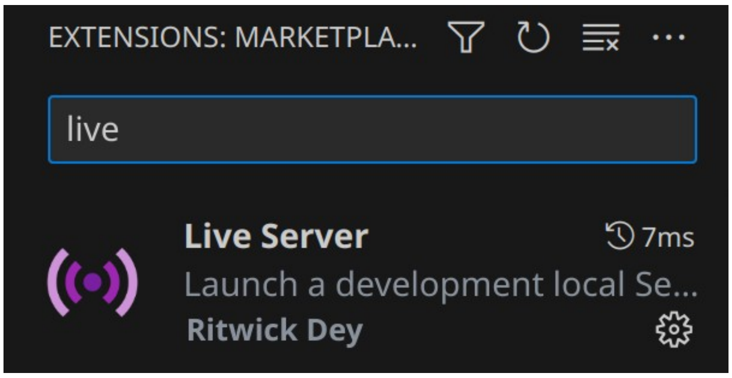
To use modules, we simply declare the imported script in HTML as **type="module"** . The only downside to this is that we must run our application under a web server for it to work.

By including this attribute, the **defer** feature is implicit, so it is no longer necessary to add this attribute.



Install Live Server

Visual Studio Code has a very useful extension, **Live Server** , that allows you to run a lightweight web server with a click of the mouse (and stop it in the same way).



In the lower right corner, you'll see a section that says " **Go live** ." A single click will launch the service and open a browser at http://localhost:5500 where you can test the application.

Now we can add the scripts with the attribute **type="module"** to our HTML document:

# index.html

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <!-- Resto cabeceras -->
    <script src="index.js" type="module"></script>
</head>
<body>
    <!-- Contenido -->
</body>
</html>
```

Operation of the modules

Each file is a module, and in principle everything (variables, constants, classes, functions) is private to that module (not accessible from other modules). For other modules to be able to use, for example, functions or classes from the current module, we must export them.

export default

The simplest form would be a module that only exports one thing. For example, a file with only one class defined and we want to use it from other modules. Using the **export default** syntax , we export the element in question, which we import from other modules (we can import it with any name).

# person.class.js

```
export default class {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
}
```

# index.js

```
export default class {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
}
```

multiple export

If we want to export more than one element (or if we want to export it with a predefined name), we won't use the default keyword; instead, whatever we export must have a name. This way, we can export multiple elements. Now, when importing, we must indicate which elements we want to import from the other module between curly braces { }.

# person.class.js

```
export class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
}

export const ROLES = ["admin", "guest", "user"];
export const GUEST_NAME = "Anonymous";
```

# index.js

```
export class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
}

export const ROLES = ["admin", "guest", "user"];
export const GUEST_NAME = "Anonymous";
```

Another way to export elements (equivalent to the previous one) is to declare at the end of the file what it will export. As with the previous method, we don't have to import everything a module exports from another, just what we need.

# person.class.js

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

const ROLES = ["admin", "guest", "user"];
const GUEST_NAME = "Anonymous";
```

```
export {Person, ROLES, GUEST_NAME};
```

# index.js

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

const ROLES = ["admin", "guest", "user"];
const GUEST_NAME = "Anonymous";

export {Person, ROLES, GUEST_NAME};
```

Other ways to import

We can also alias anything we import with the **as** keyword :

# index.js

```
import {Person as User, GUEST_NAME as GUEST} from './person.class';

let p = new User(GUEST, 30);
console.log(p.name); // Imprime "Anonymous"
```

Another option is to import everything a module exports into an object. This means that to access any imported element (class, constant, function), we must access it as a property of the created object. To do this, we create an alias (the object name) and import it using this syntax:

# index.js

```
import * as personModule from './person.class';

let p = new personModule.Person(personModule.GUEST_NAME, 30);
console.log(p.name); // Imprime "Anonymous"
```

This pre-import syntax would be the closest thing to the classic NodeJS **require** syntax , also known as CommonJS (CJS). The modern syntax with **export** and **import** , also supported by NodeJS today, is known as ES Modules (ESM).

Finally, if we want to run a JavaScript file that isn't designed as a module (an old library, for example) and doesn't export anything, we simply import the file. In this case, the entire contents of the file will be executed and processed, creating, among other things, the variables, functions, etc. defined in it and making them accessible to the current module.

# index.js

```
// Ejecuta el archivo functions.js. Equivale a importarlo en el HTML.
import './functions.js';
```

Dates

The Date class

In JavaScript, the **Date** class is used to represent a date, time, and time zone. It also encapsulates methods for operating on that date.

# index.js

```
let date = new Date(); // Crea objeto Date almacena la fecha actual
console.log(typeof date); // Imprime object
console.log(date instanceof Date); // Imprime true
console.log(date); // Imprime por ejemplo: Fri Jun 24 2016 12:27:32 GMT+0200 (CEST)
```

The Date constructor can receive the number of milliseconds since 1/1/1970 00:00:00 GMT (also called **Epoch time** or **UNIX time**). If you pass more than one number (only the first and second are required), the order should be: 1st → year, 2nd → month (0..11), 3rd → day, 4th → hours, 5th → minutes, 6th → seconds. Alternatively, you can pass a string containing the date in a standard international format.

# index.js

```
let date  = new Date(1363754739620); // Nueva fecha 20/03/2013 05:45:39 (milisegundos desde Epoch)
let date2 = new Date(2015, 5, 17, 12, 30, 50); // 17/06/2015 12:30:50 (Mes empieza en 0 -> Ene, ... 11 -> Dic)
let date3 = new Date("2015-03-25"); // Formato de fecha largo sin la hora YYYY-MM-DD (00:00:00)
let date4 = new Date("2015-03-25T12:00:00"); // Formato fecha largo con la fecha
let date5 = new Date("03/25/2015"); // Formato corto MM/DD/YYYY
let date6 = new Date("25 Mar 2015");  // Formato corto con el mes en texto (March también sería válido).
let date7 = new Date("Wed Mar 25 2015 09:56:24 GMT+0100 (CET)"); // Formato completo con timezone
```

If, instead of a Date object, we want to directly obtain the milliseconds that have passed since 1/1/1970 (Epoch), what we have to do is use the **Date.parse** (string) and **Date.UTC** (year, month, day, hour, minute, seconds) methods. We can also use **Date.now()** , which gives us the current date and time in milliseconds.

# index.js

```
let nowMs = Date.now(); // Momento actual en ms
let dateMs = Date.parse("25 Mar 2015"); // 25 Marzo 2015 en ms
let dateMs2 = Date.UTC(2015, 2, 25); // 25 Marzo 2015 en ms
```

Methods of the Date class

The Date class has setters and getters for the properties: **fullYear** , **month** (between 0: January and 11: December), **date** (day of the month: 1-31), **hours** , **minutes** , **seconds** , and **milliseconds . The getDay()** method also returns the current day of the week (between 0: Sunday and 6: Saturday). If we pass a negative value, for example, **date.setMonth(-1)** , the last month (December) of the previous year is set.

# index.js

```
// Crea un objeto fecha de hace 2 horas
let twoHoursAgo = new Date(Date.now() - (1000*60*60*2)); // (Ahora - 2 horas) en ms
// Ahora hacemos lo mismo,pero usando el método setHours
let now = new Date();
now.setHours(now.getHours() - 2);
```

When we want to print the date, we have methods that return it to us in different formats:

# index.js

```
let now = new Date(); // Imaginemos que ahora es 01/06/2016

console.log(now.toString());
console.log(now.toISOString()); // Imprime 2016-06-26T18:00:31.246Z
console.log(now.toUTCString()); // Imprime Sun, 26 Jun 2016 18:02:48 GMT
console.log(now.toDateString()); // Imprime Sun Jun 26 2016
console.log(now.toLocaleDateString()); // Imprime 26/6/2016
console.log(now.toTimeString()); // Imprime 20:00:31 GMT+0200 (CEST)
console.log(now.toLocaleTimeString()); // Imprime 20:00:31
```

Internationalization API (Intl)

The new JavaScript **Internationalization API** offers, among other things, the ability to format dates, numbers, and compare strings using the specifics of your chosen language.

Intl.Collator

Generates a language-aware string comparator. For example, if we set it to Spanish (es), it can compare words with accents, the letter ñ, etc.

# index.js

```
let a = ['adiós', 'árbol', 'oído', 'óptimo', 'ñapa', 'niño'];
a.sort();
console.log(a.toString()); // adiós,niño,oído,árbol,ñapa,óptimo (ordena en ingés - por defecto)
a.sort(new Intl.Collator('es').compare);
console.log(a.toString()); // adiós,árbol,niño,ñapa,oído,óptimo (ordena en español)
```

Intl.ListFormat

From a collection, generates a string with the elements of the list but taking into account the conjunction used for the last element according to the chosen language (in Spanish it would be 'and' or 'or').

# index.js

```
let a = ['perro', 'gato', 'pez', 'loro'];

const formatter = new Intl.ListFormat('es', { style: 'long', type: 'conjunction' });
console.log(formatter.format(a)); // perro, gato, pez y loro

const formatter2 = new Intl.ListFormat('es', { style: 'long', type: 'disjunction' });
console.log(formatter2.format(a)); // perro, gato, pez o loro

const formatter3 = new Intl.ListFormat('es', { style: 'short', type: 'unit' });
console.log(formatter3.format(a)); // perro, gato, pez, loro
```

Intl.NumberFormat

Formats a number taking into account the decimal or thousands separator used in the language. You can also add a currency symbol of your choice (before or after, depending on the language).

# index.js

```
let number = 15300.9555;

console.log(new Intl.NumberFormat('en-UK').format(number)); // 15,300.956
console.log(new Intl.NumberFormat('es-ES').format(number)); // 15.300,956

console.log(new Intl.NumberFormat('es-ES', { style: 'currency', currency: 'EUR' }).format(number));
// 15.300,96 €
console.log(new Intl.NumberFormat('en-US', { style: 'currency', currency: 'USD' }).format(number));
// $15,300.96
```

Intl.DateTimeFormat

This class allows you to format dates according to the language. Several date style parameters can be configured, as shown in the following examples:

# index.js

```
let date = new Date('2022-04-25');

console.log(new Intl.DateTimeFormat('es-ES').format(date)); // 25/4/2022 (por defecto)

console.log(new Intl.DateTimeFormat('es-ES', {
    dateStyle: "short"
}).format(date)); // 25/4/22

console.log(new Intl.DateTimeFormat('es-ES', {
    dateStyle: "full"
}).format(date)); // lunes, 25 de abril de 2022

console.log(new Intl.DateTimeFormat('es-ES', {
    day: "2-digit", month: "2-digit", year: "numeric"
}).format(date)); // 25/04/2022

console.log(new Intl.DateTimeFormat('es-ES', {
    day: "numeric", month: "long", year: "numeric" ,
    hour: 'numeric', minute: 'numeric', hourCycle: 'h12', dayPeriod: 'long'
}).format(date)); // 25 de abril de 2022 2:00 de la madrugada
```

Regular Expressions

Regular expressions allow us to search for patterns in a string, thereby finding the part that matches or fulfills the expression. In JavaScript, we can create a regular expression by instantiating a RegExp object **or** by directly enclosing the expression between two slashes (/).

An expression can also have one or more modifiers, such as **'g'** → Global match (finds all matches, not just the first), **'i'** → Case-insensitive (does not distinguish between upper and lower case), or **'m'** → Match on multiple lines (only makes sense for strings with newlines). In JavaScript, you can create a regular expression object with these modifiers in two ways. For example:

# index.js

```
let reg  = new RegExp("^[a-z]","gi");
let reg2 = /^[a-z]/gi;
console.log(reg2 instanceof RegExp); // Imprime true
```

Methods for working with regular expressions

Check if a string matches a pattern

The test() method takes a string and attempts to find a match for the regular expression. If the global modifier 'g' is specified, each time the method is called, it will be executed from the position where the last match was found, so we can determine how many matches are found. We must remember that using the 'i' modifier is case-insensitive.

# index.js

```
let str = "I am amazed in America";
let reg = /am/g;
console.log(reg.test(str)); // Imprime true
console.log(reg.test(str)); // Imprime true
console.log(reg.test(str)); // Imprime false, hay solo dos coincidencias

let reg2 = /am/gi; // "Am" será lo que busque ahora
console.log(reg.test(str)); // Imprime true
console.log(reg.test(str)); // Imprime true
console.log(reg.test(str)); // Imprime true. Ahora tenemos 3 coincidencias con este nuevo patrón
```

Find matches in a string

If we want more details about how many times a pattern is repeated within a string, we could use the **exec()** method . This method returns an object with details when it finds a match. These details include the index at which the match starts and also the entire string.

# index.js

```
let str = "I am amazed in America";
let reg = /am/gi;
console.log(reg.exec(str)); // Imprime ["am", index: 2, input: "I am amazed in America"]
console.log(reg.exec(str)); // Imprime ["am", index: 5, input: "I am amazed in America"]
console.log(reg.exec(str)); // Imprime ["Am", index: 15, input: "I am amazed in America"]
console.log(reg.exec(str)); // Imprime null. No hay más coincidencias
```

One way to go through these coincidences would be this:

# index.js

```
let str = "I am amazed in America";
let reg = /am/gi;
let match;
while(match = reg.exec(str)) {
    console.log("Patrón encontrado!: " + match[0] + ", en la posición: " + match.index);
}
/* Esto imprimirá:
  * Patrón encontrado!: am, en la posición: 2
  * Patrón encontrado!: am, en la posición: 5
  * Patrón encontrado!: Am, en la posición: 15 */
```

Similarly, there are methods in the **String** class that we can use that accept regular expressions as parameters (in other words, the other way around). These methods are **match** (which works similarly to exec) and **replace** .

The match method returns an array with all the matches found in the string if we put the **global modifier → g** , if we do not put the global modifier, it behaves the same as exec().

# index.js

```
let str = "I am amazed in America";
console.log(str.match(/am/gi)); // Imprime ["am", "am", "Am"]
```

Pattern replacement

The replace method returns a new string with the regular expression matches replaced by the string passed as the second parameter (if the global modifier is not specified, only the first match is modified). We can send an anonymous function that processes each match found and returns the string with the corresponding replacements.

# index.js

```
let str = "I am amazed in America";
console.log(str.replace(/am/gi, "xx")); // Imprime "I xx xxazed in xxerica"
console.log(str.replace(/am/gi, function(match) {
    return "-" + match.toUpperCase() + "-";
})); // Imprime "I -AM- -AM-azed in -AM-erica"
```