

DESARROLLO WEB
ENTORNO CLIENTE

2º DAW
CFGS

3. JS: PROGRAMACIÓN CON OBJETOS



Javier Polit Ramón

IES La Senia

j.politramon@edu.gva.es

Curso
2025-2026

CONTENIDO

Contenido.....	3
1. INTRODUCCIÓN	5
1.1. Características de la POO	5
2. OBJETOS	7
2.1. Creación de objetos.....	7
2.2. Propiedades de un objeto	10
2.3. Objetos dentro de objetos.....	12
2.4. Palabra reservada this.....	13
2.5. Métodos de los objetos	14
2.6. Formas de añadir métodos a los objetos.....	17
2.7. Instanceof y constructor.name	19
2.8. Recorrer un objeto.....	20
2.9. Get y Set	20
3. CLASES	22
3.1. Crear una clase y sus objetos	22
3.2. Añadir métodos a una clase	23
3.3. Métodos y atributos estáticos	24
3.4. Heredar una clase	25
3.5. Propiedades y métodos privados	29
4. ARRAY OBJECT	32
4.1. Declaración, modificación y acceso a los datos.....	32
4.2. Destructuring de arrays	37
4.3. Propiedad length	38
4.4. Métodos de array	38
4.5. Recorrer un array	46
5. DATE OBJECT	49
5.1. Declaración y sintaxis	49
5.2. Métodos get y set.....	50
5.3. Métodos para formatear las fechas	51

6.	<i>MATH OBJECT</i>	52
6.1.	Declaración variables de tipo math	52
6.2.	Métodos del objeto Math	53
7.	<i>SET OBJECT</i>	55
7.1.	Declaración	55
7.2.	Métodos de set object	56
7.3.	Propiedad size	58
7.4.	Convertir set en array	59
7.5.	Recorrer un set	59
8.	<i>MAP OBJECT</i>	60
8.1.	Declarar e instanciar map	60
8.2.	Métodos de map	60
8.3.	Propiedad size	63
8.4.	Convertir map en array	63
8.5.	Recorrer un map	63
9.	<i>ANEXO: PROTOTYPE</i>	64
9.1.	Crear un prototype	66
9.2.	Heredar un prototype: call	68
10.	<i>ANEXO: REGEXP OBJECT</i>	69
10.1.	Declaración y sintaxis	69
10.2.	Métodos de RegExp	70
10.3.	Elementos de las RegExp	71

1. INTRODUCCIÓN

La programación orientada a objetos o POO es un **modelo de programación**.

Cada objeto se programa de forma independiente, consiguiendo una **mayor modularidad**.

Los **objetos** son una **estructura que recogen datos o propiedades y acciones o métodos**. Por ejemplo, si el objeto es un coche, tendría:

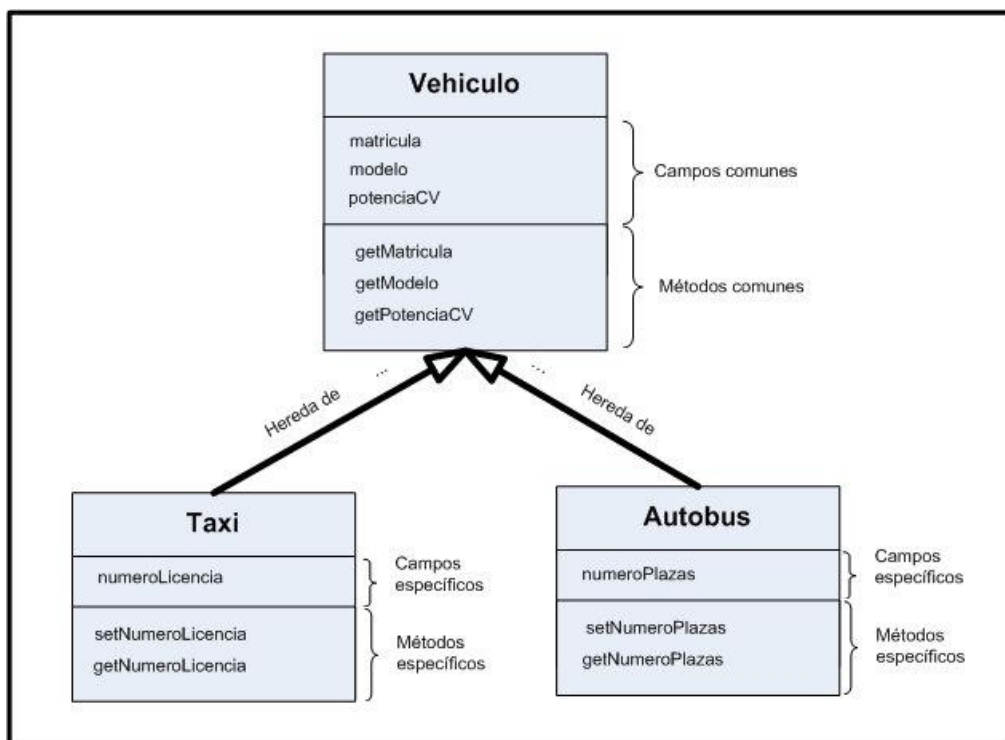
- ✓ **Propiedades:** marca, color, potencia...
- ✓ **Métodos:** arrancar, parar, acelerar, repostar...

Una vez que un **tipo de objeto** o **clase** se ha definido, se podrá utilizar las veces que queramos.

1.1. Características de la POO

Para considerar que un lenguaje es de POO, debe permitir que su programación tenga estas características:

- ✓ **Abstracción:** Debe permitir reutilizar un objeto sin conocer su funcionamiento interno. Bastará con que proporcione su **interfaz externa**. Esto son las **propiedades** y **métodos** que permiten manipular y utilizar el objeto.
- ✓ **Encapsulamiento:** Capacidad de que los objetos puedan ocultar **propiedades y métodos** de forma que podamos elegir los que nos interesen que queden visibles.
- ✓ **Polimorfismo:** Característica que permite que **distintos tipos de objetos pueden tener métodos con el mismo nombre, pero actuando de distinta forma**. Como ejemplo:
 - Objetos de **clase círculo**: tienen el **método suma** que unen su superficie a la de otro círculo.
 - Objetos de **clase númeroGigante**: tienen el **método suma** que realiza sumas aritméticas.
- ✓ **Herencia:** Los diferentes tipos de objetos se relacionan con **clases de objetos**. Esto quiere decir que la mayoría de los lenguajes define clases y luego crea objetos a partir de esas clases: cada objeto será de una determinada clase. La herencia permite que haya clases que hereden las **características de otras clases**. Por ejemplo:
 - **Clase Automóvil** puede tener un método que sea **acelerar** y otro que sea **frenar**.
 - **Clase Coche**, derivada de **Automóvil**, podrán también **acelerar y frenar**.



Pero... JavaScript cumple con todas estas características???????

JS era un lenguaje basado en prototipos y no en clases.

A partir de ES6, este concepto cambia....

Si bien JavaScript trabaja los objetos de forma diferente a otros lenguajes como Java o C++, indudablemente **los objetos son fundamentales** para este lenguaje y toda aplicación JavaScript utiliza objetos y permite modular el código gracias a las interacciones entre ellos.

A partir de ES6 o ECMAScript2015, se incorporan aspectos similares a las clases.

Los **prototipos** servirán para poder clonar nuestros objetos, es decir, clonar un objeto sin instanciarlo.

Las **clases** que se introducen en este estándar supone una gran mejora respecto a la herencia basada en prototipos. Aportan una sintaxis más clara y sencilla para crear objetos y trabajar la herencia.

```

class Animal {
  constructor(nombre) {
    this.nombre = nombre;
  }
  sonido() {
    console.log('El animal hace un sonido.')
  }
}

```

Ejemplo de Polimorfismo

```

class Perro extends Animal {
  sonido() {
    console.log('El perro ladra.');
```

2. OBJETOS

La creación de **objetos** nos va a permitir trabajar con una **estructura sólida**.

Si trabajamos con productos de nuestra tienda, no tiene sentido ir construyendo variables de “nombre”, “precio”, “descripción”, “stock”, etc.... Lo ideal es crear y trabajar con un objeto.

2.1. Creación de objetos

Podemos definir objetos de diferentes formas. Lo que haremos siempre es definir pares de *clave-valor* y, de esa forma ir añadiendo **atributos**. Estos pares NO tienen porqué ser del mismo tipo de dato. Incluso los valores pueden ser el resultado de operaciones u otros objetos.

Por otro lado, también podremos ir añadiendo **métodos** que después podremos utilizar para trabajar con los objetos.

Aunque NO es obligatorio, definiremos primero las propiedades y después los métodos

2.1.1. OBJECT LITERAL

Los objetos de tipo **Object** más sencillos que podemos crear y definir es a través de **literales** o con instancias directas.

Se trata de objetos en los que se pueden definir directamente sus propiedades y métodos sobre la marcha.

Los llamados **object literal**, instancias directas o literales son una opción muy utilizada.

```
// Definimos un objeto const
const miCoche = {
  marca: 'Ford',
  modelo: 'Fiesta'
};

// Intento reasignar la variable completa (esto dará un error)
// miCoche = { marca: 'Opel', modelo: 'Corsa' }; // Esto generará un TypeError

// Sin embargo, puedo cambiar las propiedades del objeto (esto sí es válido)
miCoche.modelo = 'Focus';
console.log(miCoche); // Salida: { marca: 'Ford', modelo: 'Focus' }
```

Nos fijamos en la sintaxis:

- El objeto se declara como una variable de tipo *const*
- Las propiedades y métodos van entre llaves.
- La clave y su valor se separan con dos puntos.
- Las propiedades se separan por comas.
- La última coma es opcional.

OJO: Hay que diferenciar entre la **reasignación de una variable** y la **modificación de las propiedades** de un objeto:

- La variable `miCoche` **NO puede ser reasignada** para apuntar a un objeto diferente.
- Pero las **propiedades del objeto** al que apunta sí **pueden ser modificadas**.

En versiones actuales de JS, si a los atributos de un objeto se le asignan valores de variables cuyo nombre es igual al atributo o propiedad, se pueden omitir. Veamos la diferencia:

```
const gasto = {
  nombre: nombre,
  cantidad: cantidad,
  id: Date.now()}

```

Los dos ejemplos crean el mismo objeto.

```
const gasto = {nombre, cantidad, id:Date.now()};

```

2.1.2. CONSTRUCTOR INTEGRADO: NEW

En JS encontramos **constructores propios del lenguaje**, son los **constructores integrados**.

En casi todos los casos, **NO será conveniente** utilizar este tipo de constructor, podremos crear una variable y asignarle valor.

El uso de los datos nativos es mucho más rápido y eficiente.

Estos objetos disponen de propiedades y métodos propios del lenguaje JavaScript.

Veamos los distintos **objetos de JavaScript** y cómo los vamos a definir y crear:

- ✓ Constructor String: nos va a permitir crear cadenas.

```
let nom = new String(); //No usar
let nombre = "Natalia Escrivá"; //Sí usar

```

- ✓ Constructor Number: Nos permite crear valores numéricos, enteros o decimales.

```
let num = new Number(); //No usar
let numero = 3.1416; //Sí usar

```

- ✓ Constructor Boolean: Nos permite crear valores lógicos true o false.

```
let bool = new Boolean(); //No usar
let booleano = true; //Sí usar

```


- ✓ Constructor Array: Nos permite crear arrays con distintos valores.

```
let masc = new Array(); //No usar
let mascotas = []; // Sí usar
```

- ✓ Constructor RegExp: Nos permite crear expresiones regulares que servirán para validar estructuras de cadenas o hacer validaciones.

```
let exp = new RegExp(); //No usar
let expReg = /<expresion regular>/; //Sí usar
```

- ✓ Constructor Function: Nos permite crear funciones.

```
let func = new Function(); //No usar
let suma = function(){}; //Sí usar
```

- ✓ Constructor Date: Nos permite crear datos de tipo temporal. **SI** que lo necesitaremos.

```
let fecha = new Date(); //SI usar
```

- ✓ Objeto Math: **No dispone de constructor integrado** ya que es un objeto global de JavaScript. Para invocar un método de este objeto únicamente debemos escribir el objeto y el método separados por un punto.

```
let num = Math.random()*11;
```

- ✓ Objeto Object: Nos va a permitir crear objetos de tipo *Object*. Es una alternativa a object literal. **NO** es el método más usado.

```
const trabajador1 = new Object();
//propiedades objeto trabajador1
trabajador1.nombre = "Natalia";
trabajador1.apellido = "Escrivá";
trabajador1.antiguedadAnos = 3;
console.log(trabajador1);
```

```
console.log(trabajador1);
```

```
{nombre: 'Natalia', apellido: 'Escrivá', antiguedadAnos: 3}
  antiguedadAnos: 3
  apellido: "Escrivá"
  nombre: "Natalia"
  ▶ [[Prototype]]: Object
```

2.1.3. OBJECT CONSTRUCTOR

Este método es el **precursor de las clases en JavaScript**.

Se define un objeto a través de una **función constructora** y así, podremos crear objetos del tipo construido.

Podríamos decir que es como crear una **plantilla de tipo de objeto** para luego reutilizarla y crear todos los objetos que queramos de ese tipo.

Usamos para el nombre del objeto la primera letra en mayúscula para diferenciarlo de los métodos que **NO** construyen objetos.


```
function Persona (nom, ape, anyos, prof){
  this.nombre = nom;
  this.apellido = ape;
  this.edad = anyos;
  this.profesion = prof;
}
//vamos a crear o instanciar un objeto tipo persona
const ana = new Persona ("Ana","López",45,"cirujana");
const pedro = new Persona ("Pedro","Ximénez",36,"vinicultor");

console.log(ana);
```

Vemos que hemos utilizado la palabra reservada **this** para hacer referencia al ámbito del objeto actual al que pertenece la función constructora.

Lo que permite la palabra reservada **this** es llegar a las propiedades concretas del objeto que se está creando, sabiendo que el valor de esas propiedades podrá ser distintas en cada objeto.

Cuando vayamos a crear objetos de tipo “Persona”, utilizaremos el constructor integrado con la palabra reservada **new**.

Este método NO es el que utilizaremos puesto que actualmente JS ya dispone de clases

2.2. Propiedades de un objeto

Los objetos en JavaScript, al igual que en los demás lenguajes de programación, tienen **propiedades**, también llamados **atributos**, que son las características del objeto.

Debemos saber que, aunque una variable de tipo *Object* esté declarada como **const**, permite añadir, editar y eliminar propiedades.



2.2.1. ACCEDER A LOS VALORES DE UN OBJETO

Dos opciones para acceder a los valores de las propiedades de un objeto.

1. A través de la “sintaxis del punto”.

Vemos que el editor directamente nos propone las propiedades.

```
const alumno1 = {
  nombre: "Natalia Escrivá",
  ciclo: "DAW",
  curso: 2
};
```

```
console.log(alumno1);
```

(property) ciclo: string
curso
nombre

```
console.log(alumno1.nombre);
```

Natalia Escrivá

2. A través de corchetes.

También podemos utilizar otro método, aunque no es el más común. En vez de utilizar el punto usaremos corchetes e indicaremos la clave a la que queremos acceder.

```
console.log(alumno1["ciclo"]);
```

DAW

```
const alumno1 = {
  nombre: "Natalia Escrivá",
  ciclo: "DAW",
  curso: 2
};
```

2.2.2. AÑADIR, EDITAR Y ELIMINAR PROPIEDADES DE UN OBJETO

1. Añadir propiedades.

La podemos añadir en la propia estructura del objeto o bien en otro momento del desarrollo de la aplicación.

Nos fijaremos que, si lo hacemos fuera, la sintaxis ya no es “clave : valor”, sino “clave = valor”.

```
alumno1.nia = "XXXXX";
console.log(alumno1.nia);
```

XXXXX

2. Editar propiedades.

```
alumno1.nia = 12345678;
console.log(alumno1["nia"]);
```

12345678

3. Borrar propiedades.

Ahora necesitaremos utilizar la palabra reservada **delete**.

```
delete alumno1.curso;
console.log(alumno1);
```

```
{nombre: 'Natalia Escrivá', ciclo: 'DAW', nia: 12345678}
  ciclo: "DAW"
  nia: 12345678
  nombre: "Natalia Escrivá"
  [[Prototype]]: Object
```

2.2.3. DESTRUCTURING DE OBJETOS

Se trata de **acceder a los valores del objeto y asignarlos a una variable DIRECTAMENTE**, todo en un mismo paso.

Utilizaremos la palabra reservada para crear la variable y, entre corchetes pondremos la propiedad de la que queremos obtener el valor. Lo debemos igualar al nombre del objeto del que queremos la información.

Lo vamos a comparar con el procedimiento habitual.

```
let {nombre} = alumno1;
console.log(nombre);
```

```
const n = alumno1.nombre;
console.log(n);
```

```
Natalia Escrivá
Natalia Escrivá
```

Si los ejecutamos vemos que el resultado es el mismo.

Podemos extraer varias propiedades del mismo objeto a la vez.

```
let {nombre, nia, ciclo} = alumno1;
console.log(nombre);
console.log(nia);
console.log(ciclo);
```

```
Natalia Escrivá
12345678
DAW
```

Si no existe la propiedad, crea la variable, pero con valor *undefined*. Vamos a intentar acceder a "curso" que es el que hemos borrado...

```
let {nombre, nia, ciclo, curso} = alumno1;
console.log(nombre);
console.log(nia);
console.log(ciclo);
console.log(curso);
```

```
Natalia Escrivá
12345678
DAW
undefined
```

2.3. Objetos dentro de objetos

Sabemos que los valores de las propiedades de los objetos no tienen por qué ser del mismo tipo. De hecho, podemos incluir una **propiedad que sea un objeto**.

En la ejecución se puede ver que el árbol se amplía con los 3 objetos.

Vamos a ver cómo se accede a las propiedades:

```
const producto = {
  articulo: "Ratón inalámbrico",
  precio: 35,
  disponible: true,
  informacion: {
    peso: "30 gramos",
    rgba: true,
    fabricacion: {
      pais: "China",
      anyo: 2022
    }
  }
};
console.log(producto);
```

```
{articulo: 'Ratón inalámbrico', precio: 35, disponible: true, informacion: {}}
  articulo: "Ratón inalámbrico"
  disponible: true
  informacion:
    fabricacion:
      anyo: 2022
      pais: "China"
    [[Prototype]]: Object
  peso: "30 gramos"
  rgba: true
  [[Prototype]]: Object
  precio: 35
```

```
console.log(producto.articulo);
console.log(producto.informacion);
console.log(producto.informacion.peso);
console.log(producto.informacion.fabricacion);
console.log(producto.informacion.fabricacion.pais);
```

```
Ratón inalámbrico
▶ {peso: '30 gramos', rgba: true, fabricacion: {...}}
30 gramos
▶ {pais: 'China', anyo: 2022}
China
```

2.3.1. DESTRUCTURING DE OBJETOS ANIDADOS

También en estos casos podremos ir accediendo a los valores de las propiedades y creando la variable a la que le asignamos ese valor en la misma instrucción.

Vamos a acceder a las mismas propiedades que en el ejemplo anterior.

```
const {articulo, informacion, informacion:{peso, fabricacion}, informacion: {fabricacion:{pais}}} = producto;
console.log(articulo);
console.log(informacion);
console.log(peso);
console.log(fabricacion);
console.log(pais);
```

```
Ratón inalámbrico
▶ {peso: '30 gramos', rgba: true, fabricacion: {...}}
30 gramos
▶ {pais: 'China', anyo: 2022}
China
```

2.4. Palabra reservada this

La palabra reservada **this** nos va a servir para hacer referencia o “apuntar” a las propiedades y métodos concretos del objeto que se está ejecutando en ese momento. Vemos un **ejemplo**:

1. Creamos unas variables globales:

```
const prod = "cama",
      tipo = "dormitorio",
      material = "metal",
      pvp = 500,
      stock = 1000;
```

2. Creamos un objeto:

```
const mueble = {
  prod : "mesa",
  tipo : "comedor",
  material: "madera",
  pvp: 350,
  stock: 105,
  //método propio del objeto
  mostrarInfo: function(){
    return `El mueble: ${prod},
    es de la categoría ${tipo}.
    Está hecho de ${mueble.material},
    tiene un precio de ${this.pvp}€ y
    hay ${this.stock} unidades en stock`;
  }
}
```

3. Llamamos a la función

```
console.log(mueble.mostrarInfo());
```

```
El mueble: cama,
  es de la categoría dormitorio.
  Está hecho de madera,
  tiene un precio de 350€ y
  hay 105 unidades en stock
```

Si nos damos cuenta, los dos primeros valores los coge de la variables globales. El 4º, 5º y 6º valor lo coge del objeto.

2.5. Métodos de los objetos

De la misma forma que los objetos tienen atributos o propiedades, es decir, características que lo definen, también tienen **métodos**, que son acciones asociadas a esos objetos.

Los métodos de los objetos son propiedades en forma de función.

METODO	FUNCIÓN
Object.freeze()	Congela el objeto para que NO se pueda modificar. No se podrá añadir, editar ni suprimir propiedades.
<pre>const modulo = { nombre: "DWES", horasSem: 7, ciclo: "DAW", curso: 2 }; console.log(modulo.nombre); modulo.nombre = "DWECE"; console.log(modulo.nombre);</pre> <pre>//intentamos editar propiedad modulo.nombre = "DIW"; console.log(modulo.nombre); //intentamos borrar propiedad delete modulo.curso; console.log(modulo); //intentamos añadir propiedad modulo.turno = "tarde"; console.log(modulo);</pre>	<p>Podemos ver que, aunque se ha declarado con const, se puede modificar su valor:</p> <pre>DWES DWECE</pre> <p>Aplicamos ahora el método:</p> <pre>Object.freeze(modulo);</pre> <p>Vamos a intentar hacer cambios.</p> <p>Vemos que NO se ha modificado nada.</p> <pre>DWECE ▶ {nombre: 'DWECE', horasSem: 7, ciclo: 'DAW', curso: 2} ▶ {nombre: 'DWECE', horasSem: 7, ciclo: 'DAW', curso: 2}</pre>
Object.isFrozen()	Indica si el objeto está congelado o no.
<pre>console.log(Object.isFrozen(modulo));</pre>	

Object.seal()	NO permite añadir ni borrar propiedades pero Sí permite editar las existentes.
<p>Creamos objeto y llamamos método. Después intentamos modificar el objeto (editamos propiedades y borramos una de ellas:</p> <pre>const perro = { nombre: "Fly", raza: "Pitbull", anyos: 2, } Object.seal(perro);</pre> <pre>perro.anyos = 5; perro.leGusta = "Jugar con pelota"; delete perro.raza; console.log(perro);</pre> <pre>{nombre: 'Fly', raza: 'Pitbull', anyos: 5} anyos: 5 nombre: "Fly" raza: "Pitbull" ▶ [[Prototype]]: Object</pre>	
Object.isSealed()	Indica si el objeto está sellado o no.
<pre>console.log(Object.isSealed(perro)); //true console.log(Object.isSealed(modulo)); //true</pre> <p>Vemos que en los dos casos el resultado es True. "modulo" está congelado que es más restrictivo que sellado.</p>	
Object.assign()	Permite reescribir objetos. Se pueden asignar valores de atributos y también funciones generales.
<p>Ejemplo 1 → con varios objetos, los une.</p> <pre>const objetosUnidos = Object.assign(modulo, perro);</pre> <pre>{nombre: 'Fly', horasSem: 7, ciclo: 'DAW', turno: 'tarde', raza: 'Pitbull', ...} anyos: 5 ciclo: "DAW" horasSem: 7 nombre: "Fly" raza: "Pitbull" turno: "tarde" ▶ [[Prototype]]: Object</pre> <p>Ejemplo 2 → asignar valores a un objeto que ya existe (en este caso, reinicia el objeto)</p> <pre>Object.assign(citaObj, { paciente: '', propietario: '', email: '', fecha: '', sintomas: '' })</pre>	

Spread o Rest Operator "..."	NO son métodos de Object exclusivamente, pero los podemos utilizar para unir varios objetos o para hacer referencia a una copia del objeto o al resto de elementos
<p>Ejemplo 1 → Unir dos objetos: usaremos el Spread Operator entre llaves para obtener un objeto.</p> <pre>const usuario = { nombre: 'Alex', edad: 28 }; const detallesContacto = { email: 'alex@email.com', telefono: '123-456-7890' }; const usuarioCompleto = { ...usuario, ...detallesContacto, pais: 'España' };</pre> <pre>console.log(usuarioCompleto); // Salida: // { // nombre: 'Alex', // edad: 28, // email: 'alex@email.com', // telefono: '123-456-7890', // pais: 'España' // }</pre> <p>Ejemplo 2 → Pasar una copia de un objeto como argumento de un método. Propagación</p> <p>El operador de propagación permite trabajar con una copia del objeto original dentro de la función, asegurando que el objeto original permanezca inmutable.</p> <pre>const productoOriginal = { nombre: 'Teclado', precio: 75, stock: 10 }; function venderProducto(productoVenta, cantidad) { // Aquí usamos la copia que se pasó como argumento console.log('Stock antes de la venta:', productoVenta.stock); productoVenta.stock -= cantidad; console.log('Stock después de la venta:', productoVenta.stock); } venderProducto({ ...productoOriginal }, 2); // Verificamos el estado del objeto original console.log('El objeto original se mantiene intacto:'); console.log(productoOriginal); // Salida: { nombre: 'Teclado', precio: 75, stock: 10 }</pre> <p>En este ejemplo, la función venderProducto trabaja sobre su propia copia del objeto, por lo que el stock dentro de la función se reduce, pero el stock del productoOriginal sigue siendo 10. Esto demuestra la utilidad del Spread Operator para mantener la integridad de tus datos.</p>	
structuredClone()	Crea una copia de un objeto
<p>Método que permite crear una copia de un objeto. Puede ser muy útil para la comunicación entre las funciones.</p>	
Object.keys()	Crea un array con las propiedades enumerables del objeto
<pre>function JuegoMesa(nombre, numJugadores, edadMin, pvp){ this.nombre = nombre ; this.numJugadores = numJugadores; this.edadMin = edadMin; this.pvp = pvp; } const juego1 = new JuegoMesa("Batalla de Genios", 2, 6, 25); console.log(Object.keys(juego1));</pre> <pre>▼ (4) ['nombre', 'numJugadores', 'edadMin', 'pvp'] 0: "nombre" 1: "numJugadores" 2: "edadMin" 3: "pvp" length: 4 ▶ [[Prototype]]: Array(0)</pre>	

Object.values()	Crea un array con los <u>valores</u> de las propiedades del objeto
<pre>console.log(Object.values(juego1));</pre>	<pre>▼ (4) ['Batalla de Genios', 2, 6, 25] 0: "Batalla de Genios" 1: 2 2: 6 3: 25 length: 4 ► [[Prototype]]: Array(0)</pre>
Object.entries()	Crea un array con las <u>propiedades y sus valores</u> de un objeto
<pre>console.log(Object.entries(juego1));</pre>	<pre>▼ (4) [Array(2), Array(2), Array(2), Array(2)] 0: (2) ['nombre', 'Batalla de Genios'] 1: (2) ['numJugadores', 2] 2: (2) ['edadMin', 6] 3: (2) ['pvp', 25] length: 4 ► [[Prototype]]: Array(0)</pre>

2.6. Formas de añadir métodos a los objetos

A los objetos además de propiedades también podemos añadirles métodos.

Si queremos agregar más de un método, los separaremos por comas, tal y como hacemos con las propiedades.

Vamos a ver distintas formas de añadir métodos a los objetos.

1. En la declaración del propio objeto (*Object Literal*):

```
const mueble = {
  prod : "mesa",
  tipo : "comedor",
  material: "madera",
  pvp: 350,
  stock: 202,
  //método propio del objeto
  mostrarInfo: function(){
    return `El mueble: ${this.prod}, es de la categoría ${this.tipo}. Está hecho de ${this.material},
    tiene un precio de ${this.pvp}€ y hay ${this.stock} unidades en stock`;
  }
}
```

```
console.log(mueble.mostrarInfo());
```

```
El mueble: mesa, es de la categoría comedor. Está hecho de madera,
tiene un precio de 350€ y hay 202 unidades en stock
```

2. En la declaración de la función "constructora" (*Object Constructor*):

```
function Ciudadano (nomCompleto, NIF, direc, CP, edad){
  //propiedades
  this.nombre = nomCompleto;
  this.NIF = NIF;
  this.domicilio = direc;
  this.CP = CP;
  this.edad = edad;
  //métodos
  this.carnetJove = function(){
    console.log((this.edad >= 12 && this.edad <=35) ? `Al ciudadano: ${this.nombre}
    se le puede expedir el Carnet Jove` : `Al ciudadano: ${this.nombre}
    NO se le puede expedir el Carnet Jove`);
  }
}

const ciudadano1 = new Ciudadano ("Marta Barranco Soler","11111111A","c/Pelayo, 3",46020, 45);
const ciudadano2 = new Ciudadano ("Daniel García Pérez","22222222B","c/Asturias, 32",46015, 15);
ciudadano1.carnetJove();
ciudadano2.carnetJove();
```

```
Al ciudadano: Marta Barranco Soler
    NO se le puede expedir el Carnet Jove
Al ciudadano: Daniel García Pérez
    se le puede expedir el Carnet Jove
```

3. Después de haber declarado el objeto (en cualquiera de los dos casos):

Ejemplo 1:

```
const alumno = {};
alumno.nombre = "Natalia";
alumno.modulos = ["DWECE", "DWES", "DIW", "DAW"];

alumno.info = function(){
  return `${alumno.nombre} --> ${alumno.modulos}`
}
console.log(alumno.info());
```

```
Natalia --> DWECE,DWES,DIW,DAW
```

Ejemplo 2:

```
mueble.hacerPedido = function(){
  if (this.stock < 10){
    console.log(`PEDIDO PENDIENTE DE ${this.prod} de ${this.tipo} de ${this.material},
    solo quedan ${this.stock} en stock`);
  }else{
    console.log(`De momento NO debemos pedir ${this.prod} de ${this.tipo} de ${this.material}`);
  }
}
mueble.hacerPedido();
```

```
De momento NO debemos pedir mesa de comedor de madera
```

2.7. Instanceof y constructor.name

Para averiguar el tipo de dato de un objeto podemos utilizar dos elementos del lenguaje JavaScript:

- ✓ Un método similar a **typeof**, pero pensado para ser usado con objetos, que se llama **instanceof**. Devuelve true o false.
- ✓ Una propiedad del objeto constructor que se llama **name**. Esta propiedad lo que hace es devolver el nombre de la función constructora y, por lo tanto, el tipo de objeto.

Debemos tener en cuenta que el **operador instanceof** **NO** considera directamente objetos en sí mismos a los tipos de **datos primitivos** (*String*, *Number* y *Boolean*), aunque JavaScript incorpora propiedades y métodos para este tipo de datos.

Si utilizamos un constructor SI lo considerará objeto, pero ya hemos visto que no va a ser necesario para poder hacer uso de los métodos y propiedades.

Cuando **instanceof** devuelve **TRUE**, nos indica que, el objeto concreto es de un tipo y, además pertenece a un tipo básico llamado Object, por lo que **heredará todo lo que posee la clase Object**.

Por otro lado, la propiedad **constructor.name** SI nos da el tipo de objeto concreto ya que mira el constructor de cada dato primitivo.

```
const edad = 15;
console.log(typeof edad); //Number
console.log(edad instanceof Number); //false
console.log(edad instanceof Object); //false
console.log(edad.constructor.name); //Number
```

```
const anyo = new Number();
console.log(typeof anyo); //Object
console.log(anyo instanceof Number); //true
console.log(anyo instanceof Object); //true
console.log(anyo.constructor.name); //Number
```

Los **arrays** sí se consideran directamente objetos, ya sean creados a través de un literal o con constructor.

```
const cosas2 = ["casa", "silla", "mesa"];
console.log(typeof cosas2); //Object
console.log(cosas2 instanceof Array); //true
console.log(cosas2 instanceof Object); //true
console.log(cosas2.constructor.name); //Array
```

```
const cosas = new Array();
console.log(typeof cosas); //Object
console.log(cosas instanceof Array); //true
console.log(cosas instanceof Object); //true
console.log(cosas.constructor.name); //Array
```

Por último, deberemos tener en cuenta que, si hemos definido y creado un objeto con un **literal**, **NO podremos crear otro objeto derivado de ese**, ya que **JavaScript NO lo considera un constructor**, deberemos cada vez crear los objetos escribiendo todo el código.

Por el contrario, si hemos utilizado un constructor, **NO tendremos ningún problema**.

```
const susana = new Persona();
console.log(typeof susana); //Object
console.log(susana instanceof Persona); //true
console.log(susana instanceof Object); //true
console.log(susana.constructor.name); //Persona
```

```
const pablo = new alumno;
console.log(pablo instanceof alumno); //error
```

```
✖ ▶ Uncaught TypeError: alumno is not a constructor
at 06-app.js:55:15
```

2.8. Recorrer un objeto

Podemos utilizar **dos métodos** para recorrer las propiedades de un objeto y trabajar con sus valores.

1. **For...in** → Para recorrer el objeto, utilizamos una variable que tomará el valor de una **propiedad** en cada iteración.

```
function recorreProp(objeto){
  for (let propiedad in objeto){
    console.log(`${propiedad}: ${objeto[propiedad]}`);
  }
}
recorreProp(trabajador);
```

```
nombre: Natalia
apellidos: Escrivá Núñez
departamento: informática
antigAnyos: 6
tutoria: true
```

2. **For...of** → Iterador propio de los objetos. Utiliza el método **Object.entries()**, este método devuelve un **array** con las **claves/valor** del objeto para recorrerlo. Al utilizarlo, permite acceder tanto a las propiedades como a los valores. Ejemplo:

```
let text = "";
for (let [propiedad,valor] of Object.entries(alumno)){
  text += `${propiedad} --> ${valor}` + "<br>";
}
document.getElementById("demo").innerHTML = text;
```

```
nombre --> Felipe
ciclo --> DAW
nia --> 12345678
```

Los nombre de las variables para recorrer los pares key/value pueden ser cualesquiera que deseemos. En este ejemplo se han usado: propiedad y valor por más claridad.

2.9. Get y Set

Consisten en **métodos** que favorecen las buenas prácticas a la hora de acceder y modificar las propiedades y métodos de los objetos.

2.9.1. GET

Sirve para crear una propiedad computada cuyo resultado saldrá de la ejecución de una función. Con get recibimos un valor.

Esta función **no tendrá parámetros**.

```
const vecino = {
  nombre: "Felipe",
  apellidos: "El Hermoso",
  edad: 30,
  mayorEdad: true,
  email: "felipeelhermoso.rey@gmail.com",
  domicilio: "Palacio Real, 1, bajo",
  esPropietario: true,
  pagaIBI: function() {
    return ((this.esPropietario) ?
      `El vecino: ${this.nombre} ${this.apellidos} debe pagar la cuota anual
      de IBI <br>del inmueble sito en: ${this.domicilio}`
      : `Al vecino: ${this.nombre} ${this.apellidos} <br>NO LE CORRESPONDE el pago
      de la cuota anual de IBI <br>de la vivienda ubicada en: ${this.domicilio}`)
  },
  get nombreCompleto() {
    return `${this.nombre} ${this.apellidos}`;
  },
}
```

Vemos en el ejemplo un **objeto** con varias propiedades o atributos, además, tiene un método propio (pagaIBI) y, a través de **get**, se ha creado otra **propiedad** adicional cuyo valor se calculará cada vez que sea llamada. Nos fijamos que para llamar a la función incluimos los paréntesis, en cambio, para llamar a la nueva propiedad creada con **get** no hemos necesitado hacerlo.

```
let text = "";
text += vecino.nombreCompleto + "<br>" ;
text += vecino.pagaIBI() + "<br>" + "<br>";
document.getElementById("demo").innerHTML = text
```

Felipe El Hermoso

El vecino: Felipe El Hermoso debe pagar la cuota anual de IBI
del inmueble sito en: Palacio Real, 1, bajo

2.9.2. SET

A través de este método lo que podremos hacer es **modificar los valores de las propiedades sin modificar el objeto directamente**. Se hará ejecutando una función.

El **setter** se define igual que el getter, pero **recibe argumento**. La invocación se hace a través de una **asignación**.

```
set mayorEdad(anyos){
  (anyos >= 18) ? this.mayorEdad = true : this.mayorEdad = false;
  this.edad = anyos;
}
```

```
console.log(vecino);
vecino.mayoriaEdad = 16;
console.log(vecino);
```

```
▶ {nombre: 'Pedro', apellidos: 'González López', edad: 35, mayorEdad: true, email: 'pedro22@gmail.com', ...}
▼ {nombre: 'Pedro', apellidos: 'González López', edad: 16, mayorEdad: false, email: 'pedro22@gmail.com', ...} ⓘ
  apellidos: "González López"
  domicilio: "San Cristóbal, 23, bajo"
  edad: 16
  email: "pedro22@gmail.com"
  mayorEdad: false
  nomCompleto: "Pedro González López"
  nombre: "Pedro"
  ▶ pagaIBI: f ()
  propietario: true
  ▶ set mayorEdad: f mayorEdad(anyos)
  ▶ get nomCompleto: f nomCompleto()
  ▶ [[Prototype]]: Object
```

3. CLASES

Una clase es una estructura que nos ayuda a simplificar la organización de nuestro código.

Nos permite agrupar variables y funciones en un elemento padre.

Internamente, JS traduce las clases al sistema basado en prototipos que usa en realidad.

Una clase en JS **NO ES** un **OBJETO**. Una **CLASE** es una **PLANTILLA** para **CREAR OBJETOS**.

3.1. Crear una clase y sus objetos

Vamos a ver las dos formas que ofrece JavaScript para declarar clases. Al igual que hacíamos para las funciones constructoras de objetos.

3.1.1. CLASS DECLARATION

Es la más popular. El nombre de la clase tendrá la inicial en mayúscula.

Utilizaremos la palabra reservada **class** para crear nuestra clase y añadiremos dentro el constructor.

Como constructor de la clase, utilizaremos el método **constructor**.

```
class Person{
  constructor(nom,ape,anyos){
    this.nombre = nom;
    this.apellido = ape;
    this.edad = anyos;
  }
}
```

```
const persona1 = new Person ("Rodolfo", "Moriente", 49);
console.log(persona1);
```

```
▼ Person {nombre: 'Rodolfo', apellido: 'Moriente', edad: 49} ⓘ
  apellido: "Moriente"
  edad: 49
  nombre: "Rodolfo"
  ▼ [[Prototype]]: Object
    ► constructor: class Person
    ► [[Prototype]]: Object
```

3.1.2. CLASS EXPRESSION

Se utiliza menos pero también la debemos conocer por si encontramos este tipo de código.

La diferencia la encontramos en que **en este caso se asigna la clase a una constante**.

```
const Person2 = class{
  constructor(nom,ape,anyos){
    this.nombre = nom;
    this.apellido = ape;
    this.edad = anyos;
  }
}
```

```
const persona2 = new Person2("Pepe", "Ximénez", 35);
console.log(persona2);
```

```
▼ Person2 {nombre: 'Pepe', apellido: 'Ximénez', edad: 35} ⓘ
  apellido: "Ximénez"
  edad: 35
  nombre: "Pepe"
  ▼ [[Prototype]]: Object
    ► constructor: class
    ► [[Prototype]]: Object
```

3.2. Añadir métodos a una clase

Añadiremos funciones específicas para los objetos creados de esa clase como si fueran atributos pero entre llaves.

```
fullName() {
  return (this.name + " " + this.surname); // Podría ser return (`${this.name} ${this.surname}`);
}
```

```
isAdult() {
  return (this.age < 18) ? `El estudiante ${this.fullName()} no es mayor de edad`
  : `El estudiante ${this.fullName()} es mayor de edad`;
  /* Alternativa con if-else:
  if (this.age < 18) {
    return `El estudiante ${this.fullName()} no es mayor de edad`;
  }
  else {
    return `El estudiante ${this.fullName()} es mayor de edad`;
  } */
}
```

```
const student1 = new Student ("John", "Doe", 18, "12345678", "DAW", 2);
const student2 = new Student ("Peter", "Parker", 17, "87654321", "DAM", 1);
```

```
let text = "";
text += student1.fullName() + "<br>";
text += "  NIA: " + student1.nia + "<br>";
text += "  Curso: " + student1.ciclo + "<br>";
text += "  Ciclo: " + student1.curso + "<br>";
text += student1.isAdult() + "<br><br>";
text += student2.fullName() + "<br>";
text += "  NIA: " + student2.nia + "<br>";
text += "  Ciclo: " + student2.ciclo + "<br>";
text += "  Curso: " + student2.curso + "<br>";
text += student2.isAdult() + "<br>";

document.getElementById("demo").innerHTML = text;
```

```
John Doe
NIA: 12345678
Curso: DAW
Ciclo: 2
El estudiante John Doe es mayor de edad

Peter Parker
NIA: 87654321
Ciclo: DAM
Curso: 1
El estudiante Peter Parker no es mavor de edad
```

Los métodos se añaden de la misma forma para las dos formas de crear clases: class declaration y class expression.

3.3. Métodos y atributos estáticos

Podemos crear **métodos y atributos asociados a las clases** y NO a los objetos creados de esa clase.

3.3.1. MÉTODOS ESTÁTICOS

Serán funciones dentro de una clase que se invocan directamente sin necesidad de instanciar la clase, es decir, NO necesitamos crear ningún objeto de esa clase para poder ejecutar ese método.

Usaremos la palabra reservada **static** para crearlo y para llamarlo usaremos la **clase**.

Ejemplo 1 → Sin parámetros: añadimos un método estático a la clase Person. Si la llamada la hacemos desde un objeto, DA ERROR!!

```
static bienvenida() {  
  return `Bienvenido/a!!!!`  
}
```

```
console.log(Person.bienvenida());  
console.log(student1.bienvenida());
```

```
Bienvenido/a!!!!  
❗ Uncaught TypeError: student1.bienvenida is not a function
```

Ejemplo 2 → Con parámetros

1. Creamos la clase y dos métodos estáticos

```
class Rectangulo {  
  constructor(x,y){  
    this.x = x;  
    this.y = y;  
  }  
  static area (a,b){  
    return a*b;  
  }  
  static perimetro (a,b){  
    return (a*2) + (b*2);  
  }  
}
```

2. Invocamos los métodos de la clase

```
console.log (Rectangulo.area(2,3)); //6  
console.log (Rectangulo.perimetro(2,3)); //10
```

3. Creamos un objeto Rectángulo

```
const rectang1 = new Rectangulo(2,3);
```

4. Invocamos los métodos con los valores del objeto pasados como parámetros

```
console.log (Rectangulo.area(rectang1.x,rectang1.y)); //6  
console.log (Rectangulo.perimetro(rectang1.x,rectang1.y)); //10
```

5. Invocamos los métodos directamente desde el objeto, observamos que da un error.

```
console.log (rectang1.area(2,3)); //ERROR!!  
console.log (rectang1.perimetro(2,3)); //ERROR!!
```

El error aparece porque NO es un método propio del objeto, si no de la clase.

```
❗ Uncaught TypeError: rectang1.area is not a function  
at 01-app.js:80:23
```

3.3.2. PROPIEDADES ESTÁTICAS

Al igual que en los métodos, podemos añadir propiedades estáticas a las clases.

Estas propiedades también **pertenecen a la clase y NO a los objetos**.

Utilizaremos la misma palabra reservada para crearla.

La crearemos y la inicializaremos.

En este caso, haremos referencia a ella en el constructor, pero NO usaremos la palabra reservada `this`, sino la clase.

En el mismo constructor la iremos autoincrementando para llevar el control del número de objetos de esa clase.

Las propiedades estáticas, al igual que los métodos también se heredan.

```
class Rectangulo {
    static contadorRectangulos = 0;

    constructor(x,y){
        this.x = x;
        this.y = y;
        Rectangulo.contadorRectangulos++;
        console.log(`Se incrementa contador: ${Rectangulo.contadorRectangulos}`);
    }
}
```

3.4. Heredar una clase

Podemos crear clases que hereden propiedades y métodos de otra clase (también los estáticos).

Por defecto, TODAS las clases heredan las propiedades y los métodos de la clase **Object**.

Para poder hacer subclases o herencia de clases, utilizaremos las palabras reservadas **extends** y **super**.

- **extends** para indicar que nuestra clase es “hija” de la indicada.
- **super** para hacer referencia a las propiedades y métodos de la clase “padre”.

Creamos una clase `Student` que va a heredar las propiedades y métodos de la clase `Person` (la clase tiene como métodos `fullName()`, `isAdult()`).

Directamente lo heredamos todo en la declaración de la clase.

```
class Student extends Person { // Student hereda de Person
  constructor(name, surname, age, nia, ciclo, curso) {
    super(name, surname, age); // Llama al constructor de la clase padre (Person)
    this.nia = nia;
    this.ciclo = ciclo;
    this.curso = curso;
  }
}
```

Vamos a crear un objeto de la nueva clase (`Student`) y vamos a llamar a los métodos de la clase “padre” (`Person`).

```
const student1 = new Student ("John", "Doe", 18, "12345678", "DAW", 2);
```

```
console.log(student1);
console.log(student1.fullName());
console.log(student1.isAdult());
```

```
► Object { name: "John", surname: "Doe", age: 18, nia: "12345678", ciclo: "DAW", curso: 2 }
John Doe
El estudiante John Doe es mayor de edad
```

Si vemos el detalle de nuestro objeto `student1` vemos que su **prototype** es `Person` y su constructor es `class Student`.

```
Prototipo de student1 (Student.prototype): ▼ Object { ... }
  ▼ constructor: class Student { constructor(name, surname, age, nia, ciclo, curso) }
    length: 6
    name: "Student"
    ► prototype: Object { ... }
    ► <prototype>: class Person { constructor(name, surname, age) } {≡}
```

3.4.1. AÑADIR MÉTODOS PROPIOS DE LA CLASE “HIJA”

Podremos crear funciones o métodos ligados únicamente a la clase “hija”. De esta forma estamos especificando la clase según las necesidades de nuestros objetos sin necesidad de repetir código.

Podemos crear métodos que invoquen a métodos de la clase “padre”. Para llamar a la función heredada, podremos utilizar dos palabras reservadas: **this** o **super**. Se usará una, otra o las dos en una misma función.

Ejemplo 1 → Método que invoca método “padre” con this

```
studentInfo() {
    return `El estudiante ${this.fullName()}, con NIA: ${this.nia},
    está en ${this.curso}º curso del ciclo ${this.ciclo}.`;
}
```

```
console.log(student1.studentInfo());
```

```
El estudiante John Doe, con NIA: 12345678,
está en 2º curso del ciclo DAW.
```

Ejemplo 2 → Método que invoca método “padre” con super

```
mayGoOut() {
    return (this.age < 18) ? `El estudiante ${this.fullName()} no puede salir del centro`
    : `El estudiante ${this.fullName()} puede salir del centro`;
}
```

```
console.log(student1.mayGoOut()); // Salida: El estudiante John Doe puede salir del centro
console.log(student2.mayGoOut()); // Salida: El estudiante Peter Parker no puede salir del centro
```

Es posible crear métodos en la clase “hija” que se llamen igual que el método en la clase “padre”. Lo que estaremos haciendo es **modificar el método del “padre” para la clase “hija”** y así adaptarlo a las necesidades de la nueva clase → **Polimorfismo** (múltiples formas en tiempo de ejecución: el método que se ejecuta depende de si es una referencia de tipo padre o de tipo hijo)

Ejemplo 3 → Método con el mismo nombre que el método “padre” añadido a la clase “hija”

```
static bienvenida() {
    return `Bienvenido/a al IES La Sènia!!!!`;
}
```

```
console.log(Student.bienvenida()); // Bienvenido/a al IES La Sènia!!!!
console.log(Person.bienvenida()); // Bienvenido/a!!!!
```

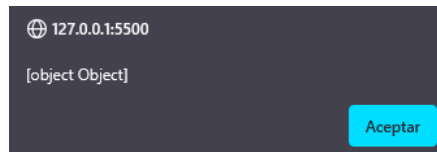
Observa la diferencia al invocar el método de la clase “padre” (Person) y el correspondiente de la clase “hija” (Student)

3.4.2. MÉTODO toString()

El método **toString()** pertenece a la clase genérica **Object**, y nos permite mostrar la información de los objetos en el navegador.

Si queremos mostrar por pantalla la información de un objeto, usaremos la siguiente instrucción:

```
alert(student1); // Muestra [object Object] porque alert() convierte el objeto a string
```



```
console.log(student1); // Muestra el objeto completo en la consola del navegador
```

```
▼ Object { name: "John", surname: "Doe", age: 18, nia: "12345678", ciclo: "DAW", curso: 2 }
  age: 18
  ciclo: "DAW"
  curso: 2
  name: "John"
  nia: "12345678"
  surname: "Doe"
  ► <prototype>: Object { _ }
```

Lo que está utilizando es el método **toString()** de **Object**. En pantalla la información no es muy útil.

Lo que podemos hacer es modificar este método para poder mostrar la información que queremos.

En la clase **Person**, modificamos **toString()** para adaptarlo localmente, así:

```
toString() {
  return this.fullName(); // Sobrescribe el método toString() de Object
  // para mostrar el nombre completo;
}
```

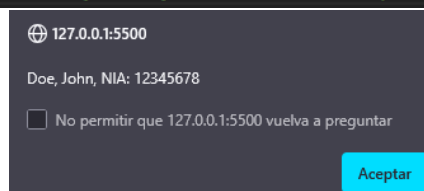


```
alert(student1); // Muestra [object Object] porque alert() convierte el objeto a string
// al sobrescribir el método toString() de Object devuelve el nombre completo
```

Si lo hacemos en la clase **Student**, “hija” de la clase **Person**, quedará así:

```
toString() {
  return `${this.surname}, ${this.name}, NIA: ${this.nia}`; // Sobrescribe el método toString()
  // de Person dentro de Student
}
```

```
alert(student1); // Ahora muestra "Doe, John, NIA: 12345678" por el método toString() sobrescrito
```



```
console.log(student1.toString()); // Ahora muestra "Doe, John, NIA: 12345678"
// porque hemos sobrescrito el método toString() de Person dentro de Student
```

```
Doe, John, NIA: 12345678
```

3.5. Propiedades y métodos privados

En JS **TODAS** las propiedades son públicas, es decir, se puede acceder a ellas desde la clase y desde cualquier objeto de esa clase.

Si queremos hacer alguna propiedad privada, es decir, que únicamente se pueda acceder a ella desde la declaración de la clase, podemos utilizar distintos métodos.

3.5.1. GETTERS Y SETTERS

A través de los **métodos get y set** lo que podemos es **extraer o modificar las propiedades** de la clase.

Debemos “marcar” las propiedades a las que vamos a aplicar estos métodos.

Cada vez que hacemos referencia a esa propiedad, bien con *this* o asignándole un valor, **NO** estaremos accediendo directamente a la propiedad, sino que estamos llamando a esos métodos.

Lo que intentamos simular es el trabajo que se realiza con una propiedad privada, sabiendo que JavaScript NO hace esta distinción.

Creamos una clase “Course” con una propiedad privada (`_difficultyLevel`). Nos fijamos en el guión bajo de la propiedad para diferenciarla de los métodos.

Añadimos un método **get** para extraer o hacer referencia a la propiedad y un método **set** para poder modificar la propiedad.

Se nombran con la palabra reservada **get** o **set** seguida del nombre de la propiedad que obtienen o modifican:

```
class Course {
  add a breakpoint (name, difficultyLevel) {
    this.name = name;
    this._difficultyLevel = difficultyLevel; //la variable se define con _ para indicar que es "privada"
    this.lessons = [];
  }
  get difficultyLevel() { // Getter para obtener el nivel de dificultad
    console.log("GETTER de difficultyLevel");
    return this._difficultyLevel;
  }
  set difficultyLevel(value) {
    console.log("SETTER de difficultyLevel"); // Valida que el nivel de dificultad esté entre 1 y 3
    (value >= 1 && value <= 3) ? this._difficultyLevel = value : // si cumple, asigna el valor
    console.log("El nivel de dificultad debe estar entre 1 y 3"); // sino, muestra un mensaje de error
  }
}
```

Creamos un nuevo objeto de la clase Curso.

Mostramos la propiedad “dificultad”. Vemos que pasa por el getter.

```
const cursoJS = new Course("JavaScript", 2);
console.log(cursoJS.difficultyLevel); // Llamamos al getter
```

```
GETTER de difficultyLevel
2
```

Modificamos la propiedad del curso y la mostramos. Vemos que primero pasa por el setter (en cuanto detecta la asignación) y luego por el getter.

```
cursoJS.difficultyLevel = 1; // Llamamos al setter
console.log(cursoJS.difficultyLevel); // Llamamos al getter
```

```
SETTER de difficultyLevel
GETTER de difficultyLevel
1
```

3.5.2. PREFIJO HASH(#)

Se trata de la última incorporación en el lenguaje JavaScript para ir avanzando en la robustez del lenguaje y permitir utilizar las propiedades como privadas.

La **sintaxis** es la siguiente: Usaremos el prefijo *hash* delante de la propiedad privada. De esta forma **podremos acceder a ellas a través de métodos**.

Ejemplo 1 → Con constructor.

```
class Cliente {
  #nombre;
  constructor(nombre, saldo){
    this.#nombre = nombre;
    this.saldo = saldo;
  }
}
```

Ejemplo 2 → Sin constructor

```
class Empresa {
  #nombre;
  #saldo;
```

Añadimos métodos para acceder a las propiedades (los métodos set y get son comunes para los dos ejemplos):

```
mostrarInformacion(){
  return `Cliente: ${this.#nombre}, tu saldo es de ${this.#saldo}`;
}
```

```
mostrarInformacion(){
  return `Empresa: ${this.#nombre}, tu saldo es de ${this.#saldo}`;
}
```

```
setSaldo(cantidad){
  if (!isNaN(cantidad)){
    this.#saldo = cantidad;
  }
}
```

```
getSaldo(){
  return this.#saldo;
}
```


Creamos un objeto de tipo Cliente y otro de tipo Empresa:

```
const cliente1 = new Cliente ("Javier", 50000); // instancia de la clase Cliente
```

```
const empresa1 = new Empresa();
```

Si queremos acceder a una propiedad privada, el propio editor nos avisa que no es posible y el navegador nos mostrará un error.

```
}
} Property '#nombre' is not accessible outside class 'Cliente'
because it has a private identifier. ts(18013)
const View Problem (Alt+F8) No quick fixes available
console.log(cliente1.#nombre);
```

```
✖ Uncaught SyntaxError: Private field '#nombre' must be declared in an enclosing class (at
03-app.js:62:21)
```

```
ERROR!! porque el atributo es privado
```

Invocamos los métodos:

Ejemplo 1 → Con constructor.

```
console.log(cliente1.mostrarInformacion());
cliente1.setSaldo(15000);
console.log(cliente1.getSaldo());
```

```
Cliente: Javier, tu saldo es de 50000
15000
```

Ejemplo 2 → Sin constructor (en este caso debemos primero asignar los datos del objeto).

```
empresa1.setNombre("AUTOS SPEED");
empresa1.setSaldo(350000);
console.log(empresa1.getNombre());
console.log(empresa1.getSaldo());
console.log(empresa1.mostrarInformacion());
```

```
AUTOS SPEED
350000
Empresa: AUTOS SPEED, tu saldo es de 350000
```

4. ARRAY OBJECT

Al igual que en el resto de los lenguajes, en JavaScript, un array es un tipo de objeto capaz de almacenar elementos o conjuntos de datos en una misma estructura.

Para acceder a un dato individual dentro del array, hay que indicar su **posición**. Esta posición es la conocemos como **índice**.

El **tipo** de elemento de los arrays es **Object**. `typeof (Array) = Object`

Los arrays en JavaScript son **dinámicos**, es decir, NO es necesario prever el tamaño de este en su creación, se podrá modificar después de su declaración, aunque se declaren con `const`.

Otra diferencia con otros lenguajes es que los arrays en JavaScript son **heterogéneos**, es decir, pueden almacenar valores de diferentes tipos.

Como ejemplos de arrays tenemos los carritos de compra, listado de amigos de red social o personas que dieron «me gusta» a una foto, tareas pendientes de un TO DO...

4.1. Declaración, modificación y acceso a los datos

Podremos declarar un array como cualquier otra variable, y lo podremos hacer con o sin elementos, o incluso, con elementos vacíos.

También podremos usar el constructor integrado (no es la opción recomendada por lo que no la usaremos). Aquí dejo varios ejemplos:

```
const alumnos = [];
const modulos = ["DWES", "DWEC", "DIW", "DAW", "EiE", "Inglés"];
const numeros = new Array(1,2,3,4);
const deTodo = ["Javier", 3, 5.25, null, {nom:"Pepe", profesion:"docente"}, [1,2,3]];
console.log(deTodo);
```

```
▼ Array(6) [ "Javier", 3, 5.25, null, {_, (3) [...]} ]
  0: "Javier"
  1: 3
  2: 5.25
  3: null
  ▼ 4: Object { nom: "Pepe", profesion: "docente" }
    nom: "Pepe"
    profesion: "docente"
    ▶ <prototype>: Object { _ }
  ▼ 5: Array(3) [ 1, 2, 3 ]
    0: 1
    1: 2
    2: 3
    length: 3
    ▶ <prototype>: Array []
  length: 6
```

- Ejecutamos `console.log` para visualizar el contenido
- Podemos ver que el **índice** que nos indica la posición de cada elemento **empieza en 0**.
- Al ser un **objeto heterogéneo** permite incluir datos de distintos tipos:
- El índice 4 incluye un objeto y el 5 otro array.

Podemos visualizar también el array a través de `console.table()`;

```
console.table(deTodo);
```

(Índice)	nom	profesion	0	1	2	Valores
0						Javier
1						3
2						5.25
3						null
4	Pepe	docente				
5			1	2	3	

4.1.1. ACCEDER A LOS ELEMENTOS DEL ARRAY

Para acceder a un elemento, usaremos el **índice**.

```
console.log(deTodo[2]);
console.log(deTodo[4]);
console.table(deTodo[5]);
console.log(deTodo[7]);
console.log(deTodo[5][0]);
```

Vemos que si accedemos a un elemento que no existe (índice 7), su valor es **undefined**.

Si queremos acceder a algún elemento del array dentro del array principal, accederemos con los **dos índices**.

5.25	01-app.js:16
▶ {nom: 'Pepe', profesion: 'docente'}	01-app.js:17
	01-app.js:18
(index)	Value
0	1
1	2
2	3
▶ Array(3)	
undefined	01-app.js:19
1	01-app.js:20

4.1.2. AÑADIR Y ELIMINAR ELEMENTOS A/DE UN ARRAY

Si queremos añadir o modificar el contenido usaremos el **índice**.

```
const alumnos = [];
```

```
alumnos[0] = "Antonio";
alumnos[1] = "Pedro";
alumnos[2] = "Rosa";
alumnos[3] = "Maria";
alumnos[10] = "José Antonio";
console.table(alumnos);
```

(index)	Value
0	'Antonio'
1	'Pedro'
2	'Rosa'
3	'Maria'
10	'José Antonio'
▶ Array(11)	

Si nos fijamos, se han creado los índices, pero no se listan. La propiedad `length` nos muestra el tamaño del array y es 11.

Debemos recordar que solo los **objetos** y **arrays** pueden ser modificados, aunque se hayan declarado con la palabra reservada **const**.

```
const notas = [0,2.5,5,7.5,10];
notas[1] = 3.5;
console.log(notas);
```

```
► (5) [0, 3.5, 5, 7.5, 10]
```

Por esa razón hemos podido modificar el segundo elemento del array. Sus elementos se podrán modificar, pero la referencia no cambia.

Si declaramos un array con **const**, lo que NO podremos hacer es crear un array y querer hacer referencia a otro array.

Esto NO ocurre si lo declaramos con **let**.

```
const mascotas = ["perro","gato","hámster","iguana","loro"];
mascotas = ["tortuga","cacaúta","pez"];
console.log(mascotas);
```

```
► Uncaught TypeError: Assignment to constant variable.
   at 03-app.js:56:9
```

Si asignamos un array a otro, cualquier modificación en cualquiera de ellos, se reflejará también en el otro, ya que los dos son referencia al mismo objeto.

```
const mascotas2 = mascotas;
mascotas[4] ="tortuga";
console.log(`El array "mascotas" contiene ${mascotas}`);
console.log(`El array "mascotas2" contiene ${mascotas2}`);
```

```
El array "mascotas" contiene perro,gato,hámster,iguana,tortuga
```

```
El array "mascotas2" contiene perro,gato,hámster,iguana,tortuga
```

```
mascotas2[0] = "tigre";
console.log(`El array "mascotas" contiene ${mascotas}`);
console.log(`El array "mascotas2" contiene ${mascotas2}`);
```

```
El array "mascotas" contiene tigre,gato,hámster,iguana,tortuga
```

```
El array "mascotas2" contiene tigre,gato,hámster,iguana,tortuga
```

Hemos visto cómo añadir y modificar elementos, lo haremos siempre a través del índice. Vamos ahora a eliminar un elemento del array.

```
const dias = ["lunes","martes","miércoles","jueves","viernes","sábado","domingo"];
delete dias[0];
console.log(dias);
```

```
► Array(7) [ <1 empty slot>, "martes", "miércoles", "jueves", "viernes", "sábado", "domingo" ]
```

El primer elemento del array ahora pasará a ser *undefined*.

Hemos visto que los arrays pueden contener elementos de distintos tipos, es decir, son **heterogéneos**.

Veamos un par de ejemplos...

```
let heter1 = [3,4,"natalia",true,Math.random()];
console.log(heter1);
let heter2 = [3.5,,"holaaa",2,"adiós"];
console.log(heter2);
console.log(heter2[2][0]);
```

```
▶ Array(5) [ 3, 4, "natalia", true, 0.6068180740200614 ]
▼ Array(3) [ 3.5, <1 empty slot>, (3) [...] ]
  0: 3.5
  ▶ 2: Array(3) [ "holaaa", 2, "adiós" ]
    length: 3
  ▶ <prototype>: Array []
holaaa
```

Vamos a ver cómo añadir y eliminar elementos de un array a través de **métodos** propios de este objeto y **Spread Operator**.

METODO / OPERADOR	FUNCIÓN
push(<elemento>)	Método que permite añadir un elemento al final del array
<pre>const muebles = ["silla","mesa"]; console.log(muebles); muebles.push("sillón"); console.log(muebles);</pre>	<pre>▶ (2) ['silla', 'mesa'] ▶ (3) ['silla', 'mesa', 'sillón']</pre>
pop()	Método que permite eliminar el último elemento del array
<pre>console.log(muebles); muebles.pop(); console.log(muebles);</pre>	<pre>▶ (4) ['silla', 'mesa', 'sillón', 'escritorio'] ▶ (3) ['silla', 'mesa', 'sillón']</pre>
unshift(<elemento>)	Método que permite añadir un elemento al inicio del array
<pre>const frutas = ["manzana", "sandía", "fresas"]; frutas.unshift("pera"); console.log(frutas);</pre>	<pre>▶ (4) ['pera', 'manzana', 'sandía', 'fresas']</pre>

shift(<elemento>)	Método que permite eliminar el primer elemento del array																
<pre>frutas.shift(); console.log(frutas);</pre>	<pre>▶ (3) ['manzana', 'sandía', 'fresas']</pre>																
splice(<inicio>,<numElem>)	Método que permite eliminar elementos. Indicamos desde qué posición se inicia el borrado y el número de elementos a eliminar.																
<pre>const hastaDiez = [1,2,3,4,5,6,7,8,9,10]; console.log(hastaDiez); //quitamos los dos últimos elementos hastaDiez.splice(hastaDiez.length-2,2); console.log(hastaDiez); //quitamos los dos primeros. hastaDiez.splice(0,2); console.log(hastaDiez);</pre>	<pre>▶ (10) [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] ▶ (8) [1, 2, 3, 4, 5, 6, 7, 8] ▶ (6) [3, 4, 5, 6, 7, 8]</pre>																
splice(<inicio>,<numElem>,<elemAñad, .. ,elemAñad>)	Método que permite eliminar elementos y añadir otros desde el lugar que indiquemos.																
<pre>const impares = [1,3,5,7,9,11,13,15,17,19]; console.log(impares); impares.splice(2,6,31,33,35); console.log(impares);</pre>	<pre>▶ (10) [1, 3, 5, 7, 9, 11, 13, 15, 17, 19] ▶ (7) [1, 3, 31, 33, 35, 17, 19]</pre>																
Los dos primeros parámetros sirven para eliminar elementos (como el caso anterior) y el resto son los elementos que vamos a añadir en el lugar donde se encontraban los eliminados. (borra 6 desde posición 2 y en su lugar pon 31,33,35).																	
Spread o Rest Operator (declarativo)	Operador muy utilizado, permite añadir elementos a un array, copiar arrays sin modificar el original, combinar propiedades de uno objeto, en funciones en las que no conocemos todos los argumentos, en destructuring.																
Partimos de un array vacío y creamos 3 objetos: Aplicamos Spread Operator:																	
<pre>let carrito = []; const producto1 = { nombre: "Disco duro SSD", capacidad: "1TB", pvp: 110 } const producto2 = { nombre: "Disco duro HDD", capacidad: "1TB", pvp: 50 } const producto3 = { nombre: "Disco duro M2", capacidad: "1TB", pvp: 175 }</pre>	<pre>carrito = [...carrito, producto1]; carrito = [producto3, ...carrito]; carrito = [...carrito, producto2]; console.table(carrito);</pre>																
<table><tr><th>(index)</th><th>nombre</th><th>capacidad</th><th>pvp</th></tr><tr><td>0</td><td>'Disco duro M2'</td><td>'1TB'</td><td>175</td></tr><tr><td>1</td><td>'Disco duro SSD'</td><td>'1TB'</td><td>110</td></tr><tr><td>2</td><td>'Disco duro HDD'</td><td>'1TB'</td><td>50</td></tr></table> <pre>▶ Array(3)</pre>		(index)	nombre	capacidad	pvp	0	'Disco duro M2'	'1TB'	175	1	'Disco duro SSD'	'1TB'	110	2	'Disco duro HDD'	'1TB'	50
(index)	nombre	capacidad	pvp														
0	'Disco duro M2'	'1TB'	175														
1	'Disco duro SSD'	'1TB'	110														
2	'Disco duro HDD'	'1TB'	50														

- Spread Operator (...) a la derecha de la igualdad para **combinar arrays**, ejemplo:

```
const fruits = ['manzana', 'plátano'];
const citricos = ['limón', 'naranja'];

const todasLasFrutas = [...fruits, ...citicros]; // Combinamos ambos arrays usando el Spread Operator

console.log(todasLasFrutas); // Salida: [ 'manzana', 'plátano', 'limón', 'naranja' ]
```

- Spread Operator (...) para crear una **copia superficial de un array**, útil para no modificar el array original, ejemplo:

```
const numerosOriginales = [1, 2, 3];
const numerosCopia = [...numerosOriginales]; // Creamos una copia del array original

numerosCopia.push(4); // Modificamos la copia eliminando el último elemento

console.log(numerosOriginales); // Salida: [ 1, 2, 3 ] (el array original no ha cambiado)
console.log(numerosCopia); // Salida: [ 1, 2, 3, 4 ]
```

4.2. Destructuring de arrays

En los **objetos** de tipo array, podemos crear la variable y asignarle el valor en un mismo paso a través del **destructuring**.

Ejemplo 1 → Destructuring en objetos

```
// Ejemplo sin destructuring
const estudiante = {
  nombre: 'Ana',
  edad: 21,
  carrera: 'Desarrollo de Aplicaciones Web'
};

const nombre = estudiante.nombre; // Se accede a cada propiedad del objeto por separado
const edad = estudiante.edad;
const carrera = estudiante.carrera;
console.log(nombre, edad, carrera); // Salida: Ana 21 Desarrollo de Aplicaciones Web

// Con destructuring definiremos las variables en una sola línea de la siguiente manera:
const { nom, age, studies } = estudiante;
console.log(nom, age, studies); // Salida: Ana 21 Desarrollo de Aplicaciones Web
```

Así como en los objetos hacemos referencia a la propiedad para acceder al valor, en los **arrays**, hacemos uso del **índice**.

Ejemplo 2 → Destructuring en arrays

```
const numbers = [10, 20, 30, 40, 50];
const [primero] = numbers;
console.log(primero); // Salida: 10
```

Otra diferencia es que en los objetos podemos acceder a cualquier propiedad sin tener en cuenta el orden; Sin embargo en los arrays, el **acceso** debe ser **secuencial y continuo** (para acceder a la posición 2, debo pasar por la 0 y la 1).

Solución: De la misma forma que cuando los creamos podemos dejar posiciones en blanco, cuando hacemos **destructuring** también podemos hacer lo mismo.

```
const [, , tercero] = numbers;
console.log(tercero); // 30
```

```
const [prim, seg, ter, , quin] = numbers;
console.log(prim, seg, ter, quin); // 10 20 30 50
```


4.2.1. DESTRUCTURING CON SPREAD OPERATOR

Al el operador ... en la desestructuración de un array, su propósito es agrupar los elementos restantes que no se han seleccionado explícitamente en una nueva variable, que será un array. Es como decir: "Dame los primeros elementos que he pedido, y pon el resto en una bolsa aparte"..

... a la **derecha** de "=" **EXPANDE** (Spread) un array o un objeto.

... a la **izquierda** de "=" **AGRUPA** (Rest) los elementos restantes en un nuevo array.

```
const ganadores = ['Carlos', 'Sofía', 'Daniel', 'Marta', 'Pedro', 'Ana'];
// Usamos la desestructuración para asignar el 1º y 2º puesto.
// El operador ...restantes agrupará el resto de los nombres en un nuevo array.
const [primerPuesto, segundoPuesto, ...restantes] = ganadores;

console.log('El ganador es:', primerPuesto);
console.log('El segundo puesto es:', segundoPuesto);
console.log('El resto de participantes son:', restantes);
/* Salida en la consola:
   El ganador es: Carlos
   El segundo puesto es: Sofía
   El resto de participantes son: [ 'Daniel', 'Marta', 'Pedro', 'Ana' ]
*/
```

4.3. Propiedad length

Propiedad que nos va a indicar el tamaño del array.

Funciona igual que en los Strings.

```
const dias = ["lunes","martes","miércoles","jueves","viernes","sábado","domingo"];

console.log(dias.length)//7
```

4.4. Métodos de array

Ya hemos visto algunos métodos a lo largo de la unidad didáctica cuyo propósito era añadir o eliminar elementos del array o recorrerlo.

Vamos a ver otros métodos cuyas funcionalidades son muy diversas.

Debemos tener en cuenta que, aquellos métodos que devuelven otro array permiten el encadenamiento de los métodos.

Se tiene en cuenta Spread o Rest Operator (...) que, si bien no es un método, es muy utilizado por su amplia funcionalidad.

MÉTODO	FUNCIÓN
instanceof	Método común para todos los tipos de datos de JavaScript. Devuelve un booleano según pertenezca a la clase indicada
<pre>const notas2 = [2,1,,9,8.5,6.12,,2,5,,7];</pre> <pre>console.log(`El array "notas2" es de tipo \${typeof(notas2)} y ...\\n pertenece a la clase Array?? \${notas2 instanceof Array}`);</pre> <pre>El array "notas2" es de tipo object y ...</pre> <pre>pertenece a la clase Array?? true</pre>	
BÚSQUEDA DE ELEMENTOS	
includes(<elemento>)	Método que devuelve true si el elemento se encuentra en el array. NO funciona en arrays de objetos.
<pre>const pares = [2,4,2,6,8,2,10,8,6,12,10,2];</pre> <pre>console.log(pares.includes(8)); //true</pre> <pre>console.log(pares.includes(30)); //false</pre>	
some(arrow function)	Método que devuelve true si el elemento cuya condición se expresa, se encuentra en el array. Para todo tipo de arrays
<pre>const carroCompra = [{articulo: "raton", pvp: 15}, {articulo: "teclado", pvp: 20}, {articulo: "altavoces", pvp: 50}, {articulo: "hub USB", pvp: 15}];</pre> <pre>const existe = carroCompra.some(art => art.articulo === 'raton');</pre> <pre>console.log(`¿He añadido el ratón al carrito? \${existe}`);</pre> <pre>¿He añadido el ratón al carrito? true</pre> <p>Este método lo usaremos para arrays de objetos, para el resto, usaremos includes(), por la reducción de código.</p>	

every(arrow function)	Método que devuelve true si TODOS los elementos del array cumplen la condición expresada.
<pre>const result3 = carroCompra.every(prod => prod.pvp < 100); console.log(result3);</pre> <p>Si solo queremos valorar un elemento, usamos some().</p>	
findIndex(arrow function)	Método que devuelve el primer índice que cumpla la condición de la función. Si no lo encuentra, devuelve -1.
<pre>const pares = [2,4,2,6,8,2,10,8,6,12,10,2];</pre> <pre>const mayor10 = pares.findIndex(num => num > 10); console.log(`findIndex: El primer num mayor que 10 está en la posic: \${mayor10}`);</pre> <pre>findIndex: El primer num mayor que 10 está en la posic: 9</pre> <pre>const indice2 = carroCompra.findIndex(art => art.pvp === 150); console.log(`findIndex: Encontrado en el indice \${indice2}`);</pre> <pre>findIndex: Encontrado en el indice -1</pre>	
indexOf(<elemento>)	Método que devuelve el primer índice del elemento pasado como argumento. Si no lo encuentra, devuelve -1
<pre>const semana = ["lunes","martes","miércoles","jueves","viernes","sábado","domingo"]; console.log(semana.indexOf("lunes")); //0</pre>	
lastIndexOf(<elemento>)	Método que devuelve el primer índice del elemento desde el final.
<pre>const pares2 = [2,4,2,6,8,2,10,8,6,12,10,2]; console.log(pares2.lastIndexOf(2)); //11</pre>	

OBTENER SUBARRAYS

find(arrow function)

Método que devuelve el elemento del primer índice resultado de la condición expresada.

```
const carroCompra = [
  {articulo: "ratón", pvp: 15},
  {articulo: "teclado", pvp: 20},
  {articulo: "altavoces", pvp: 50},
  {articulo: "hub USB", pvp: 15}
];

const result = carroCompra.find(producto => producto.pvp === 15);
console.log(result);
```

```
▶ {articulo: 'ratón', pvp: 15}
```

Si queremos TODOS los elementos, usaremos FILTER (explicado a continuación)

filter(arrow function)

Método que obtiene un nuevo array con los elementos del array que cumplan una condición determinada

Este método se utiliza mucho para realizar las búsquedas filtradas en las páginas web: apartamentos de 2 habitaciones, precio menor de X...

Ejemplo 1 → Mostrar los elementos cuyo valor es mayor o igual a 5

```
const notas4 = [2,1,,9,8.5,6.12,5,2,5,,7];
const aprobados = notas2.filter(x => x >= 5);
console.log(aprobados);
```

```
▼ (5) [9, 8.5, 6.12, 5, 7]
0: 9
1: 8.5
2: 6.12
3: 5
4: 7
length: 5
```

Ejemplo 2 → Eliminar un elemento de un carro de compra.

```
const carroCompra = [
  {articulo: "ratón", pvp: 15},
  {articulo: "teclado", pvp: 20},
  {articulo: "altavoces", pvp: 50},
  {articulo: "hub USB", pvp: 15}
];

const carritoFinal = carroCompra.filter(prod => prod.articulo !== "altavoces");
console.table(carritoFinal);
```

(index)	articulo	pvp
0	'ratón'	15
1	'teclado'	20
2	'hub USB'	15

```
▶ Array(3)
```

reduce(arrow function)	Método que obtiene otro array resultado de hacer un cálculo con cada elemento del array que se está recorriendo (acumulando el resultado).
<p>Ejemplo 1→ Sumar todos los elementos del array.</p> <pre>const hastaCinco = [1,2,3,4,5]; const suma = hastaCinco.reduce((acum,elemento)=>acum + elemento,0); console.log(suma); //15</pre> <p>Elementos del método:</p> <ul style="list-style-type: none"> 1º parámetro: Arrow function que realiza la acción en cada iteración. <ul style="list-style-type: none"> El primer parámetro es el acumulador de las sumas El segundo es el que recoge el valor de cada elemento del array que se irá sumando. 2º parámetro: Indica el valor inicial que tendrá la variable que acumula el valor de cada suma. <p>Ejemplo 2→ Total de un carrito de compra</p> <pre>const carroCompra = [{articulo: "ratón", pvp: 15}, {articulo: "teclado", pvp: 20}, {articulo: "altavoces", pvp: 50}, {articulo: "hub USB", pvp: 15}]; let total = carroCompra.reduce((total, articulo) => total + articulo.pvp ,0); console.log(`Total compra: \${total}€`); //comparado con forEach let totalCompra = 0; carroCompra.forEach(articulo => totalCompra += articulo.pvp); console.log(`Total compra: \${totalCompra}€`);</pre>	
slice(<inicio>,<fin+1>)	Método que genera OTRO array con la copia de los elementos del array. No modifica el array original. <p>Se puede acotar la copia indicando inicio y fin (el elemento "fin" NO lo copia)</p>
<p>Ejemplo 1→ NO hay argumentos → Hace una copia de todo el array</p> <pre>const meses = ["enero","feb","mar","abril","mayo","junio","julio","ago","sept","oct","nov","dic"]; const anyo = meses.slice(); console.log(anyo);</pre> <pre>► (12) ['enero', 'feb', 'mar', 'abril', 'mayo', 'junio', 'julio', 'ago', 'sept', 'oct', 'nov', 'dic']</pre> <p>Ejemplo 2→ Con argumentos → Copia desde el índice del primer argumento hasta el índice -1 del segundo argumento.</p> <pre>const trim1 = meses.slice(0,3); console.log(trim1);</pre> <pre>► (3) ['enero', 'feb', 'mar']</pre>	

Ejemplo 3 → NO hay argumento fin → Copia desde el argumento hasta el final del array original

```
const trim4 = meses.slice(9);
console.log(trim4);
```

```
► (3) ['oct', 'nov', 'dic']
```

CONVERTIR ARRAY EN STRING

join(<separador>)

Por defecto el separador es una coma, pero podemos indicar otro separador

```
console.log(trim1.join());
console.log(trim4.join("; "));
console.log(trim1.join(" <-> "));
```

```
enero, feb, mar
oct; nov; dic
enero <-> feb <-> mar
```

toString()

Lo separa directamente por comas

```
console.log(trim1.toString());
```

```
enero, feb, mar
```

UNIR ARRAYS

concat(<2º array>)

Método que permite añadir los elementos de uno o varios arrays al array que invoca la función
Genera otro array en el orden indicado

Ejemplo 1 → Concatenamos un array con otro

```
const planetas1 = new Array("Mercurio", "Venus", "Tierra", "Marte");
const planetas2 = new Array("Júpiter", "Saturno", "Urano", "Neptuno");
const planetas = planetas1.concat(planetas2);
console.log(planetas);
```

```
► (8) ['Mercurio', 'Venus', 'Tierra', 'Marte', 'Júpiter', 'Saturno', 'Urano', 'Neptuno']
```

Ejemplo 2 → Concatenamos un array con dos arrays.

```
const diasSemana1 = ['Lunes', 'Martes', 'Miércoles'];
const diasSemana2 = ['Jueves', 'Viernes'];
const finDeSemana = ['Sábado', 'Domingo'];
const semanaConcat = diasSemana1.concat(diasSemana2, finDeSemana);
console.log(semanaConcat);
```

```
► (7) ['Lunes', 'Martes', 'Miércoles', 'Jueves', 'Viernes', 'Sábado', 'Domingo']
```

Ejemplo3 → Añadimos un elemento a la vez que concatenamos

```
const semanaConcat2 = diasSemana1.concat(diasSemana2, 'Fin de Semana');
console.log(semanaConcat2);
```

```
► (6) ['Lunes', 'Martes', 'Miércoles', 'Jueves', 'Viernes', 'Fin de Semana']
```

Spread Operator

Operador que genera otro array (en el orden indicado) con los arrays que pongamos como elementos.

Ejemplo 1 → Concatenamos varios arrays

```
const semanaRestOperator = [...diasSemana1, ...diasSemana2, ...finDeSemana];
console.log(semanaRestOperator);
```

```
► (7) ['Lunes', 'Martes', 'Miércoles', 'Jueves', 'Viernes', 'Sábado', 'Domingo']
```

Ejemplo 2 → Añadimos un elemento a la vez que concatenamos (NO ponemos los tres puntos)

```
const semanaRestOperator2 = [...diasSemana1, ...diasSemana2, 'Fin de Semana'];
console.log(semanaRestOperator2);
```

```
► (6) ['Lunes', 'Martes', 'Miércoles', 'Jueves', 'Viernes', 'Fin de Semana']
```

Ejemplo 3 → Añadimos un array de elementos a la vez que concatenamos

```
const semanaRestOperator3 = [...diasSemana1, ...diasSemana2, ...'Fin de Semana'];
console.log(semanaRestOperator3);
```

```
► (18) ['Lunes', 'Martes', 'Miércoles', 'Jueves', 'Viernes', 'F', 'i', 'n', ' ', 'd', 'e', ' ', 'S', 'e', 'm', 'a', 'n', 'a']
```

MODIFICAR EL ORDEN DE LOS ELEMENTOS

reverse()

Método que invierte el orden de los elementos del array.

```
const week = ["lunes", "martes", "miércoles", "jueves", "viernes", "sábado", "domingo"];
console.log(week.reverse());
```

```
► (7) ['domingo', 'sábado', 'viernes', 'jueves', 'miércoles', 'martes', 'lunes']
```

sort()

Método que ordena los elementos del array según la tabla Unicode.

Ejemplo 1 → Ordenamos un array de strings en minúsculas

```
console.log(week.sort());
```

```
► (7) ['domingo', 'jueves', 'lunes', 'martes', 'miércoles', 'sábado', 'viernes']
```

Ejemplo 2 → Ordenamos un array de strings con mayúsculas, minúsculas y letras acentuadas

Código Unicode: Las minúsculas son mayores que las mayúsculas, una vocal con tilde está por detrás de la Z, la letra ñ también está por detrás.

```
let animales2 = ["Ñu", "Águila", "boa", "oso", "mariposa", "Nutria"];
console.log(animales2.sort());
```

```
► (6) ['Nutria', 'boa', 'mariposa', 'oso', 'Águila', 'Ñu']
```

Ejemplo 3 → Ordenamos un array de números, observamos que los toma como strings.

```
const impares2 = [1,3,5,7,9,7,5,3,1,11,13,15];
console.log(impares2.sort());
```

```
► (12) [1, 1, 11, 13, 15, 3, 3, 5, 5, 7, 7, 9]
```

sort con método localeCompare()

Método que ordena los elementos de un array

Utilizamos el método localeCompare para ordenar **strings** que, devuelve un número negativo, cero o positivo si el parámetro que invoca el método es menor, igual o mayor que el parámetro del método. Se le añade el argumento del idioma.

```
function ordena(a,b){
  return a.localeCompare(b,"es");
}
console.log(ordena("a","b")); //-1
```

Vamos a utilizarlo con el método sort()

Ejemplo 1 → Ordenamos array con strings.

```
animales2.sort((a,b) => a.localeCompare(b,"es"));
console.log(animales2);
```

```
const animales2 = ["Ñu", "Águila", "boa", "oso", "mariposa", "Nutria"];
```

```
► (6) ['Águila', 'boa', 'mariposa', 'Nutria', 'Ñu', 'oso']
```

Ejemplo 2 fallido → Ordenamos array con **números**. Recordamos que localeCompare es SOLO PARA STRINGS

```
impares2.sort((a,b) => a.localeCompare(b,"es"));
console.log(impares2);
```

```
✖ Uncaught TypeError: a.localeCompare is not a function
    at 05-app.js:187:26
    at Array.sort (<anonymous>)
    at 05-app.js:187:10
```

Ejemplo 3 exitoso → Ordenamos array con números (de menor a mayor)

```
impares2.sort((a,b) => a - b);
console.log(impares2);
```

```
► (12) [1, 1, 3, 3, 5, 5, 7, 7, 9, 11, 13, 15]
```

4.5. Recorrer un array

Para recorrer un array tenemos distintas posibilidades dentro de la estructura iterativa **for**.

Además, contamos con algunos **métodos** propios del objeto de tipo array para recorrerlos.

4.5.1. FOR

Estructura del **iterador básico** que hará uso de la propiedad `length` de los arrays para hacer dinámica la acción de recorrer el array.

Recorre todos los elementos del array.

```
const notas2 = [2,1,,9,8.5,6.12,,2,5,,7];
for (let i=0;i<notas2.length;i++){
  if (notas2[i] === undefined){
    notas2[i] = "NO presentado";
  }
  console.log(`La nota ${i} es: ${notas2[i]}`);
}
```

```
La nota 0 es: 2
La nota 1 es: 1
La nota 2 es: NO presentado
La nota 3 es: 9
La nota 4 es: 8.5
La nota 5 es: 6.12
La nota 6 es: NO presentado
La nota 7 es: 2
La nota 8 es: 5
La nota 9 es: NO presentado
La nota 10 es: 7
```

4.5.2. FOR ... IN

Esta estructura **itera sobre objetos**.

La **variable** que recorre el array solo almacena los **índices**, NO los valores.

Tendremos en cuenta que se salta los elementos vacíos. **Ejemplo:**

```
const notas2 = [2,1,,9,8.5,6.12,,2,5,,7];
```

```
for (let indice in notas2){
  console.log(`La nota ${indice} de "notas2" es: ${notas2[indice]}`);
}
```

```
La nota 0 de "notas2" es: 2
La nota 1 de "notas2" es: 1
La nota 3 de "notas2" es: 9
La nota 4 de "notas2" es: 8.5
La nota 5 de "notas2" es: 6.12
La nota 7 de "notas2" es: 2
La nota 8 de "notas2" es: 5
La nota 10 de "notas2" es: 7
```


4.5.3. FOR ... OF

Esta estructura itera sobre **Arrays y Strings**.

La **variable** que recorre el array NO almacena los índices, **almacena los propios valores del array**.

Debemos tener en cuenta que NO se salta los undefined.

```
const notas2 = [2,1,,9,8.5,6.12,,2,5,,7];
```

```
for (let nota of notas2){
  console.log(`${nota}`);
}
```

```
2
1
undefined
9
8.5
6.12
undefined
2
5
undefined
7
```

4.5.4. FOREACH

Este método **propio de los arrays** utiliza una función para recorrer el array y realizar las acciones para cada elemento del array.

Podemos utilizar uno o dos parámetros (si queremos tener en cuenta el índice del array o no).

Si hay elementos vacíos, NO los tendrá en cuenta.

Ejemplo 1 → Con **índice** si queremos hacer referencia a la posición.

```
const notas2 = [2,1,,9,8.5,6.12,,2,5,,7];
```

```
notas2.forEach(function(elemento,indice){
  console.log(`La nota ${indice} de "notas2" es: ${elemento}`);
});
```

```
La nota 0 de "notas2" es: 2
La nota 1 de "notas2" es: 1
La nota 3 de "notas2" es: 9
La nota 4 de "notas2" es: 8.5
La nota 5 de "notas2" es: 6.12
La nota 7 de "notas2" es: 2
La nota 8 de "notas2" es: 5
La nota 10 de "notas2" es: 7
```

Ejemplo 2 → Recorremos un array de objetos.

```
const animales = [
  {nombre:"Fly", tipo:"perro"},
  {nombre:"Misi", tipo:"gato"},
  {nombre:"Cuqui", tipo:"canario"},
  {nombre:"McGuiver", tipo:"tortuga"},
  {nombre:"Olaf", tipo:"perro"}
];
```

```
animales.forEach(function(mascota){
  console.log(`${mascota.nombre} es un/a ${mascota.tipo}`)
});
```

```
Fly es un/a perro
Misi es un/a gato
Cuqui es un/a canario
McGuiver es un/a tortuga
Olaf es un/a perro
```

Ejemplo 3 → Ejemplo 2 con arrow function (forma más común)

```
animales.forEach(mascota => console.log(`${mascota.nombre} es un/a ${mascota.tipo}`));
```

4.5.5. MAP

El método **map()** crea un nuevo array aplicando una función a cada elemento del array. Es parecido a **forEach()** pero, en el caso de **map()** podemos asignar el resultado a una variable y con **forEach()** NO.

Ejemplo 1 → Comparación entre **forEach** y **map**.

```
const nuevoAnimales = animales.map(function(mascota){
  return `${mascota.nombre} es un/a ${mascota.tipo}`;
});
console.log(nuevoAnimales);
const nuevoAnimales2 = animales.forEach(function(mascota){
  return `${mascota.nombre} es un/a ${mascota.tipo}`;
});
console.log(nuevoAnimales2);
```

```
► (5) ['Fly es un/a perro', 'Misi es un/a gato', 'Cuqui es un/a canario', 'McGuiver es un/a tortuga', 'Olaf es un/a perro']
undefined
```

Ejemplo2 → Arrow function.

```
const notas3 = [3,1,7,6.5,8,10];
console.log(notas3);
const mitad = notas3.map(x=>x/2);
console.log(mitad);
```

```
► (6) [3, 1, 7, 6.5, 8, 10]
► (6) [1.5, 0.5, 3.5, 3.25, 4, 5]
```

5. DATE OBJECT

Este tipo de objetos nos van a servir para manejar fechas.

Un objeto de tipo Date representa un momento concreto del tiempo.

5.1. Declaración y sintaxis

Para crear objetos de tipo Date, necesitamos el constructor integrado:

Ejemplo 1 → sin argumentos

```
let hoy = new Date();
console.log(hoy);
```

```
► Date Sat Oct 24 1970 17:09:12 GMT+0100 (hora estándar de Europa central)
```

Ejemplo 2 → con argumentos (mes, día, año)

```
const fecha = new Date('12-31-2025');
console.log(fecha);
```

```
► Date Wed Dec 31 2025 00:00:00 GMT+0100 (hora estándar de Europa central)
```

```
const fecha2 = new Date('December 31 2025');
console.log(fecha2);
```

```
► Date Wed Dec 31 2025 00:00:00 GMT+0100 (hora estándar de Europa central)
```

Ejemplo 3 → con argumentos (año, mes, día, hora, minutos) Ojo, mes inicia en 0

```
let fecha3 = new Date(2024,11,31,23,59);
console.log(fecha3);
```

```
► Date Tue Dec 31 2024 23:59:00 GMT+0100 (hora estándar de Europa central)
```

Ejemplo 4 → único argumento numérico (milisegundos desde 1 de enero de 1970)

```
let fechaDesde010170 = new Date(1000000000000);
console.log(fechaDesde010170);
```

```
Sun Sep 09 2001 03:46:40 GMT+0200 (hora de verano de Europa central)
```

5.2. Métodos get y set

A través de los métodos *get* y *set* podremos obtener información sobre las fechas o bien modificarlas.

5.2.1. GET

```
console.log(hoy);
console.log(hoy.getFullYear());
console.log(hoy.getMonth());
console.log(hoy.getDate());
console.log(hoy.getDay());
console.log(hoy.getHours());
console.log(hoy.getMinutes());
console.log(hoy.getSeconds());
console.log(hoy.getMilliseconds());
console.log(hoy.getTime());
console.log(Date.now());
```

Para tener en cuenta:

Los meses empiezan en 0

Date → día del mes

Day → día de la semana

```
► Date Sun Aug 31 2025 22:14:14 GMT+0200 (hora de verano de Europa central)
getFullYear()
2025
getMonth() --> Inicia en 0
7
getDate() --> Día del mes
31
getDay() --> Día de la semana
0
getHours()
22
getMinutes()
14
getSeconds()
14
getMilliseconds()
39
getTime() -> Los milisegundos desde 01/01/1970
1756671254039
Date.now() -> Los milisegundos desde 01/01/1970
1756671254041
```

5.2.2. SET

```
console.log(hoy);
hoy.setDate(17);
hoy.setMonth(hoy.getMonth()+2);
hoy.setFullYear(2025);
hoy.setHours(23);
hoy.setMinutes(30);
hoy.setSeconds(50);
console.log(hoy);
hoy.setTime(25632552322);
console.log(hoy);
```

Para tener en cuenta:

setTime → aplica los milisegundos desde 01/01/1970

```
► Date Sun Aug 31 2025 22:14:14 GMT+0200 (hora de verano de Europa central)
► Date Fri Oct 17 2025 23:30:50 GMT+0200 (hora de verano de Europa central)
► Date Sat Oct 24 1970 17:09:12 GMT+0100 (hora estándar de Europa central)
```

5.3. Métodos para formatear las fechas

Existen métodos que nos permiten formatear la salida de las fechas, de forma que nos sea más sencilla su lectura.

Los métodos más comunes son:

- ✓ **toString()** → Muestra en formato de texto propio de JavaScript
Mon Oct 30 2023 11:00:45 GMT+0100 (hora estándar de Europa central)
- ✓ **toTimeString()** → Muestra la hora en formato de texto
11:00:45 GMT+0100 (hora estándar de Europa central)
- ✓ **toDateString()** → Muestra la fecha (sin hora)
Mon Oct 30 2023
- ✓ **toLocaleString()** → Muestra la fecha y hora en formato texto. Por defecto el idioma del equipo
30/10/2023, 11:00:45
- ✓ **toLocaleString('es')** → Muestra la fecha y hora en formato texto
30/10/2023, 11:00:45
- ✓ **toLocaleString('en')** → Muestra la fecha y hora en formato texto
10/30/2023, 11:00:45 AM

- ✓ `toLocaleDateString()` → Muestra la fecha en formato texto (sin hora)

```
30/10/2023
```

- ✓ `toLocaleTimeString()` → Muestra la hora en formato texto

```
8:27:40
```

- ✓ `toGMTString()` → Muestra la fecha según el meridiano de Greenwich

```
Mon, 30 Oct 2023 10:00:45 GMT
```

6. MATH OBJECT

Este objeto facilita la ejecución de algunas operaciones matemáticas de alto nivel.

Además, nos va a permitir formatear la salida de datos numéricos.

6.1. Declaración variables de tipo math

Math es un objeto global, por lo que no necesitaremos un constructor (no tiene).

```
const numAleat = Math.random();
console.log(numAleat);
```

```
0.31289834519219406
```

6.1.1. CONSTANTES DE MATH

CONSTANTE	SIGNIFICADO
E	Valor matemático e
LN10	Logaritmo neperiano de 10
LN2	Logaritmo neperiano de 2

LOG10E	Logaritmo decimal de e
LOG2E	Logaritmo binario de e
PI	La constante PI (π)
SQRT1_2	Resultado de la división de uno entre la raíz cuadrado de dos
SQRT2	Raíz cuadrada de 2

Ejemplo → Cálculo de la circunferencia de un círculo o su área que utiliza la constante PI:

```
//Calcular la circunferencia de un círculo --> 2*PI*radio
function Circulo(r){
  this.radio = r;

  this.calcCircunfer = function(){
    return 2*Math.PI*this.radio;
  }
}
let miCirculo = new Circulo(10);
console.log(miCirculo.calcCircunfer());
//62.83185307179586
```

6.2. Métodos del objeto Math

MÉTODO	SIGNIFICADO	EJEMPLO USO
random()	Devuelve un número aleatorio decimal entre 0 y 1 (no llega al 1) Números de 0 a 10 (llega a 9.99999) Números entre 5 y 10 (incluye 10)	<code>console.log(Math.random());</code> <code>console.log(Math.random()*10);</code> <code>console.log(Math.random()*(10-5+1)+5);</code>
max(a,...,b) min(a,...,b)	Devuelve el número mayor Devuelve el número menor	<code>const a = 5.2, b = 6.7, c = 8.5;</code> <code>console.log(Math.max(a,b,c)); //8.5</code> <code>console.log(Math.min(a,b,c)); //5.2</code>

round()	Redondea a su entero más próximo (alza o baja)	<pre>console.log(Math.round(a)); //5 console.log(Math.round(b)); //7 console.log(Math.round(c)); //9</pre>
floor() ceil()	Redondea siempre a la baja Redondea siempre al alza	<pre>console.log(Math.floor(a)); //5 console.log(Math.ceil(a)); //6 console.log(Math.floor(b)); //6 console.log(Math.ceil(b)); //7 console.log(Math.floor(c)); //8 console.log(Math.ceil(c)); //9</pre>
trunc()	Trunca el número	<pre>console.log(Math.trunc(a)); //5 console.log(Math.trunc(b)); //6 console.log(Math.trunc(c)); //8</pre>
pow(b,e)	Devuelve la potencia	<pre>console.log(Math.pow(a,2)); //27.04</pre>
sqrt(a) cbrt(a)	Devuelve la raíz cuadrada de a Devuelve la raíz cúbica de a	<pre>console.log(Math.sqrt(121)); //11 console.log(Math.cbrt(27)); //3</pre>
abs(a)	Devuelve el valor absoluto de a	<pre>console.log(Math.abs(-25)); //25</pre>
sign(a)	Indica el signo de a: <ul style="list-style-type: none"> • 1 si + • -1 si - • 0 o -0 	<pre>console.log(Math.sign(a)); //1 console.log(Math.sign(0)); //0 console.log(Math.sign(-0)); //-0 console.log(Math.sign(-25)); //-1 console.log(Math.sign(NaN)); //NaN</pre>

Por último, nombrar también los métodos trigonométricos:

sin(), cos(), tan(), asin(), acos(), atan()

7. SET OBJECT

Los Sets o conjuntos son una estructura de datos (son **objetos**) que permiten, al igual que los arrays, almacenar datos.

Los conjuntos **NO admiten valores duplicados**. De hecho para considerar dos elementos iguales se cumple la igualdad estricta (===).

7.1. Declaración

La sintaxis de la declaración de un conjunto es utilizando el constructor integrado.

Sus elementos deben ser iterables y NO acepta valores repetidos.

```
let lista = new Set();
```

No podemos crear un conjunto y almacenar directamente valores si éstos NO son iterables, nos dará error:

Ejemplo 1 → Set de números

```
const miSet = new Set(2,4,6,8);
console.log(miSet);
```

```
✖ ▶ Uncaught TypeError: number 2 is not iterable (cannot read property
  Symbol(Symbol.iterator))
    at new Set (<anonymous>)
    at 01-app.js:6:15
```

Ejemplo 2 → Set de números (pasados como array)

```
const miSet2 = new Set ([2,4,6,8,2,4,6]);
console.log(miSet2);
```

```
▼ Set(4) {2, 4, 6, 8} ⓘ
  ▼ [[Entries]]
    ▶ 0: 2
    ▶ 1: 4
    ▶ 2: 6
    ▶ 3: 8
    size: 4
    ▶ [[Prototype]]: Set
  console.log(typeof miSet2); //object
  console.log(miSet2 instanceof Set); //true
```

Ejemplo 3 → Set de Strings (solo almacena la primera, es lo que puede iterar directamente).

```
let letras = new Set("dfjasdfjasdfasdf", "FFFFF");
console.log(letras);
```

```
▶ Set(5) {'d', 'f', 'j', 'a', 's'}
```

7.2. Métodos de set object

7.2.1. SET.ADD()

A través de este método, iremos **añadiendo elementos** a los conjuntos.

Tendremos en cuenta que los conjuntos son, al igual que los arrays, estructuras de datos heterogéneos.

Ejemplo 1 → Declarar set vacío e ir añadiendo elementos

```
let prueba = new Set();
prueba.add(Math.random());
prueba.add(8);
prueba.add(Math.random());
prueba.add("natalia");
console.log(prueba);
```

```
▶ Set(4) [ 0.2714362076452397, 8, 0.6406336379664426, "natalia" ]
```

Ejemplo 2 → Añadir elementos a un set no vacío.

```
letras.add("Natalia");
console.log(letras);
```

```
▼ Set(6) {'d', 'f', 'j', 'a', 's', ...} ⓘ
  ▼ [[Entries]]
    ▶ 0: "d"
    ▶ 1: "f"
    ▶ 2: "j"
    ▶ 3: "a"
    ▶ 4: "s"
    ▶ 5: "Natalia"
    size: 6
    ▶ [[Prototype]]: Set
```

Ejemplo 3 → Añadir objetos al conjunto (directamente no podemos hacer un set de objetos)

```
const artic1 = {
  prenda: 'Camisa',
  pvp: '60€'
}
```

```
const artic2 = {
  prenda: 'Corbata',
  pvp: '25€'
}
```

```
const carritoSet = new Set();
carritoSet.add(artic1).add(artic2);
console.log(carritoSet);
```

```
▼ Set(2) {{...}, {...}} ⓘ
  ▼ [[Entries]]
    ▼ 0:
      ▶ value: {prenda: 'Camisa', pvp: '60€'}
    ▼ 1:
      ▶ value: {prenda: 'Corbata', pvp: '25€'}
  size: 2
  ▶ [[Prototype]]: Set
```

7.2.2. SET.HAS()

A través de este método podremos verificar si el conjunto contiene o no un elemento concreto.

Recordemos el set “letras”

```
▼ Set(6) {'d', 'f', 'j', 'a', 's', ...} ⓘ
  ▼ [[Entries]]
    ▶ 0: "d"
    ▶ 1: "f"
    ▶ 2: "j"
    ▶ 3: "a"
    ▶ 4: "s"
    ▶ 5: "Natalia"
  size: 6
  ▶ [[Prototype]]: Set
```

```
console.log(letras.has('Natalia'))//true
```

7.2.3. SET.DELETE()

A través de este método podremos **eliminar elementos** del conjunto.

Tenemos un set llamado “lista”

```
► Set(4) {8, 3.5, 5, 1}
```

```
if (lista.delete(8)){
  console.log(`Se ha eliminado el elemento "8" con éxito`);
}
console.log(lista);
```

```
Se ha eliminado el elemento "8" con éxito
```

```
► Set(3) {3.5, 5, 1}
```

7.2.4. SET.CLEAR()

A través de este método podemos **vaciar el conjunto**.

Seguimos con el mismo set.

```
console.log(lista);
console.log("Vamos a eliminar el contenido de 'lista'...");
lista.clear();
console.log(lista);
```

```
► Set(3) {3.5, 5, 1}
```

```
Vamos a eliminar el contenido de 'lista'...
```

```
► Set(0) {size: 0}
```

7.3. Propiedad size

A través de esta propiedad, podremos saber el tamaño de un conjunto.

```
let pares = new Set();
pares.add(8).add(2).add(4);
console.log(pares);
console.log(pares.size);
```

```
► Set(3) [ 8, 2, 4 ]
3
```

7.4. Convertir set en array

Utilizaremos el Spread o Rest Operator

```
console.log("-----")
let num = new Set([5,8,6,6,6,3,36,22,5,,2,]);
console.log(num);
let numArray = [...num];
console.log(numArray);
console.log(numArray instanceof Array);
```

► Set(8) {5, 8, 6, 3, 36, ...}
 ► (8) [5, 8, 6, 3, 36, 22, undefined, 2]
 true

7.5. Recorrer un set

Utilizaremos el bucle *for ... of* o *forEach* para poder recorrer cada elemento de nuestro conjunto.

```
let num = new Set([5,8,6,6,6,3,36,22,5,,2,]);
console.log(num);
```

```
▼ Set(8) {5, 8, 6, 3, 36, ...} ⓘ
  ▼ [[Entries]]
    ► 0: 5
    ► 1: 8
    ► 2: 6
    ► 3: 3
    ► 4: 36
    ► 5: 22
    ► 6: undefined
    ► 7: 2
    size: 8
    ► [[Prototype]]: Set
```

```
for (let i of num){
  console.log(i);
}
```

```
num.forEach(i => console.log(i));
```

```
5
8
6
3
36
22
undefined
2
```

8. MAP OBJECT

Al igual que los conjuntos, los mapas aparecen en el estándar ECMA2015 o ES6.

Los mapas permiten crear **estructuras de clave-valor**.

Las claves no se pueden repetir y tendrán asociado un valor (que sí se podrá repetir). Podríamos decir que son como objetos con una sola propiedad.

Esta estructura de datos es **heterogénea**, es decir, tanto las claves como los valores pueden ser de cualquier tipo.

Para hacer referencia a los elementos del mapa, nos referiremos siempre a las claves NO a los valores.

8.1. Declarar e instanciar map

Se utiliza el constructor.

```
let andalucia = new Map();
```

8.2. Métodos de map

8.2.1. MAP.SET()

A través de este método podremos ir añadiendo elementos al map.

Ejemplo 1 → clave numérica, valor String

```
andalucia.set(4,"Almería");
andalucia.set(11,"Cádiz");
andalucia.set(14,"Córdoba");
andalucia.set(29,"Málaga");
andalucia.set(41,"Sevilla");
andalucia.set(18,"Granada");
andalucia.set(21,"Huelva").set(18,"Jaén");

console.log(andalucia);
console.log(typeof andalucia);//object
console.log(andalucia instanceof Map);//true
```

```
▼ Map(7) ⓘ
  ▼ [[Entries]]
    ▶ 0: {4 => "Almería"}
    ▶ 1: {11 => "Cádiz"}
    ▶ 2: {14 => "Córdoba"}
    ▶ 3: {29 => "Málaga"}
    ▶ 4: {41 => "Sevilla"}
    ▶ 5: {18 => "Jaén"}
    ▶ 6: {21 => "Huelva"}
    size: 7
  ▶ [[Prototype]]: Map
```

Ejemplo 2 → Utilizando arrays

```
const notas = new Map([["baja",3.5],["media",7.5],["alta",9]]);
notas.set(["raspados",5]); //undefined
notas.set("raspado",5);
console.log(notas);
```

```
▼ Map(5) {'baja' => 3.5, 'media' => 7.5, 'alta' => 9, Array(2) => undefined, 'raspado' => 5} ⓘ
```

Ejemplo 3 → Sobrecribir valores

```
const cartaPapaNoel = new Map();
cartaPapaNoel.set('ropa', 'vestido').set('calzado', 'botas').set('coche','Tesla');
console.log(cartaPapaNoel);
```

```
► Map(3) {'ropa' => 'vestido', 'calzado' => 'botas', 'coche' => 'Tesla'}
```

```
cartaPapaNoel.set('coche','BMW');
console.log(cartaPapaNoel);
```

```
► Map(3) {'ropa' => 'vestido', 'calzado' => 'botas', 'coche' => 'BMW'}
```

8.2.2. MAP.HAS()

A través de este método podremos **verificar si el conjunto contiene o no un elemento concreto**.

```
console.log(notas.has("alta")); //true
console.log(notas.has("notable")); //false
```

8.2.3. MAP.GET()

A través de este método podremos **consultar los valores de los elementos del map**. Si no existe, devolverá "undefined".

```
console.log(notas.get("baja"));
console.log(andalucia.get(18));
```

```
3.5
Jaén
```

8.2.4. MAP.DELETE()

A través de este método eliminaremos elementos del map.

```
comunVal.set(1,"Castellón").set(2,"Valencia").set(3,"Alicante");
comunVal.set(4,"Murcia");
console.log(comunVal);
```

```
► Map(4) {1 => 'Castellón', 2 => 'Valencia', 3 => 'Alicante', 4 => 'Murcia'}
```

```
comunVal.delete(4);
console.log(comunVal);
```

```
► Map(3) {1 => 'Castellón', 2 => 'Valencia', 3 => 'Alicante'}
```

8.2.5. MAP.KEYS() Y MAP.VALUES()

A través de **map.keys()** obtenemos las **claves** del map y a través de **map.values()** obtendremos los **valores** de esas claves.

```
let calificaciones = new Map();
calificaciones.set(0,"Suspenso").set(1,"Suspenso").set(2,"Suspenso").set(3,"Suspenso").set(4,"Suspenso");
calificaciones.set(5,"Suficiente").set(6,"Bien");
calificaciones.set(7,"Notable").set(8,"Notable");
calificaciones.set(9,"Sobresaliente").set(10,"Sobresaliente");
```

```
let claves = calificaciones.keys();
console.log(claves);
```

```
▼ MapIterator {0, 1, 2, 3, 4, ...} ⓘ
  ▼ [[Entries]]
    No properties
    ► [[Prototype]]: Map Iterator
      [[IteratorHasMore]]: false
      [[IteratorIndex]]: 11
      [[IteratorKind]]: "keys"
```

```
let valores = calificaciones.values();
console.log(valores);
```

```
▼ MapIterator {'Suspenso', 'Suspenso', 'Suspenso', 'Suspenso', 'Suspenso', ...} ⓘ
  ▼ [[Entries]]
    No properties
    ► [[Prototype]]: Map Iterator
      [[IteratorHasMore]]: false
      [[IteratorIndex]]: 11
      [[IteratorKind]]: "values"
```


8.3. Propiedad size

Nos indicará cuantos pares de valores tiene el map.

```
console.log(notas.size); //5
```

8.4. Convertir map en array

Utilizaremos el Spread o Rest Operator para convertir un map en Array

```
let calificacionesArray = [...calificaciones];
console.log(calificacionesArray);
console.log(calificacionesArray.length);
console.log(typeof calificacionesArray); //object
console.log(calificacionesArray instanceof Array); //true
```

```
(11) [Array(2), Array(2), Array(2),
(2)] i
  ▶ 0: (2) [0, 'Suspendo']
  ▶ 1: (2) [1, 'Suspendo']
  ▶ 2: (2) [2, 'Suspendo']
  ▶ 3: (2) [3, 'Suspendo']
  ▶ 4: (2) [4, 'Suspendo']
  ▶ 5: (2) [5, 'Suficiente']
  ▶ 6: (2) [6, 'Bien']
  ▶ 7: (2) [7, 'Notable']
  ▶ 8: (2) [8, 'Notable']
  ▶ 9: (2) [9, 'Sobresaliente']
  ▶ 10: (2) [10, 'Sobresaliente']
    length: 11
  ▶ [[Prototype]]: Array(0)
11
```

8.5. Recorrer un map

Al igual que los set, podremos recorrer los map con *for... of* o con *forEach*

```
for (let i of notas){
  console.log(i);
}
```

```
▶ (2) ['baja', 3.5]
▶ (2) ['media', 7.5]
▶ (2) ['alta', 9]
▶ (2) [Array(2), undefined]
▶ (2) ['raspado', 5]
```

```
for (let [clave,valor] of calificaciones){
  console.log(`Clave: ${clave} <--> Valor ${valor}`);
}
```

```
Clave: 0 <--> Valor Suspenso
Clave: 1 <--> Valor Suspenso
Clave: 2 <--> Valor Suspenso
Clave: 3 <--> Valor Suspenso
Clave: 4 <--> Valor Suspenso
Clave: 5 <--> Valor Suficiente
Clave: 6 <--> Valor Bien
Clave: 7 <--> Valor Notable
Clave: 8 <--> Valor Notable
Clave: 9 <--> Valor Sobresaliente
Clave: 10 <--> Valor Sobresaliente
```

9. ANEXO: PROTOTYPE

Hasta ES6, para implementar lo que en otros lenguajes se conoce como **herencia**, JavaScript utilizaba los **prototipos**.

En otros lenguajes, cada objeto pertenece a una clase. Para definir un objeto, primero creamos una clase. Además, una clase puede ser heredera de otra clase, disponiendo así de las propiedades y métodos definidos en la clase de la que hereda.

JavaScript se basaba en que los objetos procedentes del mismo tipo de función constructora tienen un mismo prototipo con el que enlazan. Es decir, todos los objetos tienen un prototipo.

El enlace es dinámico, es decir, el prototipo se puede modificar y los objetos que enlazan con el prototipo estarán actualizados.

El prototipo de un objeto es la parte común de los objetos del mismo tipo (propiedades y métodos)

Ahora los proyectos ya se desarrollan utilizando clases, pero debemos conocer el uso de los prototipos.

Hemos visto dos formas de crear objetos: object literal y object constructor.

El primer método es muy común pero muy estático y el segundo le aporta dinamismo a la creación de objetos del mismo tipo.

Ejemplo 1 → Object literal

```
const cliente = {
  nombre: "Natalia Escrivá Núñez",
  importePte: 325
}
console.log(cliente);
```

```
▼ {nombre: 'Natalia Escrivá Núñez', importePte: 325} ⓘ
  importePte: 325
  nombre: "Natalia Escrivá Núñez"
  ▼ [[Prototype]]: Object
    ▶ constructor: f Object()
    ▶ hasOwnProperty: f hasOwnProperty()
    ▶ isPrototypeOf: f isPrototypeOf()
    ▶ propertyIsEnumerable: f propertyIsEnumerable()
    ▶ toLocaleString: f toLocaleString()
    ▶ toString: f toString()
    ▶ valueOf: f valueOf()
    ▶ __defineGetter__: f __defineGetter__()
    ▶ __defineSetter__: f __defineSetter__()
    ▶ __lookupGetter__: f __lookupGetter__()
    ▶ __lookupSetter__: f __lookupSetter__()
    ▶ __proto__: (...)
    ▶ get __proto__: f __proto__()
    ▶ set __proto__: f __proto__()
```

Ejemplo 2 → Object constructor

```
▼ Cliente {nombre: 'Alberto Peña Cano', importePte: 0} ⓘ
  importePte: 0
  nombre: "Alberto Peña Cano"
  ▼ [[Prototype]]: Object
    ▶ constructor: f Cliente(nombre, importePte)
    ▼ [[Prototype]]: Object
      ▶ constructor: f Object()
      ▶ hasOwnProperty: f hasOwnProperty()
      ▶ isPrototypeOf: f isPrototypeOf()
      ▶ propertyIsEnumerable: f propertyIsEnumerable()
      ▶ toLocaleString: f toLocaleString()
      ▶ toString: f toString()
      ▶ valueOf: f valueOf()
      ▶ __defineGetter__: f __defineGetter__()
      ▶ __defineSetter__: f __defineSetter__()
      ▶ __lookupGetter__: f __lookupGetter__()
      ▶ __lookupSetter__: f __lookupSetter__()
      ▶ __proto__: (...)
      ▶ get __proto__: f __proto__()
      ▶ set __proto__: f __proto__()
```

```
function Cliente(nombre, importePte) {
  this.nombre = nombre;
  this.importePte = importePte;
}

const alberto = new Cliente("Alberto Peña Cano", 0);
console.log(alberto);
```

Podemos ver que la única diferencia entre los dos objetos es la función constructora.

Con *prototype* lo que hacemos es poder añadir propiedades y métodos a los distintos objetos sin tener que ir modificando código según tengamos la necesidad (pensemos en un proyecto grande con muchos empleados manteniéndolo y con muchas funciones o métodos).

9.1. Crear un prototype

Utilizaremos el **método prototype** de los objetos y **crearemos una función** tradicional (no una arrow function). Esto es porque las funciones van a buscar en el objeto y las arrow function en la ventana global.

Vamos a simular que somos un banco.

Creamos el objeto a través del **object constructor**:

```
function Cliente(nombre, saldo) {
  this.nombre = nombre;
  this.saldo = saldo;
}
const juan = new Cliente("Juan Montero Salas", 10000);
console.log(juan);
```

Ejemplo 1 → Función prototype

```
Cliente.prototype.tipoCliente = function(){
  let tipo;
  if (this.saldo >= 50000){
    tipo = "Platinum";
  }else if (this.saldo >= 10000){
    tipo = "Gold";
  }else{
    tipo = "Normal"
  }
  return tipo;
}
console.log(juan.tipoCliente());
```

Gold

Ejemplo 2 → Función prototype que llama a otro prototype. Comparativa con función flecha. NO usaremos arrow function!!

```
Cliente.prototype.infoCte = function() {
  return(`Nombre: ${this.nombre}, saldo: ${this.saldo},
  tipo cliente: ${this.tipoCliente()}`)
}
```

```
Cliente.prototype.infoCte2 = () => {
  return(`Nombre: ${juan.nombre}, saldo: ${juan.saldo},
  tipo cliente: ${juan.tipoCliente()}`)
}
```

```
console.log(juan.infoCte());
console.log(juan.infoCte2());
```

```
Nombre: Juan Montero Salas, saldo: 10000,
tipo cliente: Gold
Nombre: Juan Montero Salas, saldo: 10000,
tipo cliente: Gold
```

Ejemplo 3 → Función prototype con parámetros

```
Cliente.prototype.retiraSaldo = function(importe){
  this.saldo -= importe;
}
console.log(juan);
juan.retiraSaldo(200);
console.log(juan);
```

```
► Cliente {nombre: 'Juan Montero Salas', saldo: 10000}
► Cliente {nombre: 'Juan Montero Salas', saldo: 9800}
```

El uso de prototype nos permite agrupar de forma dinámica las propiedades y los métodos y expandir los objetos.

Ejemplo → Prototype de Cliente.

```
▼ Cliente {nombre: 'Juan Montero Salas', saldo: 9800} ⓘ
  nombre: "Juan Montero Salas"
  saldo: 9800
  [[Prototype]]: Object
    ► infoCte: f ()
    ► infoCte2: () => {...}
    ► retiraSaldo: f (importe)
    ► tipoCliente: f ()
    ► constructor: f Cliente(nombre, saldo)
    ► [[Prototype]]: Object
```

También podemos crear propiedades a través de prototype.

```
Cliente.prototype.activo;
juan.activo = true;
console.log(juan);
```

```
▼ Cliente {nombre: 'Juan Montero Salas', saldo: 9800, activo: true} ⓘ
  activo: true
  nombre: "Juan Montero Salas"
  saldo: 9800
  ► [[Prototype]]: Object
```

9.2. Heredar un prototype: call

Vamos a poder **simular la herencia de las clases** con los prototipos.

Creamos otro objeto que comparte algunas propiedades del objeto creado anteriormente.

Para ello usaremos el método **call**.

```
function Empresa(nombre, saldo, categoria){
  Cliente.call(this, nombre, saldo);
  this.categoria = categoria;
}

const empresa1 = new Empresa("ACADEMIA OPOS", 15000, "docencia");
console.log(empresa1);
```

En este momento lo que hemos heredado son las propiedades.

```
▼ Empresa {nombre: 'ACADEMIA OPOS', saldo: 15000, categoria: 'docencia'}
  categoria: "docencia"
  nombre: "ACADEMIA OPOS"
  saldo: 15000
  ▼ [[Prototype]]: Object
    ► constructor: f Empresa(nombre, saldo, categoria)
    ► [[Prototype]]: Object
```

Si queremos heredar también las funciones, debemos asignarle el prototype del objeto.

```
Empresa.prototype = Object.create(Cliente.prototype);
```

```
▼ Empresa {nombre: 'ACADEMIA OPOS', saldo: 15000, categoria: 'docencia'} ⓘ
  categoria: "docencia"
  nombre: "ACADEMIA OPOS"
  saldo: 15000
  ▼ [[Prototype]]: Cliente
    ▼ [[Prototype]]: Object
      ► infoCte: f ()
      ► infoCte2: () => {...}
      ► retiraSaldo: f (importe)
      ► tipoCliente: f ()
      ► constructor: f Cliente(nombre, saldo)
      ► [[Prototype]]: Object
```

10. ANEXO: REGEXP OBJECT

Las expresiones nos van a servir en todos los lenguajes para **establecer un patrón** que permita poner condiciones avanzadas en los textos, de modo que podamos validar los que encajen con ese patrón.

Las expresiones regulares serán muy útiles para: validación de errores, búsqueda de textos con reglas complejas y modificación avanzada de textos.

Como ejemplo los números de identificación personal en España, donde se suele exigir el NIF (Número de Identificación Fiscal): Este número está formado por 8 números y una letra cuando resulta ser el DNI (Documento Nacional de Identidad), pero puede haber una letra en la primera cifra cuando, por ejemplo, es un NIE (Número de Identificación de Extranjero), y esa letra solo puede ser K, L, X, Y o Z.

Validar todas esas posibilidades sin contar con la letra final que debe cumplir una regla matemática, es muy complejo.

Una sola regla como expresión regular puede establecer el patrón que cumple el NIF y así facilitar su validación.

Las expresiones regulares en JavaScript son **objetos** de tipo **RegExp**.

10.1. Declaración y sintaxis

Las expresiones regulares se pueden crear indicando las mismas **entre dos slash** o barras inclinadas. Entre ambas, colocaremos la expresión regular y **tras las barras se pueden indicar modificadores**.

También podremos utilizar la notación más formal con el constructor integrado.

```
const expr1 = /javascript/g;  
const expr2 = new RegExp('javascript','g');
```

La respuesta por defecto cuando tratamos las expresiones regulares es que el intérprete pare cuando exista la coincidencia.

10.1.1. INTRODUCCIÓN A LOS ELEMENTOS DE LAS REGEXP

Para interpretar las expresiones regulares utilizaremos **símbolos**. Además, nos serviremos de algunos **métodos** para poder validarlas y/o trabajar con ellas.

Nombramos también los delimitadores o **modificadores o flags** que nos van a permitir trabajar nuestras expresiones según las necesidades:

- ✓ i → Significa *case-insensitive*, no distinguirá entre mayúsculas y minúsculas
- ✓ g → Significa *global*, sin este flag, se parará la coincidencia en la primera que encuentre.
- ✓ m → Significa *multilinea*, buscará cuando hay líneas de espacios entre texto. Si no está, la búsqueda parará cuando haya un “\n” en un String.

Estos modificadores se colocarán detrás de la segunda barra de la expresión regular o como segundo argumento (entre comillas) si se utiliza el constructor.

10.2. Métodos de RegExp

Podemos utilizar distintos métodos para trabajar con las expresiones regulares (algunos vinculados al objeto String y otros a RegExp).

10.2.1. STRING.SEARCH()

Busca en un string un valor y **devuelve su posición** si lo encuentra. Si no lo encuentra, devuelve -1.

```
let string = "Estamos aprendiendo JavaScript, las RegExp en JavaScript";
let regexp1 = /JavaScript/g;

console.log(string.search(regexp1)); //20
```

10.2.2. STRING.REPLACE()

Busca las coincidencias y cambia el valor por el nuevo patrón.

```
let string = "Estamos aprendiendo JavaScript, las RegExp en JavaScript";
let regexp1 = /JavaScript/g;

console.log(string.replace(regexp1,"JS"));

Estamos aprendiendo JS, las RegExp en JS
```


10.2.3. STRING.TEST()

Busca si el patrón coincide con alguna sección del String. Devuelve un **booleano**. La sintaxis es:

```
string = "Estamos aprendiendo JavaScript, las RegExp en JavaScript";
regExp1 = /javascript/i;
console.log(regExp1.test(string));
```

true

10.2.4. STRING.EXEC()

Busca si el patrón coincide con alguna sección el String. Devuelve un **objeto**.

```
string = "Estamos aprendiendo JavaScript, las RegExp en JavaScript";
console.log(regExp1.exec(string));
```

```
02-app.js:22
[ 'JavaScript', index: 20, input: 'Estamos aprendiendo JavaScript, las RegExp
p en JavaScript', groups: undefined]
  0: "JavaScript"
  groups: undefined
  index: 20
  input: "Estamos aprendiendo JavaScript, las RegExp en JavaScript"
  length: 1
▶ [[Prototype]]: Array(0)
```

10.2.5. STRING.MATCH()

Las coincidencias de la búsqueda las guarda en un array.

```
string = "Estamos aprendiendo JavaScript, las RegExp en JavaScript";
regExp1 = /a/g;
console.log(string.match(regExp1));
```

```
▶ (7) ['a', 'a', 'a', 'a', 'a', 'a', 'a']
```

10.3. Elementos de las RegExp

Además de los **flags** y **métodos**, debemos tener en cuenta otros **elementos** que nos facilitarán la construcción de las RegExp.

En la página <https://regex101.com/> se puede practicar y ver los resultados de la composición de expresiones regulares.

10.3.1. TIPOS DE CARACTERES

Servirán para ver la coincidencia con un tipo de carácter, ya sea un espacio, un dígito, una letra o cualquier otro.

El **punto** → sustituye a **cualquiera** de ellos.

```

: / .
TEST STRING
tenis•Tenis•T9nis•T\ni8s•Tonic•T%nis•T•nis•_+*=?
string = "45 gatos caminan en 7 calles";
console.log(string.match(/./g));
▶ (28) ['4', '5', ' ', 'g', 'a', 't', 'o', 's', ' ', 'c', 'a', 'm', 'i', 'n', 'a', 'n', ' ', 'e', 'n', ' ', '7', ' ', 'c', 'a', 'l', 'l', 'e', 's']

```

\w → busca todo lo que sea una **letra**, **número** y **guión bajo** en el string.

```

: / \w
TEST STRING
tenis•Tenis•T9nis•T•nis•Tonic•T%nis•T•nis•_+*=?
string = "45 gatos caminan en 7 calles";
console.log(string.match(/\w/g));
▶ (23) ['4', '5', 'g', 'a', 't', 'o', 's', 'c', 'a', 'm', 'i', 'n', 'a', 'n', 'e', 'n', '7', 'c', 'a', 'l', 'l', 'e', 's']

```

\W → busca todo lo que **NO** sea una **letra**, **número** y **guión bajo** en el string.

```

: / \W
TEST STRING
tenis•Tenis•T9nis•T•nis•Tonic•T%nis•T•nis•_+*=?

: / T\Wnis
TEST STRING
tenis•Tenis•T9nis•T•nis•Tonic•T%nis•T•nis•_+*=?
string = "45 gatos caminan en 7 calles";
console.log(string.match(/T\W/g));
▶ (5) [' ', ' ', ' ', ' ', ' ', ' ']

```

`\s` → busca los **espacios** en el string.

```

: / \s
TEST STRING
tenis•Tennis•T9nis•T•nis•Tonic•T%nis•T••nis•_+*=?

```

```

: / nis\s
TEST STRING
tenis•Tennis•T9nis•T•nis•Tonic•T%nis•T••nis•_+*=?

```

```
string = "45 gatos caminan en 7 calles";
```

```
console.log(string.match(/\s/g));
```

```
▶ (5) [' ', ' ', ' ', ' ', ' ']
```

`\S` → busca todo lo que **NO** sean **espacios** en el string.

```

: / \S
TEST STRING
tenis•Tennis•T9nis•T•nis•Tonic•T%nis•T••nis•_+*=?

```

```

: / nis\S
TEST STRING
tenis•Tennis•T9nis•T•nis•Tonic•T%nis•T••nis•_+*=?

```

```

: / \Snis
TEST STRING
tenis•Tennis•T9nis•T•nis•Tonic•T%nis•T••nis•_+*=?

```

```
string = "45 gatos caminan en 7 calles";
```

```
console.log(string.match(/\S./g));
```

```
▶ (23) ['4', '5', 'g', 'a', 't', 'o', 's', 'c', 'a', 'm', 'i', 'n', 'a', 'n', 'e', 'n', '7', 'c', 'a', 'l', 'l', 'e', 's']
```

\d → busca cualquier número

```

: / \d
TEST STRING
tenis•Tennis•T99nis•T•ni8s•Tonic•T%nis•T••nis•_+*=?

: / T\dnis
TEST STRING
tenis•Tennis•T9nis•T•ni8s•Tonic•T%nis•T••nis•_+*=?

string = "45 gatos caminan en 7 calles";
console.log(string.match(/\d/g));
▶ (3) ['4', '5', '7']

```

\D → busca todo lo que **NO** sea un número.

```

: / T\D
TEST STRING
tenis•Tennis•T9nis•T•\ni8s•Tonic•T%nis•T••nis•_+*=?

: / T\Dnis
TEST STRING
tenis•Tennis•T9nis•T•\ni8s•Tonic•T%nis•T••nis•_+*=?

string = "45 gatos caminan en 7 calles";
console.log(string.match(/\D/g));
▶ (25) [' ', 'g', 'a', 't', 'o', 's', ' ', 'c', 'a', 'm', 'i', 'n', 'a', 'n', ' ', 'e', 'n', ' ', ' ', 'c', 'a', 'l', 'l', 'e', 's']

```

10.3.2. CUANTIFICADORES

Si queremos que un patrón se repita varias veces o que sea opcional dependiendo la situación, utilizaremos los **cuantificadores**.

{x,y} → delimita el rango de veces que se repite el patrón.

Ejemplo 1 → Selecciona entre 2 y 3 caracteres [a-zA-z0-9_] seguidos de espacio/s

```

REGULAR EXPRESSION 7 matches (0.0ms)
: / \w{2,3}\s / gm

TEST STRING
Estamos aprendiendo JS, sacaré muy buena nota en JS

```

Ejemplo 2 → Selecciona 5 caracteres seguidos de espacio

```

REGULAR EXPRESSION 3 matches (0.1ms)
: / \w{5}\s / gm

TEST STRING
Estamos aprendiendo JS, sacaré muy buena nota en JS

```

Ejemplo 3 → Selecciona mínimo 8 caracteres.

```

REGULAR EXPRESSION 1 match (0.0ms)
: / \w{8,} / gm

TEST STRING
Estamos aprendiendo JS, sacaré muy buena nota en JS

```

? → La expresión de su izquierda se repetirá 0 ó 1 vez.

```

REGULAR EXPRESSION 6 matches (0.0ms)
: / ga?l?t?r? / gm

TEST STRING
gatos gaaaaaaalgos
garrotes
galapagos

```

***** → La expresión de su izquierda se repetirá 0 ó más veces

```

REGULAR EXPRESSION 5 matches (0.1ms)
: / ga*t* / gm

TEST STRING
gattiklkljlkhgaaaaattttnnntlkhkhkhhhtos
gaaaaaaahkjkhhkhkhkhkhkhkgos
gansos

```

+ → La expresión de su izquierda se repetirá 1 ó más veces

```
REGULAR EXPRESSION 4 matches (0.0ms)
: / ga+ / gm

TEST STRING
gatos gaaaaaaalgnos
garrotes
galapagos
```

10.3.3. LITERALES

Busca la palabra o el conjunto de letras que deseamos buscar hasta los códigos Unicode.

\n → Buscará una nueva línea de texto

```
REGULAR EXPRESSION 3 matches (0.1ms)
: / \n / gm

TEST STRING
Con diez cañones por banda,
viento en popa, a toda vela,
no corta el mar, sino vuela
un velero bergantín.
```

\t → Busca espacios de tabulación

```
REGULAR EXPRESSION 2 matches (0.1ms)
: / \t / gm

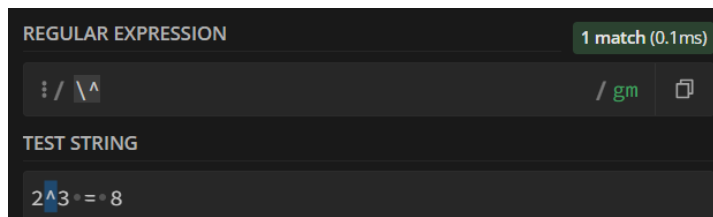
TEST STRING
Busca espacios de tabulación
```

\uXXXX → Busca el carácter correspondiente al código Unicode facilitado

```
REGULAR EXPRESSION 1 match (0.0ms)
: / \u00f1 / gm

TEST STRING
100 años han pasado y no existe calma
```

\\$.()^ → Buscará los caracteres especiales de las RegExp de forma literal

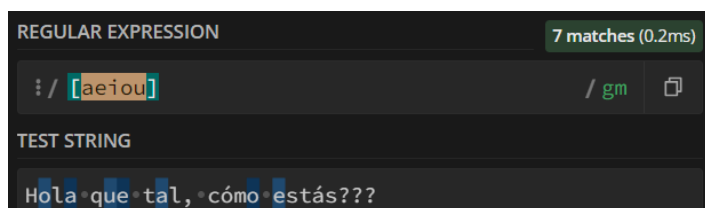


10.3.4. CONJUNTO DE CARACTERES

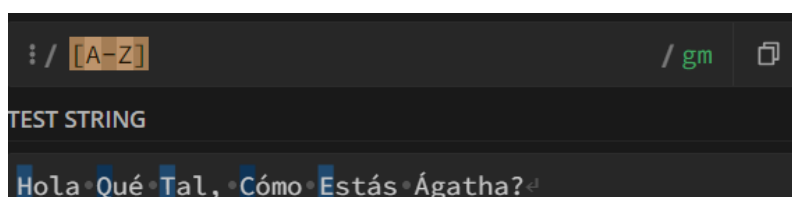
Es una forma de agrupar caracteres. Busca todas las coincidencias dentro de los corchetes.

Tendremos en cuenta que los caracteres (`$`, `*`, `+`, `?`) perderán su significado y se convertirán en literales.

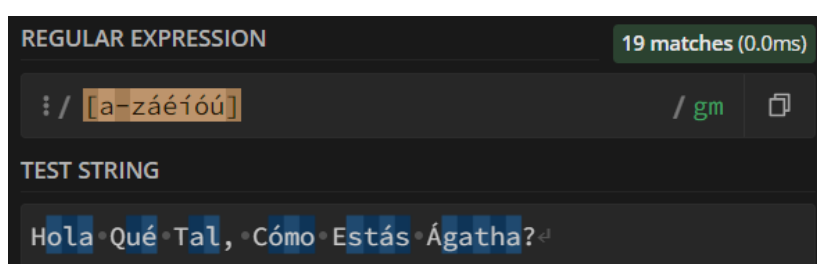
[aeiou] → Busca las vocales minúsculas no acentuadas



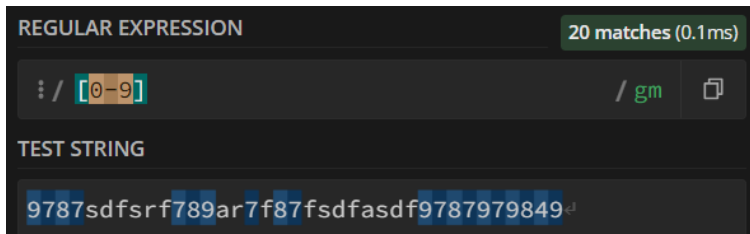
[A-Z] → Busca mayúsculas, sin acentuar



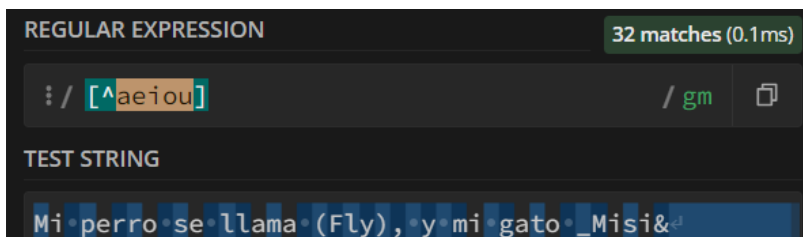
[a-z] → Busca minúsculas, sin acentuar. En el ejemplo se añaden las vocales acentuadas



[0-9] → Buscará los números (igual que `\d`)



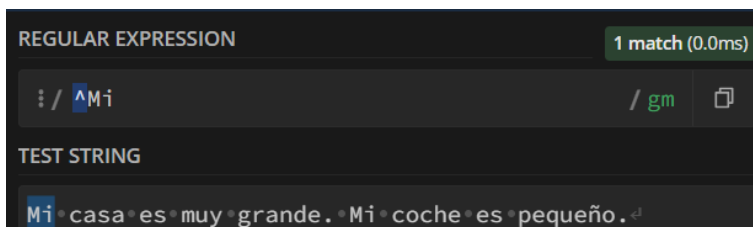
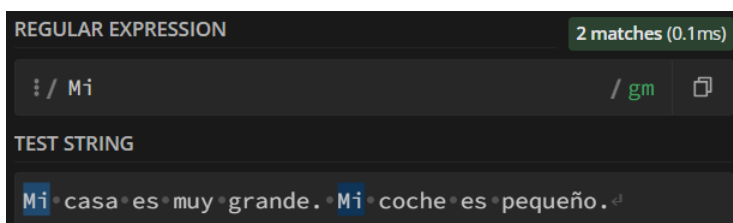
[^aeiou] → Buscará todos los elementos que NO estén entre los corchetes. En este caso todo menos las vocales sin acentuar.



10.3.5. LIMITADORES

El objetivo es, como su nombre indica, limitar la búsqueda de caracteres según su posición en el String.

^ → Buscará solo al inicio del String.



\$ → Buscará solo al final del String.

```
REGULAR EXPRESSION 2 matches (0.0ms)
:/ JS / gm
TEST STRING
Estamos aprendiendo JS, sacaré muy buena nota en JS
```

```
REGULAR EXPRESSION 1 match (0.0ms)
:/ JS$ / gm
TEST STRING
Estamos aprendiendo JS, sacaré muy buena nota en JS
```

\b → Dependiendo de su posición, buscará que no haya nada (\w) delante o detrás.

```
REGULAR EXPRESSION 4 matches (0.1ms)
:/ ras / gm
TEST STRING
Coge esas barras. Cuidado con la rasante!
Hoy comemos carne a la brasa
Estás yendo a ras de la carretera
```

Ejemplo 1 → que no haya nada detrás.

```
REGULAR EXPRESSION 2 matches (0.0ms)
:/ ras\b / gm
TEST STRING
Coge esas barras. Cuidado con la rasante!
Hoy comemos carne a la brasa
Estás yendo a ras de la carretera
```

Ejemplo 2 → que no haya nada delante.

```
REGULAR EXPRESSION 2 matches (0.0ms)
:/ \bras / gm
TEST STRING
Coge esas barras. Cuidado con la rasante!
Hoy comemos carne a la brasa
Estás yendo a ras de la carretera
```

Ejemplo 3 → que no haya nada delante ni detrás.

```
REGULAR EXPRESSION 1 match (0.1ms)
:/ \bras\b / gm

TEST STRING
Coge esas barras. Cuidado con la rasante!
Hoy comemos carne a la brasa
Estás yendo a ras de la carretera
```

\B → Actúa igual que el anterior pero buscará (\w, \W), delante, detrás o ambas.

10.3.6. AGRUPADORES Y ALTERNACIÓN

Podemos necesitar trabajar las expresiones según conjuntos o agrupaciones, de forma que utilicemos una lógica separada de la RegExp general y así aplicar patrones más concretos.

```
REGULAR EXPRESSION 2 matches (0.3ms)
:/ (Natalia){2,3} / gm

TEST STRING
NataliaNataliaNatalia
NataliaNataliaNataliaNatalia

EXPLANATION
MATCH INFORMATION
Match 1 0-21 NataliaNataliaNatalia
Group 1 14-21 Natalia
Match 2 22-43 NataliaNataliaNatalia
Group 1 36-43 Natalia
```

opcion1 | opcion2 → Busca una opción u otra.

```
REGULAR EXPRESSION 4 matches (0.0ms)
:/ norte|sur|este|oeste / gmi

TEST STRING
Vivo al norte de España, no me gusta el sur.
Galicia, la comunidad al NordOeste del país
Comunidad Valenciana, al este.
```

x(?=y) → **Lookahead positivo**. X hará match siempre que lo que hay entre paréntesis exista (delante de x)

```
REGULAR EXPRESSION 1 match (0.1ms)
: / nota*(?=(baja|media|alta)) / gmi

TEST STRING
La nota más alta ha sido un 9.75
Has sacado una nota alta, enhorabuena!!
Esta baja nota no se va a consentir más.
```

```
REGULAR EXPRESSION 1 match (0.1ms)
: / nota*(?=alta) / gm

TEST STRING
La nota más alta ha sido un 9.75
Has sacado una nota alta, enhorabuena!!
Esta baja nota no se va a consentir más.
```

x(?!y) → **Lookahead negativo**. X hará match siempre que lo que hay entre paréntesis NO exista (delante de x)

```
REGULAR EXPRESSION 2 matches (0.0ms)
: / nota*(?!(baja|media|alta)) / gmi

TEST STRING
La nota más alta ha sido un 9.75
Has sacado una nota alta, enhorabuena!!
Esta baja nota no se va a consentir más.
```

(?<=y)x → **Lookbehind positivo**. X hará match siempre que lo que hay entre paréntesis exista (detrás de x)

```
REGULAR EXPRESSION 1 match (0.1ms)
: / (?<=(baja|media|alta))*nota / gmi

TEST STRING
La nota más alta ha sido un 9.75
Has sacado una nota alta, enhorabuena!!
Esta baja nota no se va a consentir más.
```

```

REGULAR EXPRESSION 1 match (0.1ms)
:/ (?<=baja)nota /gm

TEST STRING
La nota más alta ha sido un 9.75
Has sacado una nota alta, enhorabuena!
Esta baja nota no se va a consentir más

```

(?<!y)x → Lookbehind negativo. X hará match siempre que lo que hay entre paréntesis NO exista (detrás de x)

```

REGULAR EXPRESSION 2 matches (0.2ms)
:/ (?<!(baja|media|alta))nota /gmi

TEST STRING
La nota más alta ha sido un 9.75
Has sacado una nota alta, enhorabuena!!
Esta baja nota no se va a consentir más.

```

(?:) → Considerado Non Capturing Group. Busca lo que se encuentra dentro del paréntesis

```

REGULAR EXPRESSION 2 matches (0.1ms)
:/ (?:nota)\w+ /gmi

TEST STRING
La notaza más alta ha sido un 9.75
Has sacado una nota alta: anota
Has sacado un notable.

```

JavaScript Array Reference

Name	Description
[]	Creates a new Array
new Array()	Creates a new Array
at()	Returns an indexed element of an array
concat()	Joins arrays and returns an array with the joined arrays
constructor	Returns the function that created the Array prototype
copyWithin()	Copies array elements within the array, to and from specified positions
entries()	Returns a key/value pair Array Iteration Object
every()	Checks if every element in an array pass a test
fill()	Fill the elements in an array with a static value
filter()	Creates a new array with every element in an array that pass a test
find()	Returns the value of the first element in an array that pass a test
findIndex()	Returns the index of the first element in an array that pass a test
findLast()	Returns the value of the last element in an array that pass a test
findLastIndex()	Returns the index of the last element in an array that pass a test
flat()	Concatenates sub-array elements
flatMap()	Maps all array elements and creates a new flat array
forEach()	Calls a function for each array element
from()	Creates an array from an object
includes()	Check if an array contains the specified element
indexOf()	Search the array for an element and returns its position
isArray()	Checks whether an object is an array
join()	Joins all elements of an array into a string
keys()	Returns a Array Iteration Object, containing the keys of the original array
lastIndexOf()	Search the array for an element, starting at the end, and returns its position
length	Sets or returns the number of elements in an array
map()	Creates a new array with the result of calling a function for each array element
of()	Creates an array from a number of arguments
pop()	Removes the last element of an array, and returns that element
prototype	Allows you to add properties and methods to an Array object
push()	Adds new elements to the end of an array, and returns the new length
reduce()	Reduce the values of an array to a single value (going left-to-right)
reduceRight()	Reduce the values of an array to a single value (going right-to-left)
reverse()	Reverses the order of the elements in an array
shift()	Removes the first element of an array, and returns that element
slice()	Selects a part of an array, and returns the new array
some()	Checks if any of the elements in an array pass a test
sort()	Sorts the elements of an array
splice()	Adds or Removes array elements
toReversed()	Reverses the order of array elements (to a new array)
toSorted()	Sorts the elements of an array (to a new array)
toSpliced()	Adds or Removes array elements (to a new array)
toString()	Converts an array to a string, and returns the result
unshift()	Adds new elements to the beginning of an array, and returns the new length
valueOf()	Returns the primitive value of an array
with()	Returns a new array with updated elements