Desarrollo Web en Entorno Cliente

UT1. Javascript

Actualizado septiembre 2022

ÍNDICE DE CONTENIDO

1. INTRODUCCIÓN

"Programar es el arte de decirle a otro humano lo que quieres que el ordenador haga" – Donald Knuth.

Javascript es un lenguaje de programación que inicialmente nació como un lenguaje que permitía ejecutar código en nuestro navegador (cliente), ampliando la funcionalidad de nuestros sitios web.

Una de las versiones más extendidas de Javascript moderno, es la llamada por muchos **Javascript ES6** (ECMAScript 6), también llamado ECMAScript 2015 o incluso por algunos llamado directamente Javascript 6. Al crear esta pequeña guía nos hemos basado en esta versión. Si ya sabes Javascript pero quieres conocer novedades de Javascript ES6, te recomendamos este curso https://didacticode.com/curso/curso-javascript-es6/

Actualmente esa es una de sus principales funciones, pero dada su popularidad el lenguaje ha sido portado a otros ámbitos, entre los que destaca el popular NodeJS https://nodejs.org/ que permite la ejecución de Javascript como como lenguaje escritorio y lenguaje servidor.

Aunque en este módulo nos centramos en la ejecución de Javascript en el navegador, lo aprendido puede ser utilizado para otras implementaciones de Javascript.

¿Por qué la gente no aprende a programar?: https://www.youtube.com/watch?v=kTX7smcwfeo

¿Qué se puede hacer con JS?: https://www.youtube.com/watch?v=qY2JD78kShQ

2. HOLA MUNDO Y COMENTARIOS

Para añadir JavaScript se usa la etiqueta **SCRIPT**.

Este puede estar en cualquier lugar de la página. El código se ejecuta en el lugar donde se encuentra de forma secuencial a como el navegador lo va encontrando.

```
<script>
  Aquí va todo el código
  // Esto es un comentario en Javascript de una sola línea */
 </script>
```

En Javascript los **comentarios** se pueden hacer con // para una línea y con /* */ para varias líneas.

Asimismo, desde Javascript es posible escribir mensajes a la consola de desarrollo mediante

la orden “console.log(texto)”.

Este ejemplo podría ser un pequeño hola mundo que al ejecutarse se mostrará en la consola de desarrollo. Ejemplo con “console.log” y comentario multilínea.

```
<script>
 console.log ("Hola mundo");
 /* Esto es un comentario en
   Javascript multilínea */
</script>
```

Otra vía para mostrar información al usuario desde una ventana, es el comando “alert(texto)”.

```
<script>
 alert ("Hola mundo");
</script>
```

Hay una forma mucho más práctica y ordenada de usar código Javascript. Se pueden **incluir uno o varios ficheros con código Javascript** en nuestro documento HTML. Se puede incluir tantos como se desee. Esta es la forma más adecuada para trabajar con código Javascript.

```
<script src="./js/parImpar.js"></script>
```

Lo más recomendable es colocarlo antes de la etiqueta de cierre del cuerpo del HTML </body> ya que es posible que el código javascript haga uso de algún elemento HTML.

Recuerda que JavaScript se ejecuta en dos ”vueltas” (este comportamiento es conocido como *hoisting*), en la primera registra las funciones y en la segunda ejecuta, por eso se pueden poner las definiciones después de la llamada.

3. VARIABLES Y TIPOS

3.1. Nomenclatura para un código limpio

Existen una serie de recomendaciones sencillas sobre como nombrar las diferentes partes de nuestro código, para que este sea fácilmente legible y mantenible.

Los **nombres de las variables y constantes** deben ser preferiblemente en inglés y deben ser pronunciables. Esto quiere decir que no deben ser abreviaturas, ni llevar guion bajo o medio, priorizando el estilo CamelCase[1] (en concreto el lowerCamelCase). Por otro lado,

[1] https://medium.com/@alonsus91/convenci%C3%B3n-de-nombres-desde-el-camelcase-hasta-el-kebab-case-787e56d6d023

debemos intentar no ahorrarnos caracteres en los nombres, la idea es que seamos lo más expresivos posible.

Bad:

```
const yyyymmdstr = moment().format("YYYY/MM/DD");
```

Good:

```
const currentDate = moment().format("YYYY/MM/DD");
```

Evitar información técnica en los nombres.

```
//bad
class AbstractUser(){...}

//better
class User(){...}
```

Léxico coherente, usando el mismo vocabulario para hacer referencia al mismo concepto, no debemos usar en algunos lados User, en otro Client y en otro Customer, a no ser que representen claramente conceptos diferentes.

```
//bad
getUserInfo();
getClientData();
getCustomerRecord();

//better
getUser()
```

Según el tipo de dato es recomendable seguir una serie de pautas:

- **Arrays**. Pluralizar el nombre de la variable puede ser una buena idea.

```
//bad
const fruit = ['manzana', 'platano', 'fresa'];
// regular
const fruitList = ['manzana', 'platano', 'fresa'];
// good
const fruits = ['manzana', 'platano', 'fresa'];
// better
const fruitNames = ['manzana', 'platano', 'fresa'];
```

- **Booleanos**. Como sólo pueden tener dos valores, el uso de prefijos como is, has y can pueden ayudar a inferir el tipo de variable.

```
//bad
const open = true;
const write = true;
const fruit = true;

// good
const isOpen = true;
const canWrite = true;
const hasFruit = true;
```

- **Números**. Es interesante escoger palabras que describan estos números, como min, max, total, …

```
//bad
const fruits = 3;

//better
const maxFruits = 5;
const minFruits = 1;
const totalFruits = 3;
```

- **Funciones**. Los nombres de funciones deben representar acciones, por lo que deben construirse usando el verbo que representa la acción seguido de un sustantivo. Deben ser descriptivos y concisos, expresando lo que hace, pero abstrayéndose de su implementación.

```
//bad
createUserIfNotExists()
updateUserIfNotEmpty()
sendEmailIfFieldsValid()

//better
createUser(...)
updateUser(...)
sendEmail()
```

En el caso de las funciones de acceso, modificación o predicado, el nombre debe ser el prefijo *get*, *set* e *is*, respectivamente.

```
getUser()
setUser(...)
isValidUser()
```

- **Get y set**. En el caso de los *getters* y *setters*, sería interesante hacer uso de las palabras clave *get* y *set* cuando estemos accediendo a propiedades de objetos.

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

  get name() {
    return this._name;
  }

  set name(newName) {
    this._name = newName;
  }
}

let person = new Person('Miguel');
console.log(person.name); // Outputs 'Miguel'
```

- **Clases**. Las clases y los objetos deben tener nombres formados por un sustantivo o frases de sustantivo como *User*, *UserProfile*, *Account* o *AddressParser*. Debemos evitar nombres demasiado genéricos como *Data*, *Info*, …

Ver: https://github.com/ryanmcdermott/clean-code-javascript

3.2 Variables y constantes

Las variables son elementos del lenguaje que permiten almacenar distintos valores en cada momento. Se puede almacenar un valor en una variable y consultar este valor posteriormente. También podemos modificar su contenido siempre que queramos.

Para declarar las variables en JavaScript podemos utilizar **var, let o const**. A la hora de usa una u otra hay que tener en cuenta aspectos como el ámbito ("scope") donde se puede usar dicha variable o si se va a modificar o no.

- **var**: permite declarar una variable que sea accesible por todos los lugares de la **función** donde ha sido declarada. Si una variable con var se declara fuera de cualquier función, el ámbito de esta son todas las funciones del código.
- **let**: permite declarar una variable que sea accesible únicamente dentro del **bloque** donde se ha declarado (llamamos bloque al espacio delimitado por { }). Si se declara una variable con let fuera de cualquier bloque, tendrá un ámbito global.
  Ejemplo 1:

```
let greeting = 'hello world!';

function greet(){
        console.log(greeting);
}

greet(); //"Hello world";
```

  Ejemplo 2:

```
{
        let greeting = "Hello world!";
        var lang = "English";
        console.log(greeting); //Hello world!
}

console.log(lang);//"English"
console.log(greeting);//// Uncaught ReferenceError: greeting is not def\
ined
```

- **const**: permite declarar una variable a la que no se le puede reasignar su valor, pero si modificarlo. Es decir, se puede modificar en el caso de un objeto o un array, pero no si se trata de un tipo primitivo. Su ámbito es el mismo que el de **let**, es decir, solo son accesibles en el bloque que se han declarado.

```
const PI=3.1416;
```

```
console.log(PI);
PI=3; // Esto falla
```

Al declarar un array/objeto, realmente lo que ocurre es que la variable almacena la dirección de memoria del objeto/array. Si lo declaramos usando **const**, lo que haremos es que no pueda cambiarse esa dirección de memoria, pero nos permitirá cambiar sus valores.

```
const miArray=[1,2,3]
console.log(miArray[0]); // Muestra el valor 1
miArray[0]=4;
console.log(miArray[0]); // Muestra el valor 4
miArray=[]; // Esto falla
```

- **Variables sin declarar**: Javascript nos permite usar variables no declaradas. Si hacemos esto, será equivalente a declararlas con **var**.

```
function ejemplo(){
    ejemplo=3; // Equivale a declararla fuera de la funcion como var
    if (ejemplo === 3){
     var variable1 = 1;
     let variable2 = 2;
    }
    console.log(variable1);  // variable1 existe en este lugar
    console.log(variable2); // variable2 no existe en este lugar
  }
```

En general, recomendamos encarecidamente usar **let** o **const.** Deberías tener una buena razón para usar **var**, ya que su uso es peligroso ya que podría modificarse una variable desde un lugar que no controles por accidente.

Ámbito estático

El ámbito de las variables en JS tiene un comportamiento de naturaleza estática, esto quiere decir que se determina en tiempo de compilación en lugar de tiempo de ejecución.

```
const number = 10;
function printNumber() {
  console.log(number);
}

function app() {
  const number = 5;
  printNumber();
}

app(); //10
```

3.3 Tipos de datos primitivos y objetos

Los principales tipos de **datos primitivos** que pueden contener variables en Javascript son:

- **Numéricos (tipo "number")**: puede contener cualquier tipo de número real (0.3, 1.7, 2.9) o entero (5, 3, -1).
- **Enteros grandes (tipo "bigint"):** pueden contener enteros con valores superiores a 2^53 -1. Se pueden nomenclar escribiendo una letra "n" al final del entero. No pueden utilizarse con la mayoría de operadores matemáticos de Javascript.
- **Booleanos (tipo "boolean")**: puede contener uno de los siguientes valores: true, false, 1 y 0.
- **Cadenas (tipo "string")**: cualquier combinación de caracteres (letras, números, signos especiales y espacios). Las cadenas se delimitan mediante comillas dobles o simples ("Lolo",'lalO'). Para concatenar cadenas puede usarse el operador "+" o con el uso de ***template strings***:

```
// Concatenación
console.log("Nombre Cliente: " + nombre + " Email: " + email);


// Template Strings - String Literals
console.log(`Nombre Cliente: ${nombre} Email: ${email}`);
```

Dentro de un template string se puede colocar una expresión (con llamadas a funciones, por ejemplo), como un condicional ternario:

```
`${<condicion>?valorSiVerdadero:valorSiFalso}`
```

Puede tener más sentido cuando veamos DOM, para escribir cierto código HTML dependiendo de una condición.

```
$id.innerHTML=`
    <ul>
        <li>User Agent: <b>${ua}</b></li>
        <li>Plataforma:
<b>${isMobile.any()?isMobile.any():isDesktop.any()}</b></li>
        <li>Navegador: <b>${isBrowser.any()}</b></li>
    </ul>`
```

- El tipo de una variable puede comprobarse usando la estructura **typeof** variable===”tipo” (puede usarse incluso para saber si es una function, un object o undefined).

```
let edad=23, nueva_edad, incremento=4;
const nombre="Rosa García";
console.log(typeof incremento===”number”)
nueva_edad=edad+incremento;
console.log(nombre+ " tras "+incremento +" años tendrá "+ nueva_edad);
```

Asimismo, existen otros tipos que Javascript considera primitivos: “undefined” y “symbol”.

Todos los valores que **NO son de un tipo básico** son considerados objetos: arrays, funciones, objetos, etc.

Esta distinción es muy importante porque los valores primitivos y los valores objetos se comportan de distinta forma cuando son asignados y cuando son pasados como parámetro a una función:

- por valor: Cuando asignamos valores primitivos (boolean, null, undefined, number, string y symbol), el valor asignado es una copia del valor que estamos asignando.
- por referencia: Pero cuando asignamos valores NO primitivos o complejos (object, array y function), JavaScript copia “la referencia”, lo que implica que no se copia el valor en sí, si no una referencia a través de la cual accedemos al valor original.

```
//TIPOS PRIMITIVOS: POR VALOR
let a = "hola";
let b = a;
console.log(a);//hola
console.log(b);//hola

//TIPOS NO PRIMITIVOS: POR COPIA
let v1 = ["hola"];
let v2 = v1;

v1.push("chao");

console.log(v2);//['hola','chao']
```

Para evitar este último comportamiento, se puede asignar una copia con el operador

spread … (aunque esto lo veremos más adelante)

```
let v1 = ["hola"];
let v2 = {...v1};

v1.push("chao");

console.log(v1);//['hola','chao']
console.log(v2);//['hola']
```

Más información en https://developer.mozilla.org/es/docs/Web/JavaScript/Data_structures.

3.4 Coerción

Javascript es un lenguaje de tipado blando (es decir, al declarar una variable no se le asigna un tipo), aunque internamente Javascript si maneje tipos de datos.

En determinados momentos, resulta necesario convertir un valor de un tipo a otro. Esto en JS se llama "coerción", y puede ocurrir de forma implícita o podemos forzarlo de forma explícita.

Por ejemplo:

```
let numero = 5;
console.log(numero);
```

En este código, ocurre una coerción **implícita** de número (que es de tipo number) a un tipo string, de modo que puede ser impreso por consola. Podríamos realizar la conversión de forma **explícita** de la siguiente forma:

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

Las coerciones implícitas ocurren muy a menudo en JS, aunque muchas veces no seamos conscientes de ello. Resulta muy importante entender cómo funcionan para poder deducir cuál será el resultado de una comparación.

```
let a = "2", b = 5;
console.log( typeof a + " " + typeof b); // string number
console.log( a + b ); //25
```

En los lenguajes de tipado duro (por ejemplo, Java) se nos prohíbe realizar operaciones entre distintos tipos de datos. Sin embargo, Javascript, no hace eso, ya que permite operar entre distintos tipos, siguiendo una serie de reglas:

- Javascript tiene el operador === y !=== para realizar comparaciones estrictas, pero no posee esos operadores para desigualdades (>, <, >=, <=).
- Si es posible, JS prefiere hacer coerciones a tipo number por encima de otros tipos básicos. Por ejemplo, la expresión ("15" < 100) se resolverá como true porque JS

cambiará “15”, de tipo string, por 15 de tipo number.
    - Ten en cuenta que, si conviertes “15” a string, al compararlo con “100” la expresión se resolvería como false.
- A la hora de hacer coerción a boolean, los siguientes valores se convertirán en false: undefined, null, 0, “”, NaN. El resto de valores se convertirán en true.

Ejemplo coerción a number

```
// <, <=, >, <= también hacen coercion. No existe >== ni <==
let arr = [ "1", "10", "100", "1000" ];
for (let i = 0; i < arr.length && arr[i] < 500; i++) {
    console.log(i);
}//0,1,2
```

Ejemplo donde no se hace coerción

```
var x = "10";
var y = "9";
console.log(x < y);  // true, los dos son String y los compara como cadena
```

Ejemplo de coerción con undefined

```
let altura; // variable no definida
console.log(altura ? true : false); //Al no estar definido, false
```

Al realizar comparaciones, si usas == o != para comparar los datos, Javascript realiza coerción. Si quieres que la comparación no convierta tipos y solo sea cierta si son del mismo tipo, debes usar === o !==. Esta es una buena práctica muy recomendada para que estas conversiones no nos jueguen malas pasadas.

Lo mejor es que seas tú el que controles con qué tipos de datos y qué operaciones vas a realizar.

Para más información https://www.etnassoft.com/2011/04/06/coercion-de-datos-en-javascript/

3.5 Conversiones entre tipos

Javascript no define explícitamente el tipo de datos de sus variables. Según se almacenen, pueden ser cadenas (entre comillas simples o dobles), enteros (sin parte decimal) o decimales (con parte decimal).

Elementos como la función “prompt” para leer de teclado con una ventana emergente del navegador, leen los elementos siempre como cadena. Para estos casos y otros, merece la pena usar funciones de conversión de datos.

```
let num="100"; //Es una cadena
let num2="100.13"; //Es una cadena
let num3=11; // Es un entero
```

```
let n=parseInt(num); // Almacena un entero. Si hubiera habido parte decimal la truncará
let n2=parseFloat(num); // Almacena un decimal
let n3=num3.toString(); // Almacena una cadena
```

3.6 Modo estricto "use strict"

Javascript ES6 incorpora el llamado "modo estricto". Si en algún lugar del código se indica la sentencia "use strict" indica que ese código se ejecutará en modo estricto:

- Escribir "use strict" fuera de cualquier función afecta a todo el código.
- Escribir "use strict" dentro de una función afecta a esa función.

"use strict" es ignorado por versiones anteriores de Javascript, al tomarlo como una simple declaración de cadena.

Las principales características de "use strict" son:

- No permite usar variables/objetos no declarados (los objetos son variables).
- No permite eliminar (usando delete) variables/objetos/funciones.
- No permite nombres duplicados de parámetros en funciones.
- No permite escribir en propiedades de objetos definidas como solo lectura.
- Evita que determinadas palabras reservadas sean usadas como variables (eval, arguments, this, etc…)

4. ENTRADA Y SALIDA EN NAVEGADORES

Estos métodos forman parte del objeto Window. Para hacer llamada a estos métodos no hace falta nombrar explícitamente Window (el navegador ya se encarga de ello).

4.1 alert

El método alert() permite mostrar al usuario información literal o el contenido de variables en una ventana independiente. La ventana contendrá la información a mostrar y el botón Aceptar.

```
alert("Hola mundo");
```

4.2 console.log

El método console.log[2] permite mostrar información en la consola de desarrollo. En versiones de Javascript de escritorio tipo NodeJS, permite mostrar texto "por pantalla".

```
console.log("Otro hola mundo");
```

[2] Si escribes clg en VSCode te lo autocompletará a console.log(object);

```
console.table(array);//esta variación es muy útil para visualizar la
variable en forma de tabla
```

También se puede usar para saber cuánto tarde en ejecutarse un código:

```
console.time('Cuanto tiempo tarda mi código');
const arreglo = new Array(1000000);
for (let i = 0; i < arreglo.length; i++) {
  arreglo[i] = i;
}
console.timeEnd('Cuanto tiempo tarda mi código'); //el texto de .time y
.timeEdnd debe ser el mismo
```

4.3 confirm

A través del método confirm() se activa un cuadro de diálogo que contiene los botones Aceptar y Cancelar. Cuando el usuario pulsa el botón Aceptar, este método devuelve el valor true; Cancelar devuelve el valor false. Con ayuda de este método el usuario puede decidir sobre preguntas concretas e influir de ese modo directamente en la página.

```
let respuesta;
respuesta=confirm ("¿Desea cancelar la suscripción?");
alert("Usted ha contestado que "+respuesta);
```

4.4 prompt

El método prompt() abre un cuadro de diálogo en pantalla en el que se pide al usuario que introduzca algún dato. Si se pulsa el botón Cancelar, el valor de devolución es false/null. Pulsando en Aceptar se obtiene la cadena de caracteres introducida se guarda para su posterior procesamiento.

```
let provincia;
provincia=prompt("Introduzca la provincia ","Valencia");
alert("Usted ha introducido la siguiente información "+provincia);
console.log("Operación realizada con éxito");
```

5. OPERADORES

Combinando variables y valores, se pueden formular expresiones más complejas. Las expresiones son una parte clave en la creación de programas. Para formular expresiones utilizamos los llamados operadores. Pasamos a comentar los principales operadores de

Javascript.

5.1 Operadores de asignación

Los operadores de asignación se utilizan para asignar valores a las variables. Algunos de ellos, además de asignar el valor, también incluyen operaciones (Ejemplo += asigna y suma).

Operador	Descripción
=	Asigna a la variable de la parte izquierda el valor de la parte derecha.
+=	Suma los operandos izquierdo y derecho y asigna el resultado al operando izquierdo.
-=	Resta el operando derecho del operando izquierdo y asigna el resultado al operando izquierdo.
*=	Multiplica ambos operandos y asigna el resultado al operando izquierdo.
/=	Divide ambos operandos y asigna el resultado al operando izquierdo.

```
let num1=3;
let num2=5;
num2+=num1;
num2-=num1;
num2*=num1;
num2/=num1;
num2%=num1;
```

5.2 Operadores aritméticos

Los operadores aritméticos se utilizan para realizar cálculos aritméticos.

Operador	Descripción
+	Suma.
-	Resta.
*	Multiplicación.
/	División.
%	Calcula el resto de una división entera.

Además de estos operadores, también existen operadores aritméticos unitarios: incremento (++), disminución (--) y la negación unitaria (-).

Los operadores de incremento y disminución pueden estar tanto delante como detrás de una variable, con distintos matices en su ejecución. Estos operadores aumentan o disminuyen en 1, respectivamente, el valor de una variable.

Operador	Descripción (Suponiendo x=5)
y = ++x	Primero el incremento y después la asignación. x=6, y=6.
y = x++	Primero la asignación y después el incremento. x=6, y=5.
y = --x	Primero el decremento y después la asignación. x=4, y=4.
y = x--	Primero la asignación y después el decremento x=4, y=5.
y =-x	Se asigna a la variable "y" el valor negativo de "x", pero el valor de la variable "x" no varía. x=5, y= -5.

```
let num1=5, num2=8,resultado1, resultado2;
resultado1=((num1+num2)*200)/100;
resultado2=resultado1%3;
resultado1=++num1;
resultado2=num2++;
resultado1=--num1;
resultado2=num2--;
resultado1=-resultado2;
```

Acostúmbrate a utilizar paréntesis para **controlar el orden de las operaciones**.

5.3 Operadores de comparación

Operadores utilizados para comparar dos valores entre sí. Como valor de retorno se obtiene siempre un valor booleano: *true* o *false.*

Operador	Descripción
===	Compara dos elementos, incluyendo su tipo interno. Si son de distinto tipo, no realiza conversión y devuelve false ya que siempre los considera diferentes. **Uso recomendado.**
!==	Compara dos elementos, incluyendo su tipo interno. Si son de distinto tipo, no

	realiza conversión y devuelve true ya que siempre los considera diferentes. **Uso recomendado.**
==	Devuelve el valor *true* cuando los dos operandos son iguales. Si los elementos son de distintos tipos, realiza una conversión. **No está recomendado su uso.**
!=	Devuelve el valor *true* cuando los dos operandos son distintos. Si los elementos son de distintos tipos, realiza una conversión. **No está recomendado su uso.**
>	Devuelve el valor *true* cuando el operando de la izquierda es mayor que el de la derecha.
<	Devuelve el valor *true* cuando el operando de la derecha es menor que el de la izquierda.
>=	Devuelve el valor *true* cuando el operando de la izquierda es mayor o igual que el de la derecha.
<=	Devuelve el valor *true* cuando el operando de la derecha es menor o igual que el de la izquierda.

```
let a=4;b=5,c="5";
console.log("El resultado de la expresión 'a==c' es igual a "+(a==c));
console.log("El resultado de la expresión 'a===c' es igual a "+(a===c));
console.log("El resultado de la expresión 'a!=c' es igual a "+(a!=c));
console.log("El resultado de la expresión 'a!==c' es igual a "+(a!==c));
console.log("El resultado de la expresión 'a==b' es igual a "+(a==b));
console.log("El resultado de la expresión 'a!=b' es igual a "+(a!=b));
console.log("El resultado de la expresión 'a>b' es igual a "+(a>b));
console.log("El resultado de la expresión 'a<b' es igual a "+(a<b));
console.log("El resultado de la expresión 'a>=b' es igual a "+(a>=b));
console.log("El resultado de la expresión 'a<=b' es igual a "+(a<=b));
```

5.4 Operadores lógicos

Los operadores lógicos se utilizan para el procesamiento de los valores booleanos. A su vez el valor que devuelven también es booleano: true o false.

Operador	Descripción
&&	Y “lógica”. El valor de devolución es true cuando ambos operandos son verdaderos.
||	O “lógica”. El valor de devolución es true cuando alguno de los operandos es verdadero o lo son los dos.

!	No “lógico”. Si el valor es true, devuelve false y si el valor es false, devuelve true.

Se muestra el resultado de distintas operaciones realizadas con operadores lógicos. (En el ejemplo se usa directamente los valore true y false en lugar de variables).

```
console.log("El resultado de la expresión 'false&&false' es igual a
"+(false&&false));
console.log("El resultado de la expresión 'false&&true' es igual a
"+(false&&true));
console.log("El resultado de la expresión 'true&&false' es igual a
"+(true&&false));
console.log("El resultado de la expresión 'true&&true' es igual a
"+(true&&true));
console.log("El resultado de la expresión 'false||false' es igual a
"+(false||false));
console.log("El resultado de la expresión 'false||true' es igual a
"+(false||true));
console.log("El resultado de la expresión 'true||false' es igual a
"+(true||false));
console.log("El resultado de la expresión 'true||true' es igual a
"+(true||true));
console.log("El resultado de la expresión '!false' es igual a "+(!false));
```

Para saber más de comparadores y expresiones, puedes consultar:
https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Expressions_and_Operators

6. ESTRUCTURAS DE CONTROL

6.1 Instrucciones if/else

Para controlar el flujo de información en los programas JavaScript existen una serie de estructuras condicionales y bucles que permiten alterar el orden secuencial de ejecución. Estas son las instrucciones if y else.

La instrucción if permite la ejecución de un bloque u otro de instrucciones en función de una condición.

Sintaxis

```
if (condición) {
    // bloque de instrucciones que se ejecutan si la condición se cumple
}
else{
    // bloque de instrucciones que se ejecutan si la condición no se cumple
```

```
}
```

Las llaves solo son obligatorias cuando haya varias instrucciones seguidas pertenecientes a la ramificación. Si no pones llaves, el if se aplicará únicamente a la siguiente instrucción.

Puede existir una instrucción if que no contenga la parte else. En este caso, se ejecutarán una serie de órdenes si se cumple la condición y si esto no es así, se continuaría con las órdenes que están a continuación del bloque if. Ejemplos de uso:

```
let diaSemana;
diaSemana=prompt("Introduce el día de la semana ", "");
if (diaSemana === "domingo")
{
    console.log("Hoy es festivo");
}
else // Al no tener {}, es un "bloque de una instrucción"
    console.log("Hoy no es domingo, a descansar!!");
```

Otro ejemplo

```
let edadAna,edadLuis;
// Usamos parseInt para convertir a entero
edadAna=parseInt(prompt("Introduce la edad de Ana",""));
edadLuis=parseInt(prompt("Introduce la edad de Luis",""));

if (edadAna > edadLuis){
    console.log("Ana es mayor que Luis.");
    console.log(" Ana tiene "+edadAna+" años y Luis "+ edadLuis);
}
else{
    console.log("Ana es menor o de igual edad que Luis.");
    console.log(" Ana tiene "+edadAna+" años y Luis "+ edadLuis);
}
```

6.2 Ramificaciones anidadas

Para las condiciones ramificadas más complicadas, a menudo se utilizan las ramificaciones anidadas. En ellas se definen consultas if dentro de otras consultas if, por ejemplo:

```
let edadAna,edadLuis;
// Convertirmos a entero las cadenas
edadAna=parseInt(prompt("Introduce la edad de Ana",""));
edadLuis=parseInt(prompt("Introduce la edad de Luis",""));
if (edadAna > edadLuis){
    console.log("Ana es mayor que Luis.");
}
else{
    if (edadAna<edadLuis){
```

```
        console.log("Ana es menor que Luis.");
    }else{
        console.log("Ana tiene la misma edad que Luis.");
    }
}
console.log(" Ana tiene "+edadAna+" años y Luis "+ edadLuis);
```

6.3. Switch

Se usa normalmente cuando se quiere comparar una variable con una serie de constantes.

Sintaxis:

```
switch (variable) {
  case valor_comparar1:
    // bloque de instrucciones que se ejecutan si variable===valor_comparar1
    break;
  case valor_comparar2:
    // bloque de instrucciones que se ejecutan si variable===valor_comparar2
    break;
  case valor_compararN:
    // bloque de instrucciones que se ejecutan si variable===valor_compararN
    break;
  default:
    // bloque de instrucciones que se ejecutan no se ha cumplido ningún case
y sale del switch
    break;
}
```

Ejemplo:

```
const metodoPago = 'efectivo';

switch(metodoPago) {
    case 'tarjeta':
        console.log('Pagaste con tarjeta');
        break;
    case 'cheque':
        console.log('El usuario va a pagar con cheque, revisaremos los fondo
s primero');
        break;
    case 'efectivo':
        console.log('Pagaste con efectivo');
        break;
    default:
        console.log('Aún no has pagado');
        break;
```

```
}
```

6.4. Operador condicional ternario

El operador condicional (ternario) es el único operador en JavaScript que tiene tres operandos. Este operador se usa con frecuencia como atajo para la instrucción if.

Sintaxis

condición ? expr1 : expr2

Si la condición es true, el operador retorna el valor de la expr1; de lo contrario, devuelve el valor de expr2. Por ejemplo, para mostrar un mensaje diferente en función del valor de la variable isMember, se puede usar esta declaración:

```
"La Cuota es de:  " + (isMember ? "$2.00" : "$10.00")
```

Puedes encontrar aquí más información: https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Operators/Conditional_Operator

6.5. No más else, emplea claúsulas de guarda

A veces nos encontramos que nuestro código contiene muchas sentencias if. Por ejemplo, aquí tenemos nuestra clase *User* que contiene un método para enviar notificaciones donde solo se enviarán si se cumplen una serie de condiciones.

```php
function register()
{
    if (!empty($_POST)) {
        $msg = '';
        if ($_POST['user_name']) {
            if ($_POST['user_password_new']) {
                if ($_POST['user_password_new'] === $_POST['user_password_repeat']) {
                    if (strlen($_POST['user_password_new']) > 5) {
                        if (strlen($_POST['user_name']) < 65 && strlen($_POST['user_name']) > 1) {
                            if (preg_match('/^[a-z\d]{2,64}$/i', $_POST['user_name'])) {
                                $user = read_user($_POST['user_name']);
                                if (!isset($user['user_name'])) {
                                    if ($_POST['user_email']) {
                                        if (strlen($_POST['user_email']) < 65) {
                                            if (filter_var($_POST['user_email'], FILTER_VALIDATE_EMAIL)) {
                                                create_user();
                                                $_SESSION['msg'] = 'You are now registered so please login';
                                                header('Location: ' . $_SERVER['PHP_SELF']);
                                                exit();
                                            } else $msg = 'You must provide a valid email address';
                                        } else $msg = 'Email must be less than 64 characters';
                                    } else $msg = 'Email cannot be empty';
                                } else $msg = 'Username already exists';
                            } else $msg = 'Username must be only a-z, A-Z, 0-9';
                        } else $msg = 'Username must be between 2 and 64 characters';
                    } else $msg = 'Password must be at least 6 characters';
                } else $msg = 'Passwords do not match';
            } else $msg = 'Empty Password';
        } else $msg = 'Empty Username';
        $_SESSION['msg'] = $msg;
    }
    return register_form();
}
```

Ejemplo de código

```javascript
class User
{
    constructor(){
        const NOTIFICATION_NOT_SENT = 0;
        const NOTIFICATION_SUCCESSFULLY_SENT = 1;
    }

    notify()
    {
        let user_id = this.id;
        if (this.isVerified(user_id)) {
            if (this.hasNotificationsEnabled(user_id)) {
                if (this.shouldUserBeNotified(user_id)) {
                    this.sendNotification();

                    return NOTIFICATION_SUCCESSFULLY_SENT;
                } else {
                    return NOTIFICATION_NOT_SENT;
                }
            } else {
                throw new NotificationIsNotEnabled;
            }
```

```
        } else {
            throw new UserIsNotVerified;
        }
    }

    isVerified(user_id) {}
    hasNotificationsEnabled(user_id) {}
    shouldUserBeNotified(user_id) {}
    sendNotification(user_id) {}
}
```

El código anterior en sí, no es incorrecto. Pero, si nos fijamos más detenidamente, nos daremos cuenta que tenemos tres sentencias if/else anidadas que contienen diferentes formas de tratar las devoluciones.

Además, estamos forzando a nuestro código a tener tres niveles de indentación (sangrado) que hacen más difícil leer nuestro código.

Y el último problema que podemos detectar es que, si tenemos una gran cantidad de código en los niveles más profundos del if, debemos desplazarnos hasta el final del método para saber qué hacen cada uno de los else.

Así que vamos a aplicar una técnica de refactorización[3] llamada *Guard Clauses* (cláusulas de guarda). La idea es negar las condiciones que tenemos en los if y si se cumple dicha negación, ejecutamos el código que teníamos en el else.

```
class User
{
    constructor(){
        const NOTIFICATION_NOT_SENT = 0;
        const NOTIFICATION_SUCCESSFULLY_SENT = 1;
    }

    notify()
    {
        let user_id = this.id;
        //Claúsula de guarda
        if (!this.isVerified(user_id)) {
            throw new UserIsNotVerified;
        }

        if (!this.hasNotificationsEnabled(user_id)) {
            throw new NotificationIsNotEnabled;
        }
```

[3] técnica para reestructurar un código fuente, alterando su estructura interna sin cambiar su comportamiento externo.

```
        if (!this.shouldUserBeNotified(user_id)) {
            return NOTIFICATION_NOT_SENT;
        }

        this.sendNotification();
        return NOTIFICATION_SUCCESSFULLY_SENT;
    }

    isVerified(user_id) {}
    hasNotificationsEnabled(user_id) {}
    shouldUserBeNotified(user_id) {}
    sendNotification(user_id) {}
}
```

Con esto obtenemos una serie de beneficios:

- Código más legible (*clean code*).
- Flujo de código fácil de seguir.
- Reducción de los niveles de indentación.

7. ESTRUCTURA REPETITIVAS (BUCLES)

7.1 Bucle for

Cuando la ejecución de un programa llega a un bucle for:

- Lo primero que hace es ejecutar la "Inicialización del índice", que solo se ejecuta una vez.
- A continuación, analiza la condición de prueba y si esta se cumple ejecuta las instrucciones del bucle.
- Cuando finaliza la ejecución de las instrucciones del bucle se realiza la modificación del índice, se retorna a la cabecera del bucle for y se realiza de nuevo la condición de prueba.
- Si la condición se cumple se ejecutan las instrucciones y si no se cumple la ejecución continúa en las líneas de código que siguen posteriores al bucle.

Sintaxis

```
for (Inicialización del índice; Condición de prueba; Modificación en el
índice){
    // ...instrucciones...
```

Ejemplo: números pares del 2 al 30

```
for (let i=2;i<=30;i+=2) {
    console.log(i);
}
```

```
console.log("Se han escrito los números pares del 2 al 30");
```

Ejemplo: Escribe las potencias de 2 hasta 3000

```
let aux=1;
for (i=2;i<=3000;i*=2) {
    console.log("2 elevado a "+aux+" es igual a "+i);
    aux++;
}
console.log("Se han escrito las potencias de 2 menores de 3000");
```

7.2 Bucle while

Con el bucle while se pueden ejecutar un grupo de instrucciones mientras se cumpla una condición.

- Si la condición nunca se cumple, entonces tampoco se ejecuta ninguna instrucción.
- Si la condición se cumple siempre, nos veremos inmersos en el problema de los bucles infinitos, que pueden llegar a colapsar el navegador, o incluso el ordenador.
    - Por esa razón es muy importante que la condición deba dejar de cumplirse en algún momento para evitar bucles infinitos.

Sintaxis

```
while (condición){
    //...instrucciones...
}
```

Ejemplo: Escribe los números pares de 0 a 30

```
let i=2;
while (i<=30) {
    console.log(i);
    i+=2;
}
console.log("Ya se han mostrado los números pares del 2 al 30");
```

Ejemplo: Pregunta una clave hasta que se corresponda con una dada.

```
let auxclave="";
while (auxclave!=="vivaYO"){
    auxclave=prompt("introduce la clave ","claveSecreta")
}
console.log("Has acertado la clave");
```

7.3 Bucle do-while

La diferencia del bucle do-while frente al bucle while reside en el momento en que se comprueba la condición: el bucle do-while no la comprueba hasta el final, es decir, después del cuerpo del bucle, lo que significa que el bucle do-while se recorrerá, una vez, como mínimo, aunque no se cumpla la condición.

Sintaxis

```
do {
    // ...instrucciones...
} while(condición);
```

Ejemplo: Preguntar por una clave hasta que se introduzca la correcta

```
let auxclave;
do {
    auxclave=prompt("introduce la clave ","vivaYo")
} while (auxclave!=="EstaeslaclaveJEJEJE")
console.log("Has acertado la clave");
```

7.4 Instrucciones BREAK y CONTINUE

En los bucles for, while y do-while se pueden utilizar las instrucciones break y continue para modificar el comportamiento del bucle.

La instrucción “break” dentro de un bucle hace que este se interrumpa inmediatamente, aun cuando no se haya ejecutado todavía el bucle completo. Al llegar la instrucción, el programa se sigue desarrollando inmediatamente a continuación del final del bucle.

Ejemplo: Pregunta por la clave y permitir tres respuestas incorrectas

```
let auxclave=true;
let numveces=0;
//Mientras no introduzca la clave y no se pulse Cancelar
while (auxclave !== "anonimo" && auxclave){
    auxclave=prompt("Introduce la clave ","");
    numveces++;
    if (numveces === 3)
        break;
}
if (auxclave=="SuperClave"){
 console.log("La clave es correcta");
}else{
 console.log("La clave no es correcta correcta");
}
```

El efecto que tiene la instrucción “continue” en un bucle es el de hacer retornar a la

secuencia de ejecución a la cabecera del bucle, volviendo a ejecutar la condición o a incrementar los índices cuando sea un bucle for. Esto permite saltarse recorridos del bucle.

Ejemplo: Presenta todos los números pares del 0 al 50 excepto los que sean múltiplos de 3

```
let i;
for (i=2;i<=50;i+=2){
    if ((i%3)===0)
        continue;
    console.log(i);
}
```

No se aconseja su uso (sobre todo con continue), ya que rompen con la secuencia de ejecución del programa y hace difícil seguir su traza.

8. MANEJO DE ERRORES

Se utiliza la estructura try{}catch(error){}

```
try {
  console.log("En el Try se agrega el código a evaluar");//se ejecuta
  noExiste;//Lanza un error
  console.log("Segundo mensaje en el try");//no se ejecuta porque salta un
error en la línea anterior
} catch (error) {
  console.log("Catch, captura cualquier error surgido o lanzado en el
try");//se ejecuta
  console.log(error);//Se ejecuta
} finally {
  console.log("El bloque finally se ejecutará siempre al final de unbloque
try-catch");//se ejecuta siempre
}
```

También se puede lanzar un error con throw new Error (“mensaje de error”) y eso será capturado por el catch.

```
try {
  let numero = "y";
  if (isNaN(numero)) {
    throw new Error("El carácter introducido no es un Número");
  }
  console.log(numero * numero);
} catch (error) {
  console.log(`Se produjo el siguiente error: ${error}`);
}
```

9. FUNCIONES

Una función es un conjunto de instrucciones que se agrupan bajo un nombre de función. Se ejecuta solo cuando es llamada por su nombre en el código del programa. La llamada provoca la ejecución de las órdenes que contiene.

Las funciones son muy importantes por diversos motivos:

- Ayudan a estructurar los programas para hacerlos su código más comprensible y más fácil de modificar.
- Permiten repetir la ejecución de un conjunto de órdenes todas las veces que sea necesario sin necesidad de escribir de nuevo las instrucciones.

La sintaxis de una función *clásica* consta de las siguientes partes básicas:

- Un nombre de función.
- Los parámetros pasados a la función separados por comas y entre paréntesis.
- Las llaves de inicio y final de la función.
- Desde Javascript ES6, se pueden definir valores por defecto para los parámetros.

Sintaxis de la definición de una función

```
function nombrefuncion (parámetro1, parámetro2=valorPorDefecto...){
    // instrucciones
    //si la función devuelve algún valor añadimos:
    return valor;
    }

    const nombrefuncion  = function (parámetro1,
parámetro2=valorPorDefecto...){
    // instrucciones
    //si la función devuelve algún valor añadimos:
    return valor;
    }

    //IIFE función anónima autoejecutable (No necesitan llamarse porque se
llaman ellas mismas)
    (function (parámetro1, parámetro2=valorPorDefecto...){
    // instrucciones
    //si la función devuelve algún valor añadimos:
    return valor;
    })();
//Ejemplo IIFE
(function () {
  console.log("Mi primer IIFE");//Mi primer IIFE
})();
//Ejemplo IIFE con parámetros
(function (d, w, c) {
```

```
  console.log("Mi segunda IIFE");
  console.log(d);
  console.log(w);
  c.log("Este es un console.log")
})(document, window, console);
```

Sintaxis de la llamada a una función

```
// La función se ejecuta siempre que se ejecute la sentencia.
valorRetornado=nombrefuncion (parám1, parám2...);
```

Nota: En algunas bibliografías, a los parámetros de la llamada a una función se les llama argumentos.

Es importante entender la **diferencia entre definir una función y llamarla:**

- Definir una función es simplemente especificar su nombre y definir qué acciones realizará en el momento en que sea invocada, mediante la palabra reservada function.
- Para llamar a una función es necesario especificar su nombre e introducir los parámetros que queremos que utilice. Esta llamada se puede efectuar en una línea de órdenes o bien a la derecha de una sentencia de asignación en el caso de que la función devuelva algún valor debido al uso de la instrucción return.

La definición de una función se puede realizar en cualquier lugar del programa, pero se recomienda hacerlo al principio del código o en un fichero "js" que contenga funciones.

La llamada a una función se realizará cuando sea necesario, es decir, cuando se demande la ejecución de las instrucciones que hay dentro de ella.

Ejemplo: funciones que devuelve la suma de dos valores que se pasan por parámetros y que escriben el nombre del profesor.

```
// Definiciones de las funciones
function suma (a,b){
    // Esta función devuelve un valor
    return a+b;
}

// Esta función muestra un texto, pero no devuelve un valor
function profe (){
    console.log("El profesor es muy bueno");
    // OJO: esto es un ejemplo, pero rara vez se realiza en una función real
}
// Código que se ejecuta
let op1=5, op2=25;
let resultado;
// Llamada a función
```

```
resultado=suma(op1,op2);
// llamada a la función
console.log (op1+"+"+op2+"="+resultado);
// Llamada a función
profe();
```

Recordad que dentro de las funciones rara vez se utilizan funciones de entrada/salida. El 99.9% de las veces simplemente procesan la entrada por parámetros y devuelven un valor.

Desde Javascript ES6, las funciones soportan los llamados parámetros REST.

Los parámetros REST son un conjunto de parámetros que se almacenan como array en un "parámetro final" indicado con **...nombreParametro**. Esto nos permite manejar la función sin tener que controlar el número de parámetros con los que esta es llamada. Sólo el último parámetro puede ser REST.

Ejemplo:

```
function pruebaParREST(a, b, ...masParametros) {
    console.log("a: "+a+" b: "+ b + " otros: " + masParametros);
  }
  pruebaParREST("param1", "param1", "param3", "param4", "param5");
```

Para saber más de los parámetros REST

https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Funciones/parametros_rest

8.1. Tratamiento numérico

Hay dos funciones predefinidas (**parseInt y parseFloat**) que ya comentamos anteriormente. En este punto, además de recordarlas, ampliamos con la función **isNaN** que nos ayudará a distinguir si una cadena es un número o no.

- **parseInt(cadena)**: convierte una cadena de texto entero a entero .
- **parseFloat(cadena)**: convierte una cadena de texto en decimal.
- **isNaN(cadena)**: NaN es la abreviación de "Not a Number". Esta función comprueba si una cadena de caracteres puede ser considerada un número (false) o no (true).

Ejemplo:

```
let numero;
do{
    numero=prompt("Esto se repetirá hasta que metas un número");
}while(isNaN(numero));
```

8.2. Funciones flecha (arrow functions)

Una función flecha (arrow function) es una alternativa compacta al uso de funciones

tradicionales.

- Este tipo de funciones tienen sus limitaciones y deben ser utilizadas solo en algunos contextos donde sean útiles.
- No son adecuadas para ser utilizadas como métodos.

Soporta varias formas de sintaxis, a continuación, indicamos las más típicas:

- (parametro1, parametro2, ...) => {sentencias}
- () => {sentencias}
- parámetro => sentencia

A veces, son utilizados con la **función "map" de los arrays**. Esta función crea un nuevo array formado por la aplicación de una función a cada uno de sus elementos:

https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/map

También es muy útil con la **función "reduce" de los arrays**. Esta función ejecuta una "función reductora" sobre cada elemento del array, devolviendo un único valor.

https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/reduce

Ejemplo:

```
// Arrow Functions
//Expresión de la función
const sumar = function (n1,n2) {
    return n1+n2;
}
/*Arrow function equivalente: se elimina la palabra reservada function y fue
ra del paréntesis de los parámetros pones la flecha =>
además las {} son opcionales si solo tienes una línea y el return es
implícito. Si no hubiera parámetros hay que poner los paréntesis ()*/
const sumar = (n1, n2) => n1 + n2;

const res=sumar(5, 12);
console.log(`El resultado es ${res}`);

//Expresión de la función
const aprendiendo =function (tecnologia){
    console.log(`Aprendiendo ${tecnologia}`)
}

/*Arrow function equivalente: se elimina la palabra reservada function y fue
ra del paréntesis de los parámetros pones la flecha =>
```

```
además las {} son opcionales si solo tienes una línea y puedes quitar los pa
réntesis () si sólo tienes un parámetro Si no hubiera parámetros hay que
poner los paréntesis ()*/
const aprendiendo = tecnologia => console.log(`Aprendiendo ${tecnologia}`)

aprendiendo('JavaScript');
```

Ejemplo:

```
let nombres = ['Pedro',  'Juan',  'Elena'];
console.log(nombres.map(nom => nom.length));
// Muestra el array con los valores [5, 4, 5]
let sumaNombres= nombres.reduce((acumulador, elemento) => {
  return acumulador + elemento.length;
}, 0);
// Muestra la suma de la longitud de los nombres
console.log(sumaNombres);
```

Para saber más:

https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Funciones/Arrow_functions

8.3. Funciones recursivas

En términos simples: recursión es cuando una función sigue llamándose a sí misma, hasta que ya no tiene que hacerlo. Si, la función se sigue llamándose a sí misma, pero con una entrada más pequeña cada vez.

Piensa en recursión como una carrera en un circuito. Es como correr la misma pista una y otra vez, pero las vueltas se hacen más pequeñas cada vez. Eventualmente, correrás la vuelta más pequeña y se terminará la carrera.

Lo mismo con la recursión: La función sigue llamándose a sí misma, cada vez con una entrada menor hasta que eventualmente se detiene.

Pero, la función no decide por sí misma cuando parar, nosotros le decimos cuando. Nosotros le damos a la función una condición conocida como caso base.

El caso base es la condición que le dice a la función cuando dejar de llamarse a sí misma. Es como decirle a la función cuál será la última vuelta de la carrera para que se detenga después de la última vuelta.

Muy bien eso es recursión. Vamos a mirar algunos ejemplos para entender cómo funciona la recursión.

¿Recuerdas la primera vez que aprendiste acerca de los bucles (loops)? El primer ejemplo que hiciste probablemente fue un programa que hacia una cuenta atrás. Vamos a hacer eso.

Primero, necesitamos entender que queremos que haga nuestro programa. Cuenta atrás desde un número dado hasta el número más pequeño, restando 1 cada vez que pasa por el bucle.

Dado el número 5, esperamos que la salida sea algo así:

```
// 5
// 4
// 3
// 2
// 1
```

Muy bien, ¿Cómo podemos programar este programa con recursión?

```
let cuentaAtras = numero => {
    //base case
    if (numero === 0) {
        return;
    }
    console.log(numero);
    return cuentaAtras(numero - 1);
};
console.log(cuentaAtras(5)) // 5, 4, 3, 2, 1
```

¿Qué exactamente es lo que pasa aquí?

Si te diste cuenta, lo primero que hicimos fue definir el caso base. ¿Por qué? Porque la función primero que nada necesita saber cuándo dejará de llamarse a sí misma.

Si no le dices a la función cuándo detenerse, entonces sucederá algo llamado stackoverflow. La pila(stack) se va a llenar con funciones que se están llamando, pero que no regresan ni se quitan de la pila.

La parte de recursión sucede en la línea 7, donde le decimos a la función que siga ejecutándose, pero reduciendo sus entradas en uno cada vez que se ejecuta.

Así que, efectivamente, esto es lo que sucede:

```
1// La entrada actual es 5
2// Es 5 igual a 0 ?
3// No, Ok entonces imprime 5 en la consola.
4// Se llama a si misma de nuevo con el numero - 1 o 5 - 1;
5// La entrada principal es 4
6// Es 4 igual a 0 ?
7// No, Ok entonces imprime 4 en la consola.
8// Repite hasta que la entrada sea 0, y asi la función deja de llamarse a s
i misma.
```

Vamos con otro ejemplo. ¿Sabes cómo podemos saber que un número es par usando el operador de módulo (%)? Entonces, si cualquier número % 2 == 0 entonces ese número es par o si cualquier número % 3 == 0 entonces ese número es impar.

Bueno, resulta que existe otro método.

Si continuamente restamos dos a un número hasta que el número más pequeño sea 0(cero) o 1(uno) entonces podemos saber si el número es par o impar.

Intentemos eso con recursión. Entonces, dado el número 6, nuestro programa debería devolver 'Par' porque 6-2-2-2 = 0. Con el número 7, nuestro programa debería devolver 'impar' porque 7-2-2-2 = 1.

Veamos el código.

```
let parImpar = (numero) => {
    if (numero === 0) {
        return 'Par';
    } else if (numero === 1) {
        return 'Impar';
    } else {
        return parImpar(numero - 2);
    }
};
console.log(parImpar(20)) // Par
console.log(parImpar(75)) // Impar
console.log(parImpar(98)) // Par
console.log(parImpar(113)) // Impar
```

De nuevo, el primer paso fue decirle a la función cuándo dejar de llamarse a sí misma. Luego le dijimos qué hacer cuando se llama a sí misma.

10. STRING

Cuando creamos una cadena, implícitamente esta cadena se convierte en un objeto String, con sus propiedades y métodos predefinidos. El objeto predefinido String es útil porque nos ayuda a realizar múltiples operaciones con cadenas. Los métodos de manipulación de cadenas de Javascript, no modifican al objeto actual, sino que devuelven el objeto resultante de aplicar la modificación. Si queremos que la modificación se aplique sobre la misma cadena que estamos trabajando, haremos algo así:

```
cadena=cadena.metodoQueDevuelveUnaModificacion();
```

En este objeto, hay una propiedad muy utilizada llamada “length”. Esta propiedad nos indica cuantos elementos (caracteres) tiene la cadena.

También existen una serie de métodos muy útiles.

- **toLowerCase()/toUpperCase()**: devuelve la cadena convertida a minúsculas/mayúsculas.
- **concat(cadena)**: devuelve el objeto con el valor de cadena concatenado al final.
- **charAt(posicion)**: devuelve el carácter que se encuentre en la posición solicitada. Debemos tener en cuenta que las posiciones comienzan a contar desde cero.
- **indexOf(texto, [indice])**: devuelve la primera posición donde se encuentra el texto buscado, empezando a buscar desde la posición "indice". Si "indice" no se indica, se toma por defecto el valor 0.
- **lastIndexOf (texto, [indice])**: como la anterior. Busca "hacia atrás" la primera ocurrencia del texto buscado. Indice indica desde qué punto se empieza a buscar "hacia atrás". Si no se indica el valor de "indice", se busca desde el final.
- **replace(texto1,texto2)**: busca ocurrencias de la cadena texto1 y las reemplaza por texto2.
- **split(caracter, [trozos])**: separa la cadena mediante un carácter separador. Trozos indica el máximo de separaciones. Si no se indica, se harán todas las separaciones posibles.
- **substring(inicio, [fin]):** devuelve la subcadena comprendida entre la posición inicio y la posición fin. Si fin no se indica, se toma como valor el final de la cadena.
- **includes(texto):** devuelve true o false dependiendo de si "texto" aparece o no.
- **trim ():** elimina los espacios en blanco a ambos lados de la cadena.
- **match()**: se usa para obtener todas las ocurrencias de una expresión regular dentro de una cadena.

Ejemplo de algunas de las funciones:

```
let cad="Martin:Barea:123456";
let tfo;
cad=cad.toUpperCase();
alert(cad);
splitTodosCampos=cad.split(":");
split1Campo=cad.split(":",1);
alert(splitTodosCampos);
alert(split1Campo);
tfo=splitTodosCampos[2];
//Cambio en el telefono los números 3 por 9s
tfo=tfo.replace("2","9");
alert(tfo);
//Muestro el quinto número del teléfono
alert(tfo.charAt(4));
alert("Bienvenido al CEEDCV");
// Includes (retorna true o false)
console.log(tweet.includes('JavaScript'));
console.log(producto2.includes('Tablet'));
```

Puedes encontrar aquí

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/String
más información.

10.1. Expresiones regulares (RegExp)

Las **expresiones regulare**s (a menudo llamadas **RegExp** o RegEx) son un sistema para buscar, capturar o reemplazar texto utilizando **patrones**. Estos patrones permiten realizar una búsqueda de texto de una forma relativamente sencilla y abstracta, de forma que abarca una gran cantidad de posibilidades que de otra forma sería imposible o muy costosa.

Constructor	Descripción
REGEXP `new RegExp(r, flags)`	Crea una nueva expresión regular a partir de `r` con los `flags` indicados.
REGEXP `/r/flags`	Simplemente, la expresión regular `r` entre barras `/`. **Notación preferida.**

Así pues, podríamos crear expresiones regulares de estas dos formas:

```
// Notación literal
const r = /.a.o/i;

// Notación de objeto
const r = new RegExp(".a.o", "i");
const r = new RegExp(/.a.o/, "i");
```

En ambos ejemplos, estamos estableciendo la expresión regular .a.o, donde el punto (como veremos más adelante) es un comodín que simboliza cualquier carácter, y la i es un flag que establece que no diferencia mayúsculas de minúsculas.

Cada expresión regular creada, tiene unas propiedades definidas, donde podemos consultar ciertas características de la expresión regular en cuestión. Además, también tiene unas propiedades de comprobación para saber si un flag determinado está activo o no:

Propiedades	Descripción
STRING `.source`	Devuelve un string con la expresión regular original al crear el objeto (*sin flags*).
STRING `.flags`	Devuelve un string con los flags activados en la expresión regular.
NUMBER `.lastIndex`	Devuelve la posición donde se encontró una ocurrencia en la última búsqueda.
BOOLEAN `.global`	Comprueba si el flag `g` está activo en la expresión regular.
BOOLEAN `.ignoreCase`	Comprueba si el flag `i` está activo en la expresión regular.
BOOLEAN `.multiline`	Comprueba si el flag `m` está activo en la expresión regular.
BOOLEAN `.unicode`	Comprueba si el flag `u` está activo en la expresión regular.
BOOLEAN `.sticky`	Comprueba si el flag `y` está activo en la expresión regular.

Con las propiedades .source y .flags se puede obtener casi toda la información que se puede hacer con dichos flags.

```
const r = /reg/gi;
```

```
r.source; // 'reg'
r.flags; // 'ig'

r.flags.includes("g"); // true (equivalente a r.global)
r.flags.includes("u"); // false (equivalente a r.unicode)
```

También podemos comprobar si un flag está activo con la propiedad .flags combinada con includes(), como se puede ver en el ejemplo anterior. Por otro lado, .source nos devuelve un string con la expresión regular definida (y sin flags). La utilidad de lastIndex() la veremos más adelante.

El segundo parámetro del new RegExp() o el que se escribe después de la segunda barra / delimitadora del literal de las expresiones regulares, son una serie de caracteres que indican los **flags** activos en la expresión regular en cuestión:

```
const r1 = /reg/;
const r2 = /reg/i;
const r3 = /reg/gi;
```

La expresión regular r1 no tiene ningún flag activado, mientras que r2 tiene el flag i activado y r3 tiene el flag i y el flag g activado. Veamos para que sirve cada flag:

Flag	Booleano	Descripción
i	.ignoreCase	Ignora mayúsculas y minúsculas. Se suele denominar **insensible a mayús/minús**.
g	.global	Búsqueda global. Sigue buscando coincidencias en lugar de pararse al encontrar una.
m	.multiline	Multilínea. Permite a ^ y $ tratar los finales de línea \r o \n.
u	.unicode	Unicode. Interpreta el patrón como un código de una secuencia Unicode.
y	.sticky	Sticky. Busca sólo desde la posición indicada por lastIndex.

Cada una de estas flags se pueden comprobar si están activas desde Javascript con su booleano asociado, que es una propiedad de la expresión regular:

```
const r = /reg/gi;

r.global; // true
r.ignoreCase; // true
r.multiline; // false
r.sticky; // false
r.unicode; // false
```

Los **objetos RegExp** tienen varios métodos para utilizar expresiones regulares contra textos y saber si «casan» o no, es decir, si el patrón de la expresión regular encaja con el texto propuesto.

Método	Descripción
BOOLEAN test(str)	Comprueba si la expresión regular «casa» con el texto str pasado por parámetro.
ARRAY exec(str)	Ejecuta una búsqueda de patrón en el texto str. Devuelve un array con las capturas.

Por ejemplo, veamos cómo utilizar la expresión regular del ejemplo anterior con el método **test()** para comprobar si encaja con un texto determinado:

```
const r = /.a.o/i;

r.test("gato"); // true
r.test("pato"); // true
r.test("perro"); // false
r.test("DATO"); // true   (el flag i permite mayús/minús)
```

El método **exec()** lo veremos un poco más adelante, cuando estudiemos la **captura de patrones**, ya que es algo más complejo. Primero debemos aprender que caracteres especiales existen en las expresiones regulares para dominarlas.

Antes de comenzar a utilizar expresiones regulares hay que aprender la parte más compleja de ellas: los **caracteres especiales**. Dentro de las expresiones regulares, existen ciertos caracteres que tienen un significado especial, y también, muchos de ellos dependen de donde se encuentren para tener ese significado especial, por lo que hay que aprender bien cómo funcionan. Empecemos con algunos de los más sencillos:

Caracter especial	Descripción
.	Comodín, cualquier caracter.
\	Invierte el significado de un carácter. Si es especial, lo escapa. Si no, lo vuelve especial.
\t	Caracter especial. Tabulador.
\r	Caracter especial. Retorno de carro. A menudo denominado CR.
\n	Caracter especial. Nueva línea. A menudo denominado «line feed» o LF.

En esta pequeña tabla vemos algunos caracteres especiales que podemos usar en expresiones regulares. Observa que al igual que con otros tipos de datos, podemos utilizar el método **test()** sobre el literal de la expresión regular, sin necesidad de guardarla en una variable previamente:

```
// Buscamos RegExp que encaje con "Manz"
/M.nz/.test("Manz"); // true
/M.nz/.test("manz"); // false  (La «M» debe ser mayúscula)
/M.nz/i.test("manz"); // true   (Ignoramos mayús/minús con el flag «i»)

// Buscamos RegExp que encaje con "A."
/A./.test("A."); // true   (Ojo, nos da true, pero el punto es comodín)
/A./.test("Ab"); // true   (Nos da true con cualquier cosa)
```

```
/A\./.test("A."); // true   (Solución correcta)
/A\./.test("Ab"); // false  (Ahora no deja pasar algo que no sea punto)
```

Dentro de las expresiones regulares los **corchetes []** tienen un significado especial. Se trata de un mecanismo para englobar un **conjunto de caracteres personalizado**. Por otro lado, si incluimos un circunflejo **^** antes de los caracteres del corchete, invertimos el significado, pasando a ser que **no exista** el conjunto de caracteres personalizado. Por último, tenemos el **«pipe» |**, con el que podemos establecer alternativas.

Caracter especial	Descripción
[]	Rango de carácteres. Cualquiera de los caracteres del interior de los corchetes.
[^]	No exista cualquiera de los caracteres del interior de los corchetes.
|	Establece una alternativa: lo que está a la izquierda o lo que está a la derecha.

Veamos un ejemplo aplicado a esto, que se verá más claro:

```
const r = /[aeiou]/i; // RegExp que acepta vocales (mayús/minús)

r.test("a"); // true (es vocal)
r.test("E"); // true (es vocal, y tiene flag «i»)
r.test("t"); // false (no es vocal)

const r = /[^aeiou]/i; // RegExp que acepta lo que no sea vocal
(mayús/minús)

r.test("a"); // false
r.test("E"); // false
r.test("T"); // true
r.test("m"); // true

const r = /casa|cama/; // RegExp que acepta la primera o la segunda opción

r.test("casa"); // true
r.test("cama"); // true
r.test("capa"); // false
```

En el interior de los corchetes, si establecemos dos caracteres separados por guion, por ejemplo **[0-9]**, se entiende que indicamos el rango de caracteres entre **0 y 9**, sin tener que escribirlos todos explícitamente.

De esta forma podemos crear **rangos** como [A-Z] (mayúsculas) o [a-z] (minúsculas), o incluso varios rangos específicos como [A-Za-z0-9]:

Caracter especial	Alternativa	Descripción
[0-9]	\d	Un dígito del 0 al 9.
[^0-9]	\D	No exista un dígito del 0 al 9.
[A-Z]		Letra mayúscula de la A a la Z. Excluye ñ o letras acentuadas.
[a-z]		Letra minúscula de la a a la z. Excluye ñ o letras acentuadas.
[A-Za-z0-9]	\w	Carácter alfanumérico (letra mayúscula, minúscula o dígito).
[^A-Za-z0-9]	\W	No exista carácter alfanumérico (letra mayúscula, minúscula o dígito).
[\t\r\n\f]	\s	Carácter de espacio en blanco (espacio, TAB, CR, LF o FF).
[^ \t\r\n\f]	\S	No exista carácter de espacio en blanco (espacio, TAB, CR, LF o FF).
	\xN	Carácter hexadecimal número N.
	\uN	Carácter Unicode número N.

Observa que en esta tabla tenemos una **notación alternativa** que es equivalente al carácter especial indicado. Por ejemplo, es lo mismo escribir [0-9] que \d. Algunos programadores encuentran más explicativa la primera forma y otros más cómoda la segunda.

Dentro de las expresiones regulares, las **anclas** son un recurso muy importante, ya que permiten delimitar los patrones de búsqueda e indicar si empiezan o terminan por caracteres concretos, siendo mucho más específicos al realizar la búsqueda:

Caracter especial	Descripción
^	Ancla. Delimina el inicio del patrón. Significa **empieza por**.
$	Ancla. Delimina el final del patrón. Significa **acaba en**.
\b	Posición de una palabra limitada por espacios, puntuación o inicio/final.
\B	Opuesta al anterior. Posición entre 2 caracteres alfanuméricos o no alfanuméricos.

Las dos primeras son bastante útiles cuando sabemos que el texto que estamos buscando termina o empieza de una forma concreta. De este modo podemos hacer cosas como las siguientes:

```
const r = /^mas/i;

r.test("Formas"); // false  (no empieza por "mas")
r.test("Master"); // true
r.test("Masticar"); // true

const r = /do$/i;

r.test("Vívido"); // true
r.test("Dominó"); // false
```

Por otro lado, **\b** nos permite indicar si el texto adyacente está seguido o precedido de un límite de palabra (espacio), puntuación (comas o puntos) o inicio o final del string:

```
const r = /fo\b/;

r.test("Esto es un párrafo de texto."); // true (tras "fo" hay un límite de palabra)
r.test("Esto es un párrafo."); // true (tras "fo" hay un signo de puntuación)
r.test("Un círculo es una forma."); // false (tras "fo" sigue la palabra)
r.test("Frase que termina en fo"); // true (tras "fo" termina el string)
```

Por último, **\B** es la operación opuesta a \b, por lo que podemos utilizarla cuando nos interesa que el texto no esté delimitado por una palabra, puntuación o string en sí.

En las expresiones regulares los **cuantificadores** permiten indicar cuántas veces se puede repetir el carácter inmediatamente anterior. Existen varios tipos de cuantificadores:

Caracter especial	Descripción
*	El carácter anterior puede aparecer 0 o más veces.
+	El carácter anterior puede aparecer 1 o más veces.
?	El carácter anterior puede aparecer o no aparecer.
{n}	El carácter anterior aparece n veces.
{n,}	El carácter anterior aparece n o más veces.
{n,m}	El carácter anterior aparece de n a m veces.

Veamos algunos ejemplos para aprender a aplicarlos. Comencemos con * (0 o más veces):

```
// 'a' aparece 0 o más veces en el string
const r = /a*/;

r.test(""); // true ('a' aparece 0 veces)
r.test("a"); // true ('a' aparece 1 veces)
r.test("aa"); // true ('a' aparece 2 veces)
r.test("aba"); // true ('a' aparece 2 veces)
r.test("bbb"); // true ('a' aparece 0 veces)
```

El cuantificador **+** es muy parecido a *, sólo que con el primero es necesario que el carácter anterior aparezca al menos una vez:

```
// 'a' aparece 1 o más veces (equivalente a /aa*/)
const r = /a+/;

r.test(""); // false ('a' aparece 0 veces)
```

```
r.test("a"); // true ('a' aparece 1 veces)
r.test("aa"); // true ('a' aparece 2 veces)
r.test("aba"); // true ('a' aparece 2 veces)
r.test("bbb"); // false ('a' aparece 0 veces)
```

El cuantificador **?** se suele utilizar para indicar que el carácter anterior es opcional (puede aparecer o puede no aparecer). Normalmente se utiliza cuando quieres indicar que no importa que aparezca un carácter opcional:

```
const r = /disparos?/i;

r.test("Escuché disparos en la habitación."); // true
r.test("Efectuó un disparo al sujeto."); // true
r.test("La pistola era de agua."); // false
```

Los tres cuantificadores siguientes, se utilizan cuando necesitamos concretar más el número de repeticiones del carácter anterior. Por ejemplo, **{n}** indica un número exacto, **{n,}** indica al menos n veces y **{n,m}** establece que se repita de n a m veces.

```
// Un número formado de 2 dígitos (del 0 al 9)
const r = /[0-9]{2}/;

r.test(42); // true
r.test(88); // true
r.test(1); // false
r.test(100); // true
```

Observa que **el último aparece como true**. Esto ocurre porque en la expresión regular no se han establecido anclas que delimiten el inicio y/o el final del texto. Si las añadimos, es más estricto con las comprobaciones:

```
const r = /^[0-9]{2}$/;

r.test(4); // false
r.test(55); // true
r.test(100); // false

const r = /^[0-9]{3,}$/;

r.test(33); // false
r.test(4923); // true

const r = /^[0-9]{2,5}$/;

r.test(2); // false
r.test(444); // true
r.test(543213); // false
```

Si quieres profundizar con las expresiones regulares, puedes jugar a RegEx People, un pequeño y básico juego para aprender a utilizar las expresiones regulares y buscar patrones, con su código fuente disponible en GitHub.

Pero con las **expresiones regulares** no sólo podemos realizar búsquedas de patrones. Una de las características más importantes de las expresiones regulares es lo potente y versátil que resultan las **capturas de patrones**.

Toda expresión regular que utilice la **parentización** (englobe con paréntesis fragmentos de texto) está realizando implícitamente una captura de texto, que es muy útil para obtener rápidamente información.

Para ello, dejamos de utilizar el método **test(str)** y comenzamos a utilizar **exec(str)**, que funciona exactamente igual, sólo que devuelve un array con las capturas realizadas. Antes de empezar a utilizarlo, necesitamos saber detalles sobre la **parentización**:

Caracter especial	Descripción
(x)	El patrón incluído dentro de paréntesis se captura y se guarda en $1 o sucesivos.
(?:x)	Si incluímos ?: al inicio del contenido de los paréntesis, evitamos capturar ese patrón.
x(?=y)	Busca sólo si x está seguido de y.
x(?!y)	Busca sólo si x no está seguido de y.

Así pues, vamos a realizar una captura a través de los paréntesis de una expresión regular:

```
// RegExp que captura palabras de 3 letras.
const r = /\b([a-z]{3})\b/gi;
const str = "Hola a todos, amigos míos. Esto es una prueba que permitirá ver
que ocurre.";

r.global; // true (el flag global está activado)

r.exec(str); // ['una', 'una']     index: 35
r.exec(str); // ['que', 'que']     index: 46
r.exec(str); // ['ver', 'ver']     index: 60
r.exec(str); // ['que', 'que']     index: 64
r.exec(str); // null
```

El método **exec()** nos permite ejecutar una búsqueda sobre el texto str hasta encontrar una coincidencia. En ese caso, se detiene la búsqueda y nos devuelve un array con los **string capturados por la parentización**. Si el flag **g** está activado, podemos volver a ejecutar exec() para continuar buscando la siguiente aparición, hasta que no encuentre ninguna más, que devolverá **null**.

Quizás, generalmente el usuario prefiera utilizar el método **match(reg) de los string**, que permiten ejecutar la búsqueda de la expresión regular reg pasada por parámetro, sobre esa variable de texto. El resultado es que nos devuelve un array con los string capturados:

```
const r = /\b([a-z]{3})\b/gi;
const str = "Hola a todos, amigos míos. Esto es una prueba que permitirá ver
que ocurre.";

str.match(r); // Devuelve ['una', 'que', 'ver', 'que']

const r = /\bv([0-9]+)\.([0-9]+)\.([0-9]+)\b/;
const str = "v1.0.21";

str.match(r); // Devuelve ['v1.0.21', '1', '0', '21']
```

En el caso de no existir **parentización**, el array devuelto contiene un string con todo el texto capturado. En el caso de existir múltiples parentizaciones (como en el último ejemplo), el array devuelto contiene un string con todo el texto capturado, y un string por cada parentización.

Recuerda que los string tienen varios métodos que permiten el uso de expresiones regulares para realizar operaciones, como por ejemplo, **replace()**, para hacer reemplazos en todas las ocurrencias:

```
const daenerys = "Javascript es un gran gran lenguaje";

daenerys.replace(/[aeou]/g, "i"); // 'Jiviscript is in grin grin lingiiji'

daenerys.replace(/gran/g, ""); // 'Javascript es un   lenguaje '
```

Ten en cuenta que puedes usar un string literal en lugar de un patrón.

Sobre el uso de expresiones regulares tenéis más información en
https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Regular_Expressions
Puedes probar tus expresiones regulares de forma online a través de https://regex101.com/, por ejemplo.

11. OBJETOS LITERALES

Un Objeto literal es un valor en memoria al que podemos acceder por un identificador. En JavaScript los objetos pueden ser vistos como una colección de propiedades.

Para acceder a las propiedades se puede usar la notación por punto o por corchetes.

```
let persona = {
    nombre: "Marcel",
    edad: "38",
    ciudad: "Alaior",
  };

  let edad = persona.edad;
  persona.edad=3;
  edad  38
  persona  { nombre: 'Marcel', edad: 3, ciudad: 'Alaior' }

  let ciudad1 = persona["ciudad"];
  persona["ciudad"]="Granada";

  ciudad1  Alaior
  persona  { nombre: 'Marcel', edad: 3, ciudad: 'Granada' }

  const producto = {
    nombre: 'manzana',
    categoria: 'frutas',
    precio: 1.99,
    nutrientes : {
      carbs: 0.95,
      grasas: 0.3,
      proteina: 0.2
    }
  }

  producto["nutrientes"].carbs = 1.25
  producto  { nombre: 'manzana', categoria: 'frutas', precio: 1.99, nutrientes: { carbs: 1.25, grasas: 0.3, proteina: 0.2 } }
```

También es posible definir funciones dentro del objeto literal, pero entonces se suele recurrir a las clases, como veremos más adelante.

```
const jon = {
  nombre: "Javier",
  apellido: "Ferrer",
  edad: 18,
  pasatiempos: ["Jugar videojuegos", "Hacer ejercicio", "Leer"],
  soltero: false,
  contacto: {
    email: "javier.ferrer@gmail.com",
    twitter: "@jferrer",
    movil: "123456789"
  },
  saludar() {
    console.log(`Hola :)`)
  },
  decirMiNombre() {
    console.log(`Hola me llamo ${this.nombre} ${this.apellido} y tengo
${this.edad} años y me puedes seguir como ${this.contacto.twitter} en
twitter.`)
  }
}
}
console.log(jon);
```

```
console.log(jon["nombre"]);
console.log(jon["apellido"]);
console.log(jon.nombre);
console.log(jon.apellido);
console.log(jon.edad);
console.log(jon.soltero);
console.log(jon.pasatiempos);
console.log(jon.pasatiempos[1]);
console.log(jon.contacto);
console.log(jon.contacto.twitter);
jon.saludar();
jon.decirMiNombre();
console.log(Object.keys(jon));
console.log(Object.values(jon));
console.log(jon.hasOwnProperty("nombre"));
console.log(jon.hasOwnProperty("nacimiento"));
```

Para unir dos objetos sin modificarlos se usa el *spread operator*:

```
const producto = {
    nombreProducto : "Monitor 20 Pulgadas",
    precio: 300,
    disponible: true
}

const medidas = {
    peso: '1kg',
    medida: '1m'
}

const nuevoProducto = { ...producto, ...medidas };

console.log(producto);
console.log(nuevoProducto);
```

Ver https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Working_with_Objects y https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Operators/Property_Accessors

11.1. Destructuring de objetos

La sintaxis de desestructuración es una expresión de JavaScript que permite desempacar valores de arreglos o propiedades de objetos en distintas variables.

```
  const gato = {
    nombre: "Valiente",
```

```
    duerme: true,
    edad: 10,
    enemigos: ["agua", "perros"],
    otros: {
        amigos: ["Cobarde", "Tímido", "Pegajoso"],
        favoritos: {
            comida: {
                fria: "salmón",
                caliente: "pollo",
            },
        },
    },
};

const { nombre, duerme, edad, enemigos } = gato;
console.log(nombre);
console.log(duerme);
console.log(edad);
console.log(enemigos);
```

La destructuración también sirve para Array, solo reemplazar por []

```
const enemigos = ["agua", "perros"]
const [agua, perro] = enemigos;
console.log(agua);
console.log(perro);
```

12. ARRAY

Un array (también llamado vector o arreglo) es una variable que contiene diversos valores. Lo creamos simplemente con "[]" o con "new Array()" o inicializando los elementos. Podemos referenciar los elementos de un array indicando su posición.

Los arrays, al igual que un String, poseen una propiedad llamada "length" que podemos utilizar para conocer el número de elementos del array. Existen tanto array unidimensionales como de varias dimensiones (los más usados son los de dos dimensiones, llamados arrays bidimensionales o matrices)

```
// Array definido 1 a 1
let miVector=[]; // let miVector=new Array(); es equivalente
miVector[0]=22;
miVector[1]=12;
miVector[2]=33;
//Array definido en una línea inicializando elementos
let otroArray=[1,2,"Cancamusa"]; // Valores dentro del array
```

```
console.log(miVector[1]);
console.log(otroArray[2]);
console.log(miVector + " "+miVector.length ); //  array y tamanyo
// Array bidimensional 5x4 declarado sin rellenar
let matrizBi = new Array(5).fill().map(() => new Array(4));
// Otro array bidimensional 3x5, relleno de ceros
let otraMatrizBi = [...Array(3)].map(() => Array(5).fill(0))
// Otro array bidimensional
let matriz = new Array();
matriz  [0] = [1, 2, 3, 4, 5, 6, 7, 8, 9];
matriz  [1] = [7, 8, 9, 1, 2, 3, 4, 5, 6];
matriz  [2] = [4, 5, 6, 7, 8, 9, 1, 2, 3];
matriz  [3] = [3, 1, 2, 6, 4, 5, 9, 7, 8];
matriz  [4] = [9, 7, 8, 3, 1, 2, 6, 4, 5];
matriz  [5] = [6, 4, 5, 9, 7, 8, 3, 1, 2];
matriz  [6] = [2, 3, 1, 5, 6, 4, 8, 9, 7];
matriz  [7] = [8, 9, 7, 2, 3, 1, 5, 6, 4];
matriz  [8] = [5, 6, 4, 8, 9, 7, 2, 3, 1];

Array.from(Array(2), () => new Array(4))
```

12.1. Clonando arrays (y objetos)

Los arrays en Javascript internamente almacenan referencias a donde está alojado cada objeto en memoria, por lo cual copiar un array no es tan simple como hacer miArray=miOtroArray.

En Javascript ES6, se puede hacer una copia sencilla de los valores de un array unidimensional mediante el *spread operator*:

```
let miArray= [ ...miOtroArray];
```

Esta copia solo funciona con arrays unidimensionales, ya que con multidimensionales, solo copiará las referencias de memoria de cada uno de estos.

Se pueden hacer copias completas con métodos como este, que se basa en convertir el array a JSON y volver a obtenerlo desde JSON:

```
let miArrayMultidimensional =
JSON.parse(JSON.stringify(miOtroArrayMultidimensional));
```

Más información en https://www.samanthaming.com/tidbits/70-3-ways-to-clone-objects/

12.2. Operaciones

Aclarar que en el interior de un array se pueden guardar cualquier tipo de objetos (date,

string, números enteros, decimales, objetos, otros arrays, etc...).

Todo lo aplicado al uso de arrays, es aplicable al objeto Array (length para número de elementos, modificar/consultar array, etc...).

Algunos de los métodos más importantes del objeto Array: join, split, push, pop, shift, reverse, sort, slice, filter, find, some, findIndex, concat, reduce, includes, ...

- **join([separador]):** devuelve una cadena con todos los elementos del array, separados por el texto que se incluya en separador.

```
const elements = ['Fire', 'Air', 'Water'];

console.log(elements.join());
// expected output: "Fire,Air,Water"

console.log(elements.join(''));
// expected output: "FireAirWater"

console.log(elements.join('-'));
// expected output: "Fire-Air-Water"
```

- **split():** divide un objeto de tipo String en un array, mediante un separador.

```
const cadenaMeses = "Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec";

const arrayMeses = cadenaMeses.split(",");
console.log(arrayMeses);// [
'Jan','Feb','Mar','Apr','May','Jun','Jul','Aug','Sep','Oct','Nov','Dec' ]
```

- **push(elemento1, elemento2,….):** añade al final los elementos 1 y 2 en el orden proporcionado.

```
const animals = ['pigs', 'goats', 'sheep'];

const count = animals.push('cows');
console.log(count);
// expected output: 4
console.log(animals);
// expected output: Array ["pigs", "goats", "sheep", "cows"]

animals.push('chickens', 'cats', 'dogs');
console.log(animals);
// expected output: Array ["pigs", "goats", "sheep", "cows", "chickens",
"cats", "dogs"]
```

- **pop():** devuelve y elimina el último elemento del array.

```
const plants = ['broccoli', 'cauliflower', 'cabbage', 'kale', 'tomato'];

console.log(plants.pop());
// expected output: "tomato"
```

```
console.log(plants);
// expected output: Array ["broccoli", "cauliflower", "cabbage", "kale"]

plants.pop();

console.log(plants);
// expected output: Array ["broccoli", "cauliflower", "cabbage"]
```

- **shift()**: elimina el primer elemento del array y lo retorna.

```
let miPescado = ['ángel', 'payaso', 'mandarín', 'cirujano'];

console.log('miPescado antes: ' + miPescado);
// "miPescado antes: ángel,payaso,mandarín,cirujano"

var eliminado = miPescado.shift();

console.log('miPescado después: ' + miPescado);
// "miPescado after: payaso,mandarín,cirujano"

console.log('Elemento eliminado: ' + eliminado);
// "Elemento eliminado: ángel"
```

- **reverse()**: invierte el orden de los elementos de un array.

```
const array1 = ['one', 'two', 'three'];
console.log('array1:', array1);
// expected output: "array1:" Array ["one", "two", "three"]

const reversed = array1.reverse();
console.log('reversed:', reversed);
// expected output: "reversed:" Array ["three", "two", "one"]

// Careful: reverse is destructive -- it changes the original array.
console.log('array1:', array1);
// expected output: "array1:" Array ["three", "two", "one"]
```

- **sort()**: ordena los elementos de un array alfabéticamente (valor Unicode). Nota: al ser un orden alfabético, puede dar resultados incorrectos con números, por ejemplo 10 ir antes que 2.

```
let frutas = ['guindas', 'manzanas', 'bananas'];
frutas.sort(); // ['bananas', 'guindas', 'manzanas']

let puntos = [1, 10, 2, 21];
puntos.sort(); // [1, 10, 2, 21]
// Tenga en cuenta que 10 viene antes que 2
// porque '10' viene antes que '2' según la posición del valor Unicode.
```

```
let cosas = ['word', 'Word', '1 Word', '2 Words'];
cosas.sort(); // ['1 Word', '2 Words', 'Word', 'word']
// En Unicode, los números vienen antes que las letras mayúsculas
// y estas vienen antes que las letras minúsculas.
```

Se puede proporcionar una función de comparación compareFunction(a,b) para ordenar según el valor que retorne dicha función. Por ejemplo, siendo a y b dos elementos comparados, entonces:

- Si compareFunction(a, b) es menor que 0, se sitúa a en un índice menor que b. Es decir, a viene primero.
- Si compareFunction(a, b) retorna 0, se deja a y b sin cambios entre ellos, pero ordenados con respecto a todos los elementos diferentes.
- Si compareFunction(a, b) es mayor que 0, se sitúa b en un índice menor que a.

```
function compareFunction(a, b) {
    if (a es menor que b según criterio de ordenamiento) {
      return -1;
    }
    if (a es mayor que b según criterio de ordenamiento) {
      return 1;
    }
    // a debe ser igual b
    return 0;
  }
```

De esta forma, para ordenar números en lugar de cadenas podemos hacerlo así:

```
let numbers = [4, 2, 5, 1, 3];
numbers.sort(function(a, b) { //numbers.sort((a, b) => a - b);
  return a - b;
});
console.log(numbers);// [1, 2, 3, 4, 5]
```

Además, los objetos pueden ser ordenados por el valor de alguna de sus propiedades:

```
var items = [
    { name: 'Edward', value: 21 },
    { name: 'Sharpe', value: 37 },
    { name: 'And', value: 45 },
    { name: 'The', value: -12 },
    { name: 'Magnetic', value: 13 },
    { name: 'Zeros', value: 37 }
  ];
  items.sort((a, b) => {
    if (a.name > b.name) {
      return 1;
    }
    if (a.name < b.name) {
```

```
      return -1;
    }
    // a must be equal to b
    return 0;
  });

  console.log(items);
  /*[ { name: 'And', value: 45 },
  { name: 'Edward', value: 21 },
  { name: 'Magnetic', value: 13 },
  { name: 'Sharpe', value: 37 },
  { name: 'The', value: -12 },
  { name: 'Zeros', value: 37 } ]*/
```

- **slice(inicio, [final]):** devuelve los elementos de un array comprendidos entre inicio y final (sin incluir el final). Si no se indica final, se toma hasta el último elemento del array. Si el índice especificado es negativo, indica un desplazamiento desde el final del array. slice(-2)extrae los dos últimos elementos del array

```
var nombres = ['Rita', 'Pedro', 'Miguel', 'Ana', 'Vanesa'];
var masculinos = nombres.slice(1, 3);// masculinos contiene
['Pedro','Miguel']
```

- **filter (funcionCondicion()):** crea un nuevo array con todos los elementos que cumplan la condición implementada en la función dada (puede servir tanto para seleccionar una serie de elementos como para no seleccionarlos)

```
const users = [
    { uid: 1, name: "John", age: 34 },
    { uid: 2, name: "Amy", age: 20 },
    { uid: 3, name: "camperCat", age: 10 },
];

const mayor = users.filter((user) => user.age > 30);
console.log(mayor);//[ { uid: 1, name: 'John', age: 34 } ]
```

- **find ():** devuelve el valor del primer elemento del array que cumple la función de prueba proporcionada.

```
const users = [
    { uid: 1, name: "John", age: 34 },
    { uid: 2, name: "Amy", age: 20 },
    { uid: 3, name: "camperCat", age: 10 },
];

const amy = users.find((user) => user.uid === 2);
console.log(amy);//{ uid: 2, name: 'Amy', age: 20 }
//con destructurazión
```

```
const { age } = users.find((user) => user.uid === 2);
console.log(age);//20
```

- **some()**: comprueba si al menos un elemento del array cumple con la condición implementada por la función proporcionada.

```
const users = [
  { uid: 1, name: "John", age: 34 },
  { uid: 2, name: "Amy", age: 20 },
  { uid: 3, name: "camperCat", age: 10 },
];

const existe = users.some((user) => user.uid === 2);
console.log(existe);//true
```

- **findIndex()**: devuelve el índice del primer elemento de un array que cumpla con la función de prueba proporcionada. En caso contrario devuelve -1.

```
const users = [
  { uid: 1, name: "John", age: 34 },
  { uid: 2, name: "Amy", age: 20 },
  { uid: 3, name: "camperCat", age: 10 },
];

const existe = users.findIndex((user) => user.uid === 4);
console.log(existe);//-1
```

- **concat()**: se usa para unir dos o más arrays. Este método no cambia los arrays existentes, sino que devuelve un nuevo array.

```
const array1 = ["a", "b", "c"];
const array2 = ["d", "e", "f"];
const array3 = array1.concat(array2);

console.log(array3);//[ 'a', 'b', 'c', 'd', 'e', 'f' ]
```

También se pueden concatenar con el operador spread (...)

```
const array1 = ["a", "b", "c"];
const array2 = ["d", "e", "f"];
const array3 = [...array1, ...array2];

console.log(array3);//[ 'a', 'b', 'c', 'd', 'e', 'f' ]
```

- **reduce (acumulador, valorActual[, índice[, array]])[, valorInicial])**: ejecuta una función reductora sobre cada elemento de un array, devolviendo como resultado un único valor.

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

const sumaTodos = numeros.reduce((acc, valorActual) => acc + valorActual);

console.log(sumaTodos);//15

const arrayNumeros = [
  [0, 1],
  [2, 3],
  [4, 5],
];

const soloNumeros = arrayNumeros.reduce(
  (acc, current) => acc.concat(current)
);

console.log(soloNumeros);//[ 0, 1, 2, 3, 4, 5 ]
```

- **includes()**: determina si una matriz incluye un determinado elemento, devuelve true o false según corresponda

```
const array1 = [1, 2, 3];

console.log(array1.includes(2));
// expected output: true

const pets = ['cat', 'dog', 'bat'];

console.log(pets.includes('cat'));
// expected output: true

console.log(pets.includes('at'));
// expected output: false
```

Puedes ver todas las operaciones aquí: https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array y más información aquí https://bluuweb.github.io/desarrollo-web-bluuweb y aquí https://lenguajejs.com/javascript/caracteristicas/array-functions/

Más ejemplos, aquí: https://www.youtube.com/watch?v=R8rmfD9Y5-c y en https://www.youtube.com/watch?v=S1ZXSoAxEBg

12.3. ForEach, Map y for...of

Sirven para iterar sobre los elementos de un array.

Ejemplos:

```
let vector = [1,2,"hola", "esto es un vector",-1,2.4];

// ForEach
/**
vector.forEach(function(item){
    console.log(item)
});

 */

vector.forEach( item => console.log(item));

vector.forEach( (item,indice) => console.log(`${item} en posición ${indice}`
));

/*
let vector2= vector.map (function (item){
    return item;
});
*/

// map
let vector2 = vector.map( item => `${item}`);

console.table(vector2);

// map
const users = [
    { name: "John", age: 34 },
    { name: "Amy", age: 20 },
    { name: "camperCat", age: 10 },
];

const names = users.map((user) => user.name);
console.log(names);//[ 'John', 'Amy', 'camperCat' ]

// map
const numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
const numerosPorDos = numeros.map((item) => item * 2);
console.log(numerosPorDos);//[ 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 ]
```

Cuando quieras mostrar los elementos usa forEach, si quieres crear un nuevo vector para modificarlo, usa map.

Puedes ver más aquí: https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array/f

orEach

También está la posibilidad de usar **for…of** para ejecutar un bloque de código para cada elemento de un objeto iterable (String, Array, …)

Ejemplo:

```
let iterable = [10, 20, 30];
for (let value of iterable) {
  value += 1;
  console.log(value);
}
// 11
// 21
// 31
```

Ver https://www.youtube.com/watch?v=tP8JiVUiyDo sobre map, filter y reduce (a partir de 6:50).

12.4. Evitar el uso de bucles usando la función map

En ciertos problemas, puede ser útil, para mejorar la legibilidad y mantenibilidad del código (ver: https://www.youtube.com/watch?v=LALUQNpm4zk)

13. OPERADOR ...

El operador ... (punto, punto, punto) es conocido como operador **spread** (propagación en castellano), o como el parámetro **rest**, dependiendo de dónde sea usado.

Spread.

- El operador de propagación divide un objeto o un array en múltiples elementos individuales. Esto permite expandir expresiones en situaciones donde se esperan múltiples valores como en llamadas a funciones o en array literales.
- Este operador nos permite clonar objetos y arrays de una forma muy simple y expresiva. También podríamos utilizarlo para concatenar arrays de manera concisa.

```
//Operador Spread
const numbers = [1, 2, 3];
const myArray = ['a', 'b', 'c', ...numbers]; //["a", "b", "c", 1, 2, 3]

//Permite clonar objetos y arrays de una forma muy simple y expresiva:
const post = {title: "Operador spread", content:"lorem ipsum…"}
//clonado con Object.assign();
const postCloned =  Object.assign({}, book); //👍
//clonado con el spread operator
const postCloned = { ...book };//👌

const myArray = [1, 2, 3];
//clonado con slice()
const myArrayCloned = myArray.slice();//👍
//clonado con el spread operator()
const myArrayCloned = [ ...myArray ];//👌

//También podemos usar el operador spread para concatenar arrays:
const arrayOne = [1, 2, 3];
const arrayTwo = [4, 5, 6];

//concatenate with concat()
const myArray = arrayOne.concat(arrayTwo); // 👍 [1, 2, 3, 4, 5, 6]

//concatenate with spread operator
const myArray = [...arrayOne, ...ArrayTwo]; //👌 [1, 2, 3, 4, 5, 6]
```

Rest

- El parámetro rest unifica los argumentos restantes en la llamada de una función cuando el número de argumentos excede el número de parámetros declarados en la misma.
- En cierta manera, el rest actúa de forma contraria a spread. Mientras que spread 'expande' los elementos de un array u objeto dado, rest unifica un conjunto de elementos en un array.

```
function add(x, y) {
    return x + y;
}

add(1, 2, 3, 4, 5) //3

function add(...args){
    return args.reduce((previous, current)=> previous + current, 0)
}

add(1, 2, 3, 4, 5) //15

//rest es el último parámetro de la función y es de tipo array:
function process(x, y, ...args){
    console.log(args)
}

process(1, 2, 3, 4, 5);//[3, 4, 5]
```

14. DATE

Date es un objeto predefinido que nos permite trabajar con fechas. Para crear un objeto date se admiten múltiples formatos.

Los meses en Date se numeran del 0 al 11 (siendo el 0 Enero y el 11 Diciembre) mientras que los días si se numeran del 1 al 31.

Ejemplo:

```
// Crea una fecha con la fecha y hora del sistema
let d=new Date();
// Crea una fecha basándose en la cadena de fecha especificada
d=new Date("October 13, 2014 11:13:00");
// Crea una fecha indicando año, mes, dia, horas, minutos, segundos
milisegundos
d=new Date(99,5,24,11,33,30,0);
// Crea una fecha indicando año, mes, día.
d=new Date(99,5,24);
// Crea una fecha indicando dia, mes y año.
d=new Date("12/06/1980");
```

Algunos de los métodos más importantes del objeto Date:

- **Métodos set/get**: son métodos que permite cambiar el valor de alguna parte de la fecha (set) u de obtener el valor de alguna parte de la fecha (get).
  - Ejemplos: setMonth(mes), getMonth(); setDate(dia), getDate(), setHours(hora, minuto,segundo), getHours(), etc.
- **getDay()**: devuelve el día de la semana, (el día del mes es con getDate). Están numerados del 0 al 6, siendo el 0 domingo y el 6 sábado.
- **toDateString()**: convierte la fecha del objeto a una cadena en objeto fecha.
- **toGMTString()**: convierte la fecha del objeto en una cadena con formato de fecha GMT.
- **toUTCString()**: convierte la fecha del objeto en una cadena con formato de fecha UTC.

Ejemplo:

// Con fecha actual y con día uno del mes, mostramos día de la semana

```
let d=new Date();
alert(d.getDay());
d.setDate(1);
alert(d.getDay);
```

Puedes encontrar más información aquí: Ver https://lenguajejs.com/javascript/fechas/date-fechas-nativas/

15. MATH

El objeto Math es un objeto que contiene una colección métodos que nos ayudarán a trabajar realizar operaciones aritméticas, redondeos, etc.

Algunas de las constantes matemáticas predefinidas más importantes son:

- **E**: almacena el número de Euler.
- **PI**: almacena el número Pi.
- **LN2 / LN10**: logaritmo neperiano de 2 / 10.
- **LOG2E / LOG10E**: logaritmo en base 2 / 10 del número de Euler.

Algunos de los métodos más importantes que tiene disponibles son:

- **Redondeo**:
  - **floor(numero)**: redondea hacia abajo un número.
  - **ceil(numero)**: redondea hacia arriba un número.
  - **round(numero)**: redondea al entero más cercano.
- **Operaciones matemáticas:**
  - **abs(numero)**: devuelve el número en valor absoluto.
  - **max / min (x,y)**: devuelve el mayor / menor de dos números x e y.
  - **pow(x,y)**: devuelve x elevado a y.
  - **random()**: devuelve un número aleatorio con decimales mayor o igual que

0 y menor (nunca igual) que 1.

- **sqrt(numero)**: devuelve la raíz cuadrada del número indicado.

Ejemplo:

```
// Obtenemos un entero entre 1 y 11
let x=parseInt( (Math.random()*10)+1 )
// Redondeamos y hacia abajo
let y=Math.floor(11.5);
alert(Math.PI);

// Objeto Math

let resultado;

resultado = Math.PI;
resultado = Math.round(2.5);
resultado = Math.ceil(2.1); // Ceil siempre redondea hacia arriba
resultado = Math.floor(2.9); // Floor siempre redondea hacia abajo
resultado = Math.sqrt(144);
resultado = Math.abs(-200);
resultado = Math.min( 3, 5, 1, 8 , 2, 10 );
resultado = Math.max( 3, 5, 1, 8 , 2, 10 );
resultado = Math.random();
resultado = Math.floor( Math.random() * 30 );

console.log(resultado);
```

15.1. Números aleatorios

Es una de las funciones de Math, más utilizadas, veamos algunos ejemplos de uso:

Generar un número aleatorio entre 0 y 10

```
let x = Math.random()*10;
console.log(x);
// 4.133793901445541
```

Generar un número aleatorio dentro de un rango

```
const MIN = 83.1;
const MAX = 193.36;

var x = Math.random()*(MAX - MIN)+MIN;

console.log(x);
// 126.94014012699063
```

Generar un número aleatorio entero dentro de un rango

```
const MIN = 1718;
const MAX = 3429;

let x = Math.floor(Math.random()*(MAX-MIN+1)+MIN);

console.log(x);
```

16. TEMPORIZADORES

Algunos de los métodos más importantes relacionados con los temporizadores son:

- **setTimeout(cadenaFuncion,tiempo):** este método ejecuta la llamada a la función proporcionada por la cadena (se puede construir una cadena que lleve parámetros) y la ejecuta pasados los milisegundos que hay en la variable tiempo. Devuelve un identificador del "setTimeout" que nos servirá para referenciar lo si deseamos cancelar. SetTimeout solo ejecuta la orden una vez.
- **setInterval(cadenaFunción, tiempo):** exactamente igual que setTimeout, con la salvedad de que no se ejecuta una vez, sino que se repite cíclicamente cada vez que pasa el tiempo proporcionado.
- **clearTimeout / clearInterval (id):** se le pasa el identificador del timeout/interval y lo anula.

Ejemplo:

```
// Creamos un intervalo que cada 15 segundos muestra mensaje hola
let idA=setInterval("alert('hola');",15000);
// Creamos un timeout que cuando pasen 3 segundos muestra mensaje adios
let idB=setTimeout("alert('adios');",3000);
// Creamos un timeout que cuando pasen 5 segundos muestra mensaje estonoseve
let idC=setTimeout("alert('estonoseve');",5000);
// Cancelamos el último timeout
clearTimeout(idC);
// Reloj
let temporizador = setInterval(() => {
      console.log(new Date().toLocaleTimeString());
    }, 1000);
```

17. CLASES EN JAVASCRIPT

Javascript ES6 permite una mejor definición de clases que en anteriores versiones de Javascript.

Mediante la palabra reservada "class", permite definir clases, métodos, atributos, etc…

Recordad que los objetos en Javascript se guardan como referencias de memoria.

Ejemplo:

```
class Punto {
    // Constructor de la clase
    constructor ( pX , pY ){
      this.pX = pX;
      this.pY = pY;
    }
    // Método estático para calcular distancia entre dos puntos (recuerda
que los métodos estáticos son llamados sin instanciar su clase. Son
habitualmente utilizados para crear funciones para una aplicación)

    static distancia ( a , b) {
      const dx = a.pX - b.pX;
      const dy = a.pY - b.pY;

      return Math.sqrt ( dx * dx + dy * dy );
    }
   // Método indicado para ser usado como getter
    get coordX() {
       return this.pX;
     }
    // Método normal
    devuelveXporY () {
      return this.pX * this.pY;
    }
  }
  let p1 = new Punto(5, 5);
  let p2 = new Punto(10, 10);
  //Llamada método estático
  console.log (Punto.distancia(p1, p2));
  //Llamada método normal
  console.log (p1.devuelveXporY());
  // Al ser un getter, puede usarse como una propiedad
  console.log (p1.coordX);
```

Otros ejemplos:

```
class Animal {
  //el constructor es un método especial que se ejecuta en el momento de
instanciar la clase
  constructor(nombre, genero) {
    this.nombre = nombre;
    this.genero = genero;
  }
  //Métodos
  sonar() {
    console.log("Hago sonidos por que estoy vivo");
```

```
  }
  saludar() {
    console.log(`Hola me llamo ${this.nombre}`);
  }
}
//Herencia
class Perro extends Animal {
  constructor(nombre, genero, tamanio) {
    //con el método super() se manda a llamar el constructor de la clase
padre
    super(nombre, genero);
    this.tamanio = tamanio;
    this.raza = null;
  }
  sonar() {
    console.log("Soy un perro y mi sonido es un ladrido");
  }
  ladrar() {
    console.log("Guauuu Guauuu!!!");
  }
  //un método estático se pueden ejecutar sin necesidad de instanciar la
clase
  static queEres() {
    console.log(
      "Los perros somos animales mamíferos que pertenecemos a la familia de
los caninos. Somos considerados los mejores amigos del hombre."
    );
  }
  //Los setters y getters son métodos especiales que nos permiten establecer
y obtener los valores de atributos de nuestra clase
  get raza() {
    return this._raza;//hay que poner el _ para que funcione
  }
  set raza(raza) {
    this._raza = raza;//hay que poner el _ para que funcione //It is
convention to precede the name of a private variable with an underscore
  }
}
Perro.queEres();
const mimi = new Animal("Mimi", "Hembra"),
  scooby = new Perro("Scooby", "Macho", "Gigante");
console.log(mimi);
mimi.saludar();
mimi.sonar();
console.log(scooby);
scooby.saludar();
scooby.sonar();
```

```
scooby.ladrar();
console.log(scooby.raza);
scooby.raza = "Grán Danés";
console.log(scooby.raza);
```

Observa que:

- Los **constructores** nunca utilizan return.
- Todas las propiedades y métodos son, por defecto, públicas. Para que sean **privados** se usa el carácter **#** justo antes del nombre de la propiedad o nombre.

```
class Animal {
  #miSecreto = "Me gusta Internet Explorer";

  #decirSecreto() {
    return this.#miSecreto;
  }

  decirSacrilegio() {
    return this.#decirSecreto();
  }
}
const patitoFeo = new Animal();
patitoFeo.#decirSecreto();     // Error
patitoFeo.decirSacrilegio();
```

- Tenemos **métodos estáticos** (no es necesario instanciar la clase para usarlo).
- Fíjate en **get** y **set**, técnicamente, el código externo todavía puede acceder al nombre directamente usando “perro.raza". Pero hay un acuerdo ampliamente conocido de que las propiedades que comienzan con un guión bajo “_” son internas y no deben ser manipuladas desde el exterior del objeto.

Para saber más: https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Classes o aquí https://lenguajejs.com/javascript/caracteristicas/clases-es6/

18. OTROS TIPOS DE DATOS

- Symbol es un tipo de datos cuyos valores son únicos e immutables. Dichos valores pueden ser utilizados como identificadores (claves) de las propiedades de los objetos. Cada valor del tipo Symbol tiene asociado un valor del tipo String o Undefined que sirve únicamente como descripción del símbolo.
- El objeto Set permite almacenar valores únicos de cualquier tipo, incluso valores primitivos u referencias a objetos.

```
const set = new Set([1, 2, 3, 3, 4, 5, true, false, false, {}, {}, "hola", "HOla"]);
console.log(set);  Set { 1, 2, 3, 4, 5, true, false, {}, {}, 'hola', 'HOla' }
console.log(set.size);  11
const set2 = new Set();
set2.add(1);
set2.add(2);
set2.add(2);
set2.add(3);
set2.add(true);
set2.add(false);
set2.add(true);
set2.add({});
console.log(set2);  Set { 1, 2, 3, true, false, {} }
console.log(set2.size);  6
console.log("Recorriendo set");  Recorriendo set
for (item of set) {
  console.log(item);  1, 2, 3, 4, 5, true, false, {}, {}, hola, HOla
}
console.log("Recorriendo set2");  Recorriendo set2
set2.forEach(item => console.log((item)));
let arr = Array.from(set);
console.log(arr);  [ 1, 2, 3, 4, 5, true, false, {}, {}, 'hola', 'HOla' ]
console.log(arr[0]);  1
console.log(arr[9]);  hola
set.delete("HOla");
console.log(set);  Set { 1, 2, 3, 4, 5, true, false, {}, {}, 'hola' }
console.log(set.has("hola"));  true
console.log(set.has(19));  false
set2.clear();
console.log(set2);  Set { }
```

- Map es, al igual que Objet, una colección de datos identificados por claves. Pero la principal diferencia es que Map permite claves de cualquier tipo.
  Los métodos y propiedades son:
  - new Map() – crea el mapa.
  - map.set(clave, valor) – almacena el valor asociado a la clave.
  - map.get(clave) – devuelve el valor de la clave. Será undefined si la clave no existe en map.
  - map.has(clave) – devuelve true si la clave existe en map, false si no existe.
  - map.delete(clave) – elimina el valor de la clave.
  - map.clear() – elimina todo de map.
  - map.size – tamaño, devuelve la cantidad actual de elementos.

```
const mapa = new Map();
mapa.set("nombre", "Jon");
mapa.set("apellido", "MirCha");
mapa.set("edad", 35);
console.log(mapa);  Map { 'nombre' => 'Jon', 'apellido' => 'MirCha', 'edad' => 35 }
console.log(mapa.size);  3
console.log(mapa.has("correo"));  false
console.log(mapa.has("nombre"));  true
console.log(mapa.get("nombre"));  Jon
mapa.set("nombre", "Jonathan MirCha");
console.log(mapa.get("nombre"));  Jonathan MirCha
mapa.delete("apellido");
mapa.set(19, "diecinueve");
mapa.set(false, "falso");
mapa.set({}, {});
console.log(mapa);  Map { 'nombre' => 'Jonathan MirCha', 'edad' => 35, 19 => 'diecinueve', false => 'falso', {} => {} }
for (let [key, value] of mapa) {
  console.log(`Llave: ${key}, Valor:${value}`);  ... Llave: edad, Valor:35, Llave: 19, Valor:diecinueve, Llave: false, Valor:falso,
}
const mapa2 = new Map([
  ["nombre", "kEnAi"],
  ["edad", 7],
  ["animal", "perro"],
  [null, "nulo"],
]);
console.log(mapa2);  Map { 'nombre' => 'kEnAi', 'edad' => 7, 'animal' => 'perro', null => 'nulo' }
const llavesMapa2 = [...mapa2.keys()];
const valoresMapa2 = [...mapa2.values()];
console.log(llavesMapa2);  [ 'nombre', 'edad', 'animal', null ]
console.log(valoresMapa2);  [ 'kEnAi', 7, 'perro', 'nulo' ]
```

- Iteradores y generadores: https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Iterators_and_Generators

19. BIBLIOGRAFÍA

[1] Javascript Mozilla Developer: https://developer.mozilla.org/es/docs/Web/JavaScript

[2] Curso HTML, CSS y Bootstrap 5: https://bluuweb.github.io/desarrollo-web-bluuweb/11-02-js-basico/

[3] Clean Javascript, Miguel A. Gómez, Software Crafters Ed. 2020.

[4] Nuevas características de Javascript ES6: https://github.com/lukehoban/es6features

[5] Expresiones regulares
https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Regular_Expressions

20. ANEXOS

20.1. Depurar una web/aplicación desde la consola del navegador

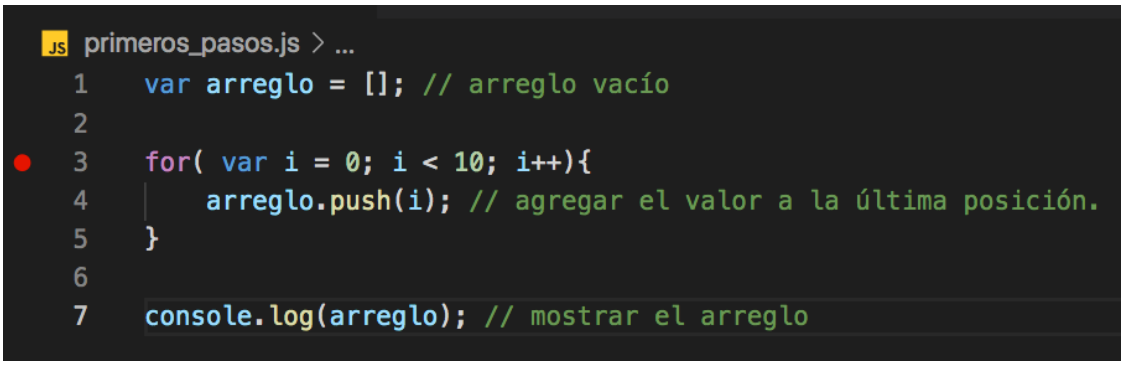
Las herramientas para desarrolladores de Google Chrome[4] o Mozilla Firefox[5] (F12) permiten **depurar nuestro código** para encontrar errores o comprobar cómo se comporta nuestro código.

Es de vital importancia no solo que la conozcas, sino que la utilices. De hecho, antes de llamar a tu profesor porque tienes un error o algo no te funciona debes depurar tu aplicación.

Ver el curso *ChromeDevToolsParaDesarrolladores* en esta misma unidad para saber cómo utilizarlas.

20.2. Depurar una web/aplicación desde VSCode

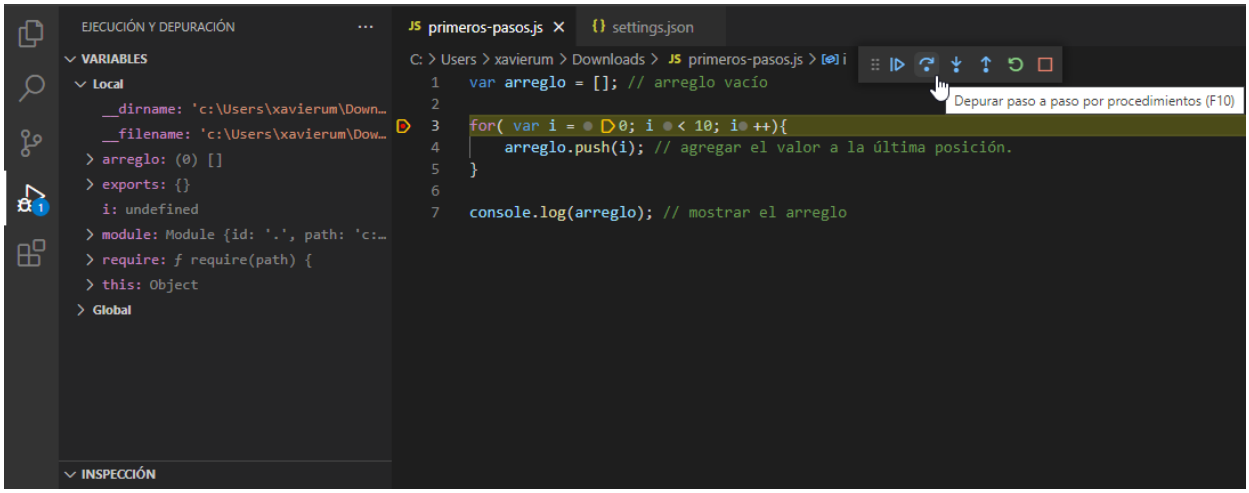
También puede ser interesante depurar tu aplicación directamente desde el entorno de trabajo, para ello, lo primero es **colocar un *breakpoint*** o punto de interrupción (recuerda que es la línea del código donde se va a detener la ejecución de tu aplicación), pulsando con el ratón a la izquierda del número de línea.



En segundo lugar, hay que hacer click en el símbolo de *ejecución y depuración* y selecciona *ejecutar y depurar* para abrir un menú y seleccionar el entorno de ejecución

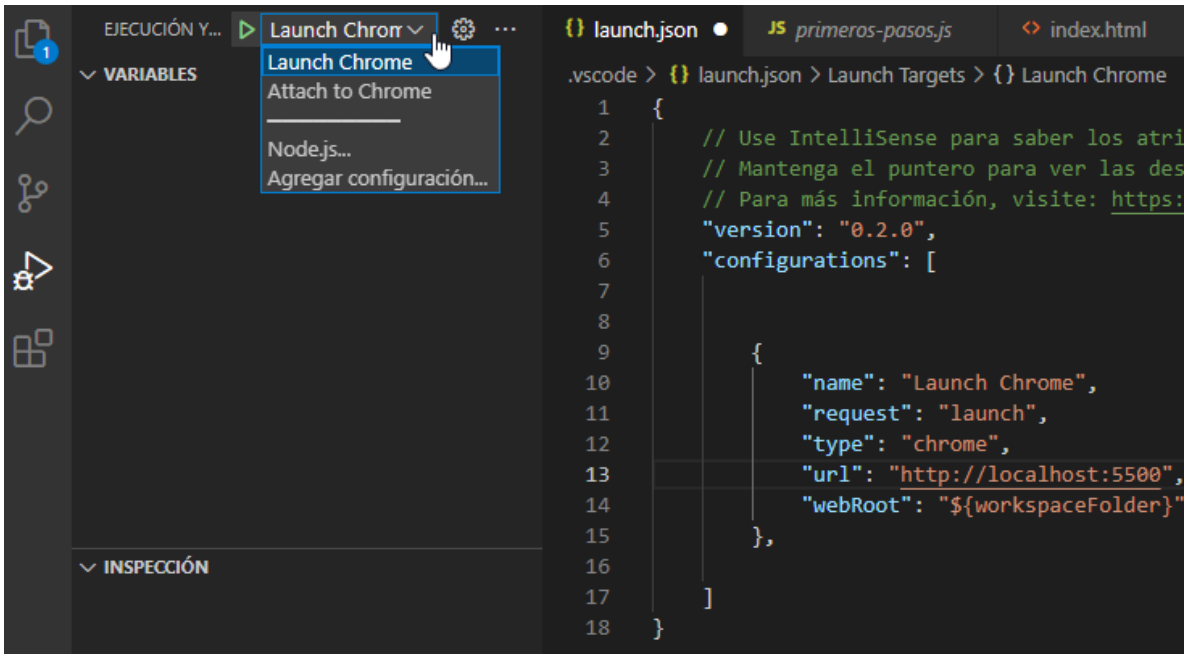
[4] https://developer.chrome.com/docs/devtools/
[5] https://firefox-source-docs.mozilla.org/devtools-user/web_console/index.html

Selecciona, por ejemplo, Node.js y a depurar.



Desde la opción del menú Ejecutar de VSCode se pueden modificar algunas configuraciones del depurador.

Se puede añadir una configuración para poder depurar desde VSCode pero mostrando los resultados en Chrome o Edge. Para ello creamos un index.html desde el que llamamos a nuestro .js. Después ejecutamos el .html desde Live Server (por ejemplo), como este lo abre por defecto en el puerto 5500 de localhost, en el fichero de configuración para la depuración hay que ponerle el mismo puerto a la opción de *Launch Chrome* o *Launch Edge*.



20.3. Quokka vs console.log

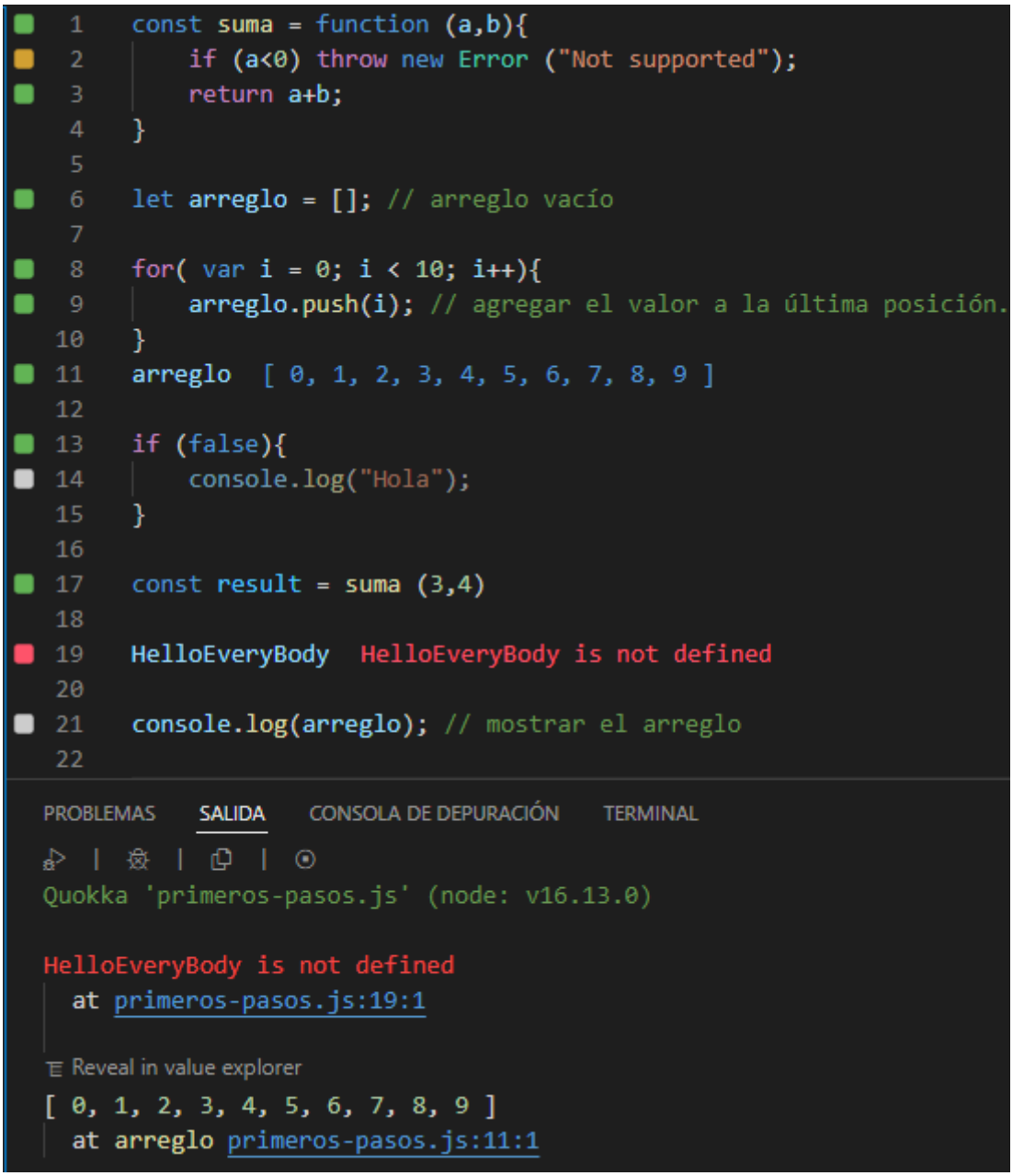
Ya hemos comentado con anterioridad que no es aconsejable el uso de console.log para depurar, aun así, lo encontraras en multitud de manuales, tutoriales y cursos.

Sin embargo, existen otras herramientas menos invasivas y que nos permiten evaluar nuestro código.

Quokka.js[6] es una extensión que nos permite, a la vez que escribimos código, evaluarlo. Para poder usarlo en VSCode pulsamos F1 y seleccionamos *Quokka.js: Start on Current File*. Como se puede observar, funciona a nivel de fichero.

- Los cuadrados de la izquierda indican el flujo de ejecución:
    - Los cuadrados grises significan que la línea de origen no se ha ejecutado.
    - Los cuadrados verdes significan que la línea de origen se ha ejecutado.
    - Los cuadrados amarillos significan que la línea de código fuente sólo se ha ejecutado parcialmente (código condicional).
    - Los cuadros rojos significan que la línea fuente es el origen de un error, o está en la pila de un error.

[6] https://quokkajs.com/



- Los valores se pueden ver en un azul oscuro.

Wallaby.js nos permite usar Quokka en proyectos grandes con Angular o React, por ejemplo y además nos ofrece algunas herramientas de testing. Funciona a nivel de proyecto.

Ver https://www.youtube.com/watch?v=SPA1IJ18ZFQ (hay algunas funcionalidades que actualmente son PRO)

20.4. Patrones de diseño

1. 4 principios de la POO: https://www.youtube.com/watch?v=tTPeP5dVuA4
2. Principios SOLID, ¡explicados!: https://www.youtube.com/watch?v=2X50sKeBAcQ
3. Patrones de Diseño, introducción: https://www.youtube.com/watch?v=3qTmBcxGlWk&list=PLJkcleqxxobUJlz1Cm8WYd-F_kckkDvc8
    a. Factory: https://www.youtube.com/watch?v=lLvYAzXO7Ek&list=PLJkcleqxxobUJlz1Cm8WYd-F_kckkDvc8&index=2
    b. Abstract factory: https://www.youtube.com/watch?v=CVlpjFJN17U&list=PLJkcleqxxobUJlz1Cm8WYd-F_kckkDvc8&index=3
    c. Strategy: https://www.youtube.com/watch?v=VQ8V0ym2JSo&list=PLJkcleqxxobUJlz1Cm8WYd-F_kckkDvc8&index=4
    d. Singleton: https://www.youtube.com/watch?v=GGq6s7xhHzY&list=PLJkcleqxxobUJlz1Cm8WYd-F_kckkDvc8&index=5
    e. Decorator: https://www.youtube.com/watch?v=nLy4x_LPPWU&list=PLJkcleqxxobUJlz1Cm8WYd-F_kckkDvc8&index=6
        i. Patrón decorator con código: https://www.youtube.com/watch?v=Ab9HxiPLryg&list=PLJkcleqxxobUJlz1Cm8WYd-F_kckkDvc8&index=7
    f. Observer: https://www.youtube.com/watch?v=HFkZb1g8faA&list=PLJkcleqxxobUJlz1Cm8WYd-F_kckkDvc8&index=8
    g. Composite: https://www.youtube.com/watch?v=ES3DnAPted0&list=PLJkcleqxxobUJlz1Cm8WYd-F_kckkDvc8&index=9
    h. Bridge: https://www.youtube.com/watch?v=6bIHhzqMdgg&list=PLJkcleqxxobUJlz1Cm8WYd-F_kckkDvc8&index=10

20.5. Utilidades de JS

https://1loc.dev/

20.6. Estructuras de datos, algoritmos y proyectos con hash o diccionario para resolver problemas específicos (ver bettatech)

- 6 estructuras de datos que deberías conocer: https://www.youtube.com/watch?v=5k2DWMRTXMM
- 5 algoritmos que deberías conocer: https://www.youtube.com/watch?v=eOow74IMTpc
- Uso de hash (diccionario): https://www.youtube.com/watch?v=HoT3y-L2t20
- Uso de pilas: https://www.youtube.com/watch?v=jIlB1D8e4rk
- Build 15 Javascript Projects – Vanilla Javascript Course: https://www.youtube.com/watch?v=3PHXvlpOkf4
- 10 Javascript projects in 10 hours: https://www.youtube.com/watch?v=dtKciwk_si4
- Learn Javascript by building 7 games: https://www.youtube.com/watch?v=ec8vSKJuZTk
- 3 Types of Projects That Will Make You a Programmer: https://www.youtube.com/watch?v=RYE0QQKJI9o
- 10 proyectos completos en Javascript: https://www.youtube.com/watch?v=YfaiDc585Eo

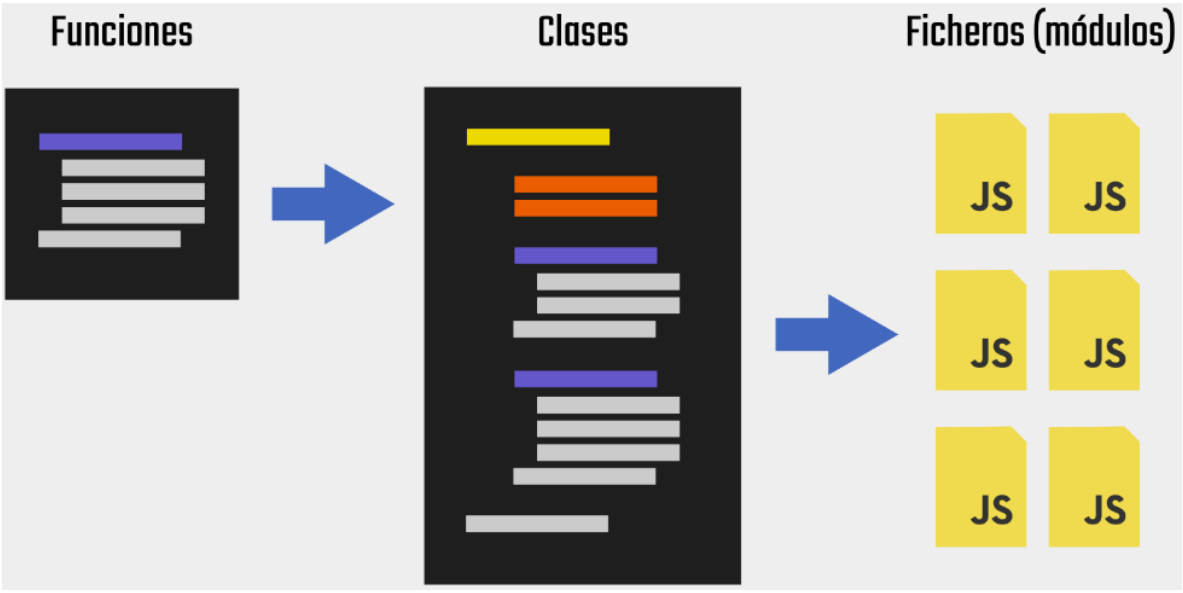
20.7. Introducción a la manipulación del DOM

Revisar el apartado DOM (querySelector, textContent, …) de UT2.

20.8. Módulos

Uno de los principales problemas que ha ido arrastrando Javascript desde sus inicios es la dificultad de organizar de una forma adecuada una aplicación grande, con muchas líneas de código. En muchos lenguajes, cuando un programa crece, se comienza a estructurar en funciones y, posteriormente, en clases. De esta forma organizamos de forma más lógica el código de nuestro programa.

Pero tener todo el código en un sólo fichero Javascript se vuelve confuso y complejo en cuanto el código crece. En la mayoría de los lenguajes de programación, el código se divide en ficheros diferentes de modo que, por ejemplo, cada clase está localizada en un fichero separado. De esta forma, todo queda mucho más organizado e intuitivo, y es fácil de localizar, aunque crezca.



En un principio, y de forma nativa, la forma más extendida era incluir varias etiquetas <script> desde nuestra página HTML. De esta forma, podíamos tener varios ficheros Javascript separados, cada uno para una finalidad concreta. Sin embargo, este sistema termina siendo muy poco modular, ofrecía algunas desventajas y resultaba lento, ya que sobrecargaba al cliente con múltiples peticiones extra (desde el cliente al servidor).

Con el tiempo, se desarrollaron sistemas no oficiales que permitían separar en varios archivos, como **CommonJS** (utilizado en NodeJS) o RequireJS (AMD), cada uno con sus particularidades, virtudes y desventajas.

En ECMAScript ES2015 o ES6, se introduce una característica nativa denominada **Módulos ES** o **ESM**, que permite la importación y exportación de datos entre diferentes ficheros Javascript, eliminando las desventajas que teníamos hasta ahora y permitiendo trabajar de forma más flexible en nuestro código Javascript.

Para trabajar con módulos tenemos a nuestra disposición las siguientes palabras clave:

Declaración	Descripción
`export`	Exporta datos (variables, funciones, clases...) del fichero actual hacia otros que los importen.
`import`	Importa datos (variables, funciones, clases...) desde otro fichero `.js` al actual.

Exportación de módulos

Mediante la palabra clave **export** crearemos lo que se llama un módulo de exportación que contiene datos. Estos datos puede ser variables, funciones, clases u objetos más complejos (a partir de ahora, elementos). Si dicho módulo ya existe, podremos ir añadiendo más propiedades. Por otro lado, con la palabra clave **import** podremos leer dichos módulos desde otros ficheros y utilizar sus elementos en nuestro código.

Un ejemplo sencillo para ver el funcionamiento de import y export en su modo más básico:

```
// Fichero constants.js
```

```
export const magicNumber = 42;

// Fichero index.js
import { magicNumber } from "./constants.js";

console.log(magicNumber);   // 42
```

Por defecto, un fichero Javascript no tiene **módulo de exportación** si no se usa un **export** al menos una vez en su código. Existen varias formas de exportar datos mediante la palabra clave de Javascript export:

Forma	Descripción
`export ...`	Declara un elemento o dato, añadiéndolo además al módulo de exportación.
`export { name }`	Añade el elemento `name` al módulo de exportación.
`export { n1, n2, n3... }`	Añade los elementos indicados (`n1` , `n2` , `n3` ...) al módulo de exportación.
`export * from './file.js'`	Añade todos los elementos del módulo de `file.js` al módulo de exportación.
`export default ...`	Declara un elemento y lo añade como módulo de exportación **por defecto**.

Además, como veremos a continuación, es posible renombrar los elementos exportados utilizando as seguido del nuevo nombre a utilizar. Por otro lado, si se realiza un export default, ese elemento será la exportación por defecto. Sólo puede haber una exportación por defecto por fichero.

Existen varias formas de exportar elementos. La más habitual, quizás, es la de simplemente añadir la palabra clave export a la izquierda de la declaración del elemento Javascript que deseamos exportar, ya sea una variable, una constante, una función, una clase u otro objeto más complejo:

```
export const number = 42;           // Se añade la constante number al
módulo
export const f1 = () => 99;         // Se añade la función f1 al módulo
export default f2 = () => "Manz";   // Se añade la función f2 como
exportación por defecto
```

Ten en cuenta que en el caso de utilizar una exportación por defecto en una declaración, no es posible utilizar var , let o const . Tampoco es posible usar export dentro de funciones, bucles o contextos específicos.

Por otro lado, si provienes de **NodeJS**, es muy probable que te resulte más intuitivo exportar módulos al final del fichero, ya que es como se ha hecho desde siempre en Node con los module.exports. Con export también podemos hacerlo de esta forma:

```
let number = 4;
const saludar = () => "¡Hola!";
const goodbye = () => "¡Adiós!";
class Clase {}
```

```
export { number };                          // Se crea un módulo y se añade
number
export { saludar, goodbye as despedir }; // Se añade saludar y despedir al
módulo
export { Clase as default };              // Se añade Clase al módulo
(default)
export { saludar as otroNombre };         // Se añade otroNombre al módulo

// En este punto, nuestro módulo exporta number, saludar, despedir, default
y otroNombre.
```

Importación de módulos

La palabra clave **import** es la opuesta de export. Si con export podemos exportar datos o elementos de un fichero, con import podemos cargar un módulo de exportación desde otro fichero Javascript, para utilizar dichos elementos en nuestro código.

Existen varias formas de importar código, utilizando esta palabra clave import:

Forma	Descripción
`import nombre from './file.js'`	Importa el elemento **por defecto** de `file.js` en `nombre`.
`import { nombre } from './file.js'`	Importa el elemento `nombre` de `file.js`.
`import { n1, n2... } from './file.js'`	Importa los elementos indicados desde `file.js`.
`import * as name from './file.js'`	Importa todos los elementos de `file.js` en el objeto `name`.
`import './file.js'`	No importa elementos, sólo ejecuta el código de `file.js`.

Recuerda, al igual que hacíamos en la exportación, también puedes renombrar elementos con import utilizando as seguido del nuevo nombre.

En el primer caso, realizamos una importación por defecto, donde importamos el elemento desde el módulo file.js y lo guardamos en nombre. En el segundo y tercer caso, realizamos una importación nombrada, donde importamos los elementos con el nombre indicado en el interior de los corchetes, desde el módulo file.js.

Las **importaciones por defecto** suelen estar ligeramente mal vistas por algunos desarrolladores. Una exportación nombrada suele ser más intuitiva y predecible a la hora de utilizar en nuestro código.

En el cuarto caso, importamos todos los elementos del módulo externo file.js en un objeto de nombre obj (es obligatorio indicar el nombre en este caso) y en el quinto y último caso, no importamos elementos, simplemente leemos el código del módulo y lo ejecutamos.

Los **bare imports** (o imports desnudos) son aquellos que se realizan indicando en from un string que no comienza por . o /, sino directamente por el nombre de una carpeta o paquete:

```
import { Howler, Howl } from "howler";  // Bare import
```

En este ejemplo, en lugar de utilizar ./howler.js (o una ruta similar), se indica el string howler. Esta característica no es estándar ni está soportada directamente por los navegadores, sino que es un invento que popularizó NodeJS a través de NPM. Generalmente, esto significa que se buscará un paquete con ese nombre en la carpeta node_modules, y que probablemente se estará utilizando algún bundler o automatizador como Webpack o similar.

Convenciones de módulos ES

- Si queremos utilizar import y export desde el navegador directamente, deberemos añadir los archivos .js que contienen los módulos con la etiqueta <script> utilizando el atributo type="module". En esta modalidad, los archivos Javascript se cargan en diferido, es decir, al final, como lo hace un <script defer>:

```
<script type="module" src="file.js"></script>
```

- Por norma general, a los archivos Javascript con módulos se les pone la extensión .js, aunque también se pueden encontrar con otras extensiones como .es2015 o .mjs (menos extendidas).
- Se aconseja utilizar las rutas UNIX en los export e import , ya que son las que tienen mejor soporte, tanto en navegadores como en NodeJS. También se pueden indicar rutas absolutas para cargar directamente desde el navegador:

```
// Incorrecto (no empieza por . o por /)
import { elemento } from "module.mjs";
import { elemento } from "folder/module.mjs";

// Correcto
import { elemento } from "./module.mjs";    // misma carpeta del .js
import { elemento } from "/module.mjs";     // carpeta raíz
import { elemento } from "../module.mjs";   // carpeta anterior al .js
import { ceil } from "https://unpkg.com/lodash-es@4.17.11/lodash.js";
```

En resumen, las exportaciones e importaciones en Javascript son una herramienta indispensable para escribir código Javascript moderno de una forma organizada y productiva.

Para saber más:

- Módulos (import/export): https://www.youtube.com/watch?v=0GEUyQXe3NI&list=PLvq-jIkSeTUZ6QgYYO3MwG9EMqC-KoLXA&index=35
- Export e import: https://bluuweb.github.io/desarrollo-web-bluuweb/11-12-js-modulos/#alternativa-2
- Módulos: https://es.javascript.info/modules
- Import dinámico en JS: https://lenguajejs.com/javascript/caracteristicas/dynamic-import/

20.9. Webs con ejercicios para practicar

- https://codesignal.com/. Usado por Facebook
- https://leetcode.com/.
- https://www.hackerrank.com/

20.10. Recursos para superar entrevistas técnicas

- https://www.techinterviewhandbook.org/
- https://github.com/DopplerHQ/awesome-interview-questions
- https://github.com/jwasham/coding-interview-university

20.11. Canales Youtube

- midulive: https://www.youtube.com/c/midulive
- bettatech: https://www.youtube.com/channel/UCSf6S_PAhXsqGMTPDiKgdRg
- holamundo: https://www.youtube.com/c/HolaMundoDev/videos
- fernandoherrera: https://fernando-herrera.com/#/
- codewithaniakubów: https://www.youtube.com/channel/UC5DNytAJ6_FISueUfzZCVsw