DESARROLLO WEB
ENTORNO CLIENTE

2º DAW
CFGS

2. JS: MANEJO DE LA SINTAXIS USO DATOS PRIMITIVOS

Natalia Escrivá Núñez

IES Serra Perenxisa

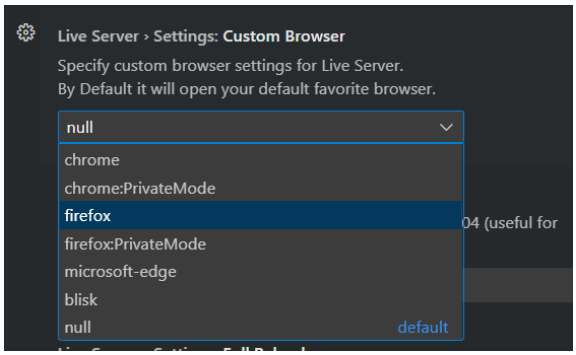
n.escrivanunez@edu.gva.es

CONTENIDO

1. VISUAL ESTUDIO CODE Y EXTENSIONES

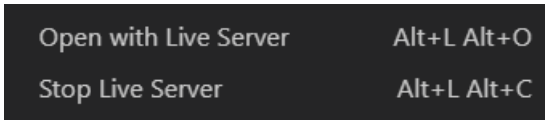
Es el editor de código que vamos a utilizar a lo largo de este curso y vamos a tener ayuda de algunas extensiones para la realización de las prácticas del curso.

Live Server: Nos proporciona un servidor local que se recargará directamente con cada cambio que hagamos en el código. Utiliza el navegador predeterminado. Si queremos utilizar otro navegador que no sea el predeterminado, lo podemos indicar en File → Preferences → Settings → Buscamos "Live Server" e indicamos el navegador que queramos utilizar.



Si queremos iniciar la ejecución, tenemos dos opciones. Para pararla, también dos alternativas.

Pantalla Visual Studio margen inferior

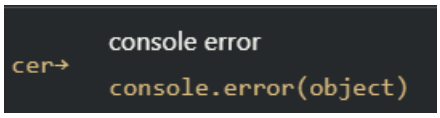


Botón derecho del ratón

Flatland Monokai Theme: Es un tema de color que resalta muy bien cada tipo de elemento del código.

Si queremos ver más temas, podemos utilizar CRTL+SHIFT+P y poner "color theme". Podremos navegar a través de las posibilidades que nos da Visual Studio Code.

JavaScript (ES6) code snippet: Facilita la codificación gracias a la simplificación.



VsCode-Icons: Proporciona un icono identificativo a cada archivo.

Open In Browser: Permite abrir la aplicación en un navegador que NO es el predeterminado.

2. CÓMO AÑADIR CÓDIGO JS EN HTML

Tenemos diferentes opciones para insertar código JavaScript en nuestra página web o documento HTML: en el propio documento o en un archivo externo.

Debemos recordar el orden en el que se inserta el código, ya que, al ser un lenguaje interpretado, lo irá ejecutando según se lo vaya encontrando.

2.1. Incluir JS en el mismo documento HTML

En este caso el código JavaScript **se encierra entre etiquetas <script></script> y se incluye en cualquier parte del documento**.

Lo podremos incluir en el <head> </head> o en el <body></body>.

Cuando el navegador ejecuta un archivo HTML primero ejecuta lo que hay en <head>, antes de cargar el cuerpo de la página.

Lo debemos poner antes del cierre del </body>.

El navegador va ejecutando el código y cuando llega a la etiqueta <script> ya sabe que ha de interpretar código JavaScript.

Este primer método que hemos visto NO es recomendable por las siguientes razones (que sí nos permite el último método que veremos):

- → No permite separar el código HTML del código JS, quedando un trabajo menos simple y más complicado de mantener y de leer.
- → No tendremos la posibilidad de reutilizar código.

LA INSERCIÓN DE JAVASCRIPT DENTRO DEL CÓDIGO HTML NO LO USAREMOS, UTILIZAREMOS SIEMPRE EL SIGUIENTE MÉTODO

2.2. Incluir JS a través de un archivo externo

Las mismas instrucciones de JavaScript que se incluyen entre un bloque <script></script>, pueden almacenarse en un fichero externo con extensión .js

La forma de acceder y enlazar esos ficheros .js con el documento HTML es a través de la propia etiqueta <script></script> que añadiremos al documento HTML indicando el atributo de la ruta donde se encuentra el fichero .js.

Las etiquetas <script> src = "…"</script> con la ruta RELATIVA del fichero las pondremos justo antes de la etiqueta </body>

```
<body>
    <div class="contenido">
        <h1>UDAD 2. Introducción a JavaScript</h1>
        <a href="http://www.w3schools.com" target="_blank">www.w3s
        <h2>Cómo añadir código JS en HTML: </h2>
        <p>--> Se puede poner el código en cualquier parte del fic
        <p>--> Se puede poner el código en una parte del HTML e in
        <p>--> Este método NO es el recomendado!!!!</p>
        <p>--> Separemos SIEMPRE el código JS</p>
        <p>--> Accedemos a él a través de la RUTA RELATIVA</p>
    </div>
        <script>
            function holamundo(){
                alert("hola mundo desde <body>");
            }
            holamundo();
        </script>
        <script src="../js/UDAD2.js"></script>
</body>
```

3. SINTAXIS DEL LENGUAJE

El lenguaje JavaScript tiene una sintaxis similar a la de Java o C++.

La sintaxis especifica aspectos como los caracteres que se deben utilizar para definir los comentarios, la forma de los nombres de las variables o la estructura de las instrucciones o sentencias del código.

3.1. Mayúsculas y minúsculas

JS es un lenguaje que **distingue entre mayúsculas y minúsculas**.

Esta regla es muy importante cuando utilizamos variables, objetos, funciones o cualquier otro símbolo del lenguaje.

No es lo mismo utilizar la función alert() que Alert()

En el primer caso, mostrará el contenido del paréntesis, y en el segundo, simplemente, se saltará esa sentencia y no la ejecutará, NO dará error.

3.2. Comentarios

Al igual que en la mayoría de los lenguajes de programación, JavaScript permite insertar comentarios dentro del código.

Estas líneas de código NO serán interpretadas por el navegador.

Su función principal es la de facilitar la lectura del código del programador. También se puede utilizar para evitar que se ejecute código.

Existen dos formas de insertar comentario:

→ Doble barra (//): Comentaremos una única línea de código

→ Barra + asterisco (/* xxx */): Comentaremos varias líneas de código

Un código bien comentado favorece su **comprensión** y su **mantenimiento**

3.3. Tabulaciones y saltos de línea

JavaScript ignora los espacios, tabulaciones y saltos de línea presentes en los símbolos del código. Si bien NO son necesarios para su interpretación y ejecución, SE RECOMIENDA que el código siga un orden con el uso de tabulaciones y saltos de línea para favorecer su comprensión.

> Un código bien indentado favorece su **lectura y comprensión**

Vamos a ver una comparativa:

```
<script>
    let i,j=0;
    for (i=0;i<5;i++){
    document.write("Variable i: "+i +";");
    for (j=0;j<5;j++) {
    if (i%2==0){
    document.write("<p></p>");
    document.write( " j: "+ j + " ");;}}}
</script>
```

```
<script>
    //declaramos variables
    var i,j=0;
    //Usamos un bucle
    for (i=0;i<5;i++){
        document.write("Variable i: "+i +";");

        //Usamos otro bucle dentro del anterior
        for (j=0;j<5;j++) {
            if (i%2==0){
                document.write("<p></p>");
                document.write( " j: "+ j + " ");
            }
        }
    }

</script>
```

3.4. El punto y coma

En JavaScript el punto y coma NO es obligatorio, a excepción de que queramos separar instrucciones.

Normalmente, se suele insertar un signo de punto y coma (;) al final de cada instrucción de JavaScript, al igual que ocurre en otros lenguajes de programación como C, C++ o Java.

Este signo tiene la **utilidad de separar y diferenciar cada instrucción**.

Podremos **omitir** este signo si cada instrucción o sentencia se encuentra en una línea independiente.

Por ejemplo, podríamos definir dos variables de esta forma:

let i=0

let j=2

Si queremos hacerlo en una única línea, haremos:

let i=0; let j=2;

Es recomendable:

PONER UNA SENTENCIA POR LÍNEA

USAR ; AL FINAL DE CADA INSTRUCCION

3.5. Palabras reservadas

JS contiene una serie de palabras que NO podemos utilizar a nuestro gusto porque dentro del lenguaje, tienen un significado especial.

Las palabras reservadas son términos que ofrece el lenguaje para poder programar.

Los nombres que damos a variables y otros elementos del lenguaje son identificadores, ya que su misión es precisamente identificar de forma inequívoca a un elemento del lenguaje.

Estas son las palabras reservadas a partir de ECMAScript 2015:

break	case	catch	class	const	continue
debugger	default (US)	delete	do	else	export
extends	finally	for	function	if	import
in	instanceof	new	return	super	switch
this	throw	try	typeof	var	void
while	with (US)	yield			

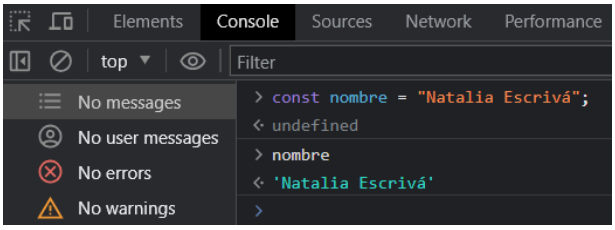
Para ampliar la información sobre palabras reservadas o clave, se puede consultar la página (gramática léxica)

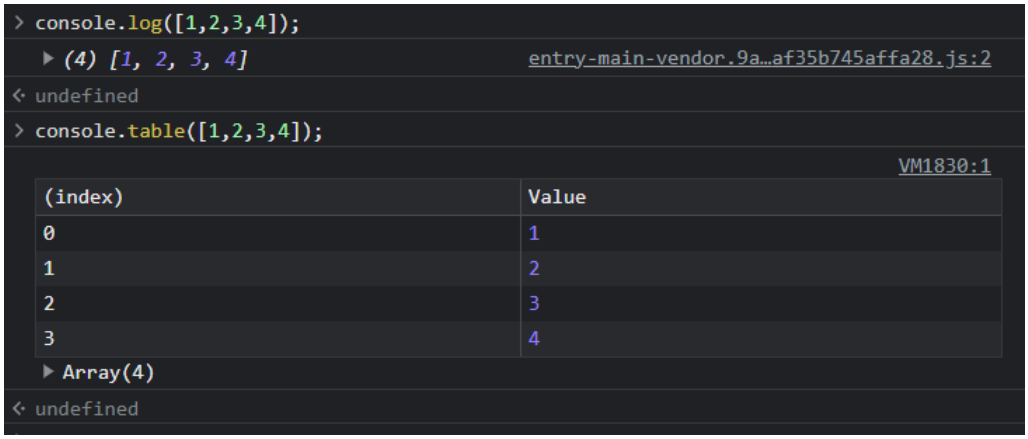
4. LA CONSOLA DE JS

Las herramientas de desarrollador de los navegadores incluyen una consola.

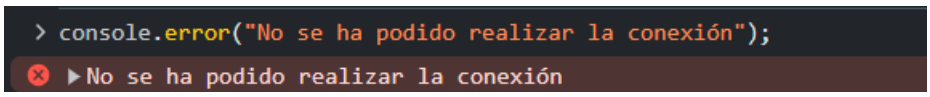
Este elemento nos va a ser muy útil ya que, a través de instrucciones, podremos mostrar información sobre nuestra aplicación o confirmar si el código está siguiendo la traza esperada.

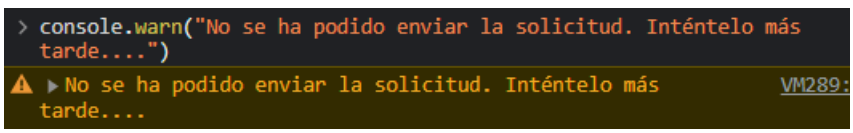
Podremos hacer pruebas y ver resultados:





Podremos ver los errores y advertencias:



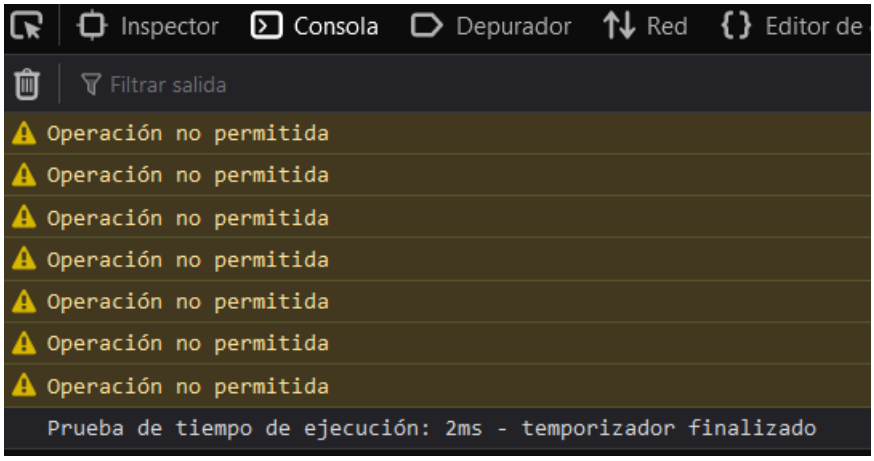


Podremos medir el tiempo de ejecución de un código:

```
console.time("Prueba de tiempo de ejecución");

console.warn("Operación no permitida");
console.warn("Operación no permitida");
console.warn("Operación no permitida");
console.warn("Operación no permitida");
console.warn("Operación no permitida");
console.warn("Operación no permitida");
console.warn("Operación no permitida");

console.timeEnd("Prueba de tiempo de ejecución");
```



5. VARIABLES Y CONSTANTES

Las variables son elementos característicos de los lenguajes de programación. NO se puede decir que HTML es un lenguaje de programación porque NO tiene variables, por ejemplo.

Son **zonas de memoria** en las que se almacenan los valores que podrán ser modificados a lo largo de la ejecución.

Estas variables son las que nos van a permitir manipular y guardar datos.

JS **NO es un lenguaje tipado y es dinámico**, por lo que **NO es necesario indicar qué tipo de valor corresponde a la variable que estamos creando**.

5.1. Declaración de variables

Las variables se deben declarar antes de ser utilizadas. Declarar una variable implica asignarle un identificador:

→ **Antes de ES6 o ES2015**, se utilizaba la palabra reservada **var**.

→ Tras **ES6**, se permite el uso de **let** para declarar variables (además de **var**).

Para asignar el valora a una variable, usaremos el **operador de asignación =.**

Cada variable tiene un **identificador único**, que, como hemos visto, NO podrá ser una palabra reservada (*keywords*). A ese identificador se le asocian los valores que tendrá la variable.

Los **identificadores** deberán ser **descriptivos**, de forma que, con leer el nombre de la variable, ya podemos imaginar el tipo de valor que almacena y el sentido de esa variable dentro del código.

Para **declarar las variables**, tendremos en cuenta lo siguiente respecto a los **identificadores**:

→ Los identificadores comenzarán por una letra, $ o guión bajo.

```
let 01_usuario;
let usuario_01;
```

→ Deben seguir por una letra, $, guion bajo o un número.

→ Si usamos dos palabras, seguiremos el **estilo Camel Case:** La primera empezará con minúscula y la segunda con mayúscula.

```
let nombreCompleto = "Natalia Escrivá Núnez";
```

También podemos utilizar el estilo Pascal Case: La primera empezará con mayúscula y la segunda también.

```
let NombreCompleto = "Natalia Escrivá Núnez";
```

También podemos separar las palabras por un guión bajo: Esta opción es conocida como **underscore** o **Snake Case**.

```
let nombre_completo = "Natalia Escrivá Núnez";
```


Aspectos que tendremos en cuenta relacionados con el manejo de las variables:

→ Si declaramos una variable y no le asignamos valor, es "**undefined**".

→ Las variables se pueden declarar e inicializarlas más tarde.

```
let rebajado;
rebajado = true;
```

→ Podemos declarar varias variables en una sola sentencia

```
var nom, apellido;
```

→ Podemos asignar valor a varias variables en una única sentencia

```
var nom = "Natalia", apellido = "Escrivá";
```

→ Podemos separar las asignaciones anteriores en distintas lineas

```
var x = 3,
    y = 5,
    w = 9;
```

→ Podemos asignar operaciones a una variable

```
var result = x + y + 5;
```

5.2. Ámbito de las variables

El ámbito de las variables o *scope*, **define la zona del programa en la que se podrá utilizar la variable**, es decir, el alcance de esta.

Existen dos ámbitos de variables: **global** y **local** (aquí incluimos el alcance de bloque de código).

Vemos las **diferencias** entre las variables locales y globales son:

→ Las variables **locales** se definen **dentro** de las funciones, las **globales** se definen **fuera** de las funciones.

→ Las variables **locales** tienen su **ámbito** de actuación **dentro** de la función o del bloque de código y las **globales** en **toda la página**.

→ Las variables locales son accesibles únicamente desde dentro de la función y las globales desde dentro y fuera de la función.

→ Las varialbles locales desaparecen cuando desaparece la función y las globales cuando se sale de la página.

→ Si declaramos una **variable sin la palabra reservada**, la convertimos en **global**.

Veamos un ejemplo:

```
//x, w e y son globales
var x = 3;
let y = 5;
w = 9;

nombre = "Natalia Escrivá";

function ambVar(){
    var x = 15; //local
    let z = 55; //local
    w = 25;
    console.log ("x vale: " + x); // 15, valor local
    console.log ("y vale: " + y); // 5, valor global
    console.log (z); // 55, local
    console.log (nombre);
    console.log ("w vale: " + w); //25, valor local
}

console.log (w); // 9, global
ambVar();


/*Estas ejecuciones mostrarán las globales,
no tienen acceso a las variables locales*/
console.log (nombre);
console.log(x); //3
console.log (y);//5
console.log (w); //vale 25, modificado valor variable global
//console.log (z); //da error, pq es local
```

Podemos ver que, dependiendo de dónde estén situadas las sentencias, las ejecuciones tendrán un resultado u otro.

SIEMPRE usaremos la palabra reservada para declarar variables para evitar generar variables globales no controladas

5.3. Use Strict

El modo estricto o *use strict* es una forma que tenemos de controlar el buen uso del lenguaje.

Utilizando esta línea de código estaremos obligados a utilizar las variables siempre y cuando se hayan declarado anteriormente con la palabra reservada.

Esta sentencia tiene **dos ámbitos de actuación: global y local.**

```
"use strict";

let nombre;
let apellidos;

function nomCompleto(){
    nombre = "Natalia";
    apellidos = "Escrivá Núñez";
    alert ("Yo soy " + nombre + " "+ apellidos);
}

nomCompleto();
```

Ámbito **global…**

En este caso la ejecución es **correcta**, se han declarado correctamente las variables y se les ha asignado valor.

JS 4.3.2_useStrict.js > ...

```
"use strict";

let nombre;
//let apellidos;

function nomCompleto(){
    nombre = "Natalia";
    apellidos = "Escrivá Núñez";
    alert ("Yo soy " + nombre + " "+ apellidos);
}

nomCompleto();
```

Ámbito **global…**

Cuando se ejecute este código, nos encontraremos un **error**, ya que hemos asignado un valor a una variable que NO ha sido previamente declarada.

```
Uncaught ReferenceError: apellidos is not defined
    at nomCompleto (03-app.js:13:16)
    at 03-app.js:19:2
```


Aquí estamos hablando de un uso estricto de las variables a nivel **global** pero también podríamos poner "use strict" dentro de la función y únicamente afectaría a las variables no declaradas en la función.

Vamos a ver cómo funciona a nivel **local**:

```
let nombre;
//let apellidos;


function nomCompleto(){
    "use strict";
    nombre = "Natalia";
    let apellidos = "Escrivá Núñez";
    mascotas = 2;
    alert ("Me llamo " + nombre);
    alert ("Yo soy " + nombre + " "+ apellidos + " y tengo " + mascotas+ " mascotas.");
}

nomCompleto();
```

```
Uncaught ReferenceError: mascotas is not defined                03-app.js:14
    at nomCompleto (03-app.js:14:15)
    at 03-app.js:19:2
```

5.4. Diferencia entre let y var

Existen varias diferencias entre el uso de var y let:

→ **Redeclaración de variables**: **var** permite la redeclaración y con let da error, incluso si está en ficheros diferentes

El valor con el que trabaja es el último asignado.

```
//var SI permite la redeclaración
var a = 5; //esta variable se declaró en el fichero 01-app.js
console.log(a);
//let NO permite la redeclaración
let rebajado = false; //esta variable se declaró en el fichero 01-app.js
console.log(rebajado);
```

La variable "a" la muestra sin problema pero veamos qué ocurre con **let**… Aquí se puede ver con **F12, en la consola, el error** que presenta.

```
Uncaught SyntaxError: Identifier 'rebajado' has already been declared (at    04-app.js:1
04-app.js:1:1)
```


→ El **ámbito de las variables**: Con **let** se limita el ámbito (*scope*) de la variable al **bloque**. NO se podrá utilizar fuera del ese ámbito. **Añade un aspecto más estricto al lenguaje.** Es más definido incluso que lo que llamamos "local".

```
function ambitoVariables(){
    let num = 7;
    if (num == 7){
        var variable1 = 1;
        let variable2 = 2;
    }
    console.log(variable1);
    console.log(variable2); //DA ERROR!!
}
ambitoVariables();
```

Cuando esta función se invoque, veremos que variable2 no se va a mostrar puesto que está fuera del alcance de su declaración.

5.5. Declaración de constantes

A partir de ES6, se incorpora **const**.

Existen muy pocas diferencias entre let y const en relación con las **reglas para crear las variables y su ámbito de actuación (bloque).**

Su valor NO puede ser reasignado. El valor con el que se declare o inicialice, será su valor para toda la aplicación.

```
const curso = "2DAW";
curso = "2DAM";
```

```
▶Uncaught TypeError: Assignment to constant variable.
    at 05-app.js:7:7
```

Además, las constantes deben inicializarse en su declaración. De hecho, el propio editor de textos nos avisa del error.

```
'const' declarations must be initialized. ts(1155)
const precio: any
View Problem (Alt+F8)    No quick fixes available
//Debe
const precio;
```

Los únicos elementos que Sí se van a poder modificar a pesar de haberse declarado con const son los objetos y los arrays.

5.6. Valores primitivos

Los valores son datos que el programa va a almacenar, manipular y/o mostrar.

En JavaScript encontramos tres tipos de **valores primitivos**:

- → Valores de texto, cadenas o Strings
- → Valores numéricos
- → Valores Booleanos

También podemos encontrar el **tipo de datos compuesto** llamado **objeto** que representa una **colección de valores primitivos o de valores compuestos por otros objetos**. Esta colección de valores nos permite definir tipos de datos.

Por último, hablaremos de los valores **undefined** y **null**.

- → **Undefined**: Incluye los tipos de datos que no están definidos. Si declaramos una variable y no le asignamos valor, su tipo es undefined.
- → **Null**: Variables que tienen tipo de dato, pero su valor es nulo. Si declaro una variable de tipo Number y le asigno valor y después le asigno null, el tipo de la variable será object, no Number

6. INSTRUCCIONES DE SALIDA

Estas instrucciones **nos van a permitir dar información al usuario**.

Los navegadores nos van a permitir utilizar un **objeto llamado window**. Este objeto tiene unos métodos que nos van a facilitar la comunicación con el usuario e **incluso pedirle información o interactuar con él.**

Si bien acabaremos utilizando otro tipo de componentes para pedir información y formularios, para empezar, vamos a conocer estos métodos simples.

Encontramos distintos tipos de instrucciones de salida:

6.1. window.alert()

Mostrará una **ventana emergente** en la página. Podemos mostrar una cadena de caracteres o evaluar una expresión.

Puesto que es un método de un objeto predefinido por JS, no será necesario anteponer a la función el objeto window.

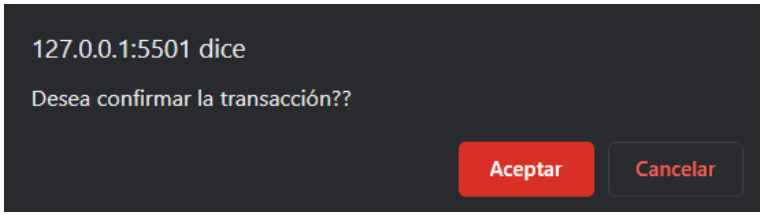
```
window.alert ("Usando el método alert()");
alert (3*5);
```

6.2. window.confirm()

Mostrará una **ventana emergente en la página con botones de Aceptar y Cancelar.** La función devuelve true o false en función del botón que pulse el usuario.

Puesto que es un método de un objeto predefinido por JS, no será necesario anteponer a la función el objeto window.

```
window.confirm ("Desea confirmar la transacción??");
```



6.3. window.prompt()

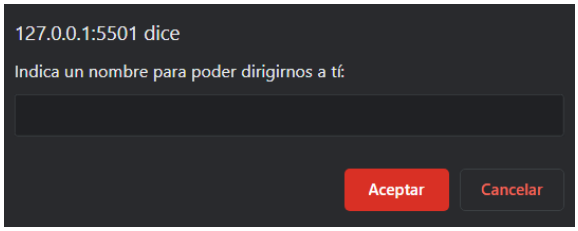
Mostrará una **ventana emergente** en la página que muestra un mensaje e **interactúa con el usuario para recoger información** que se deberá escribir en la propia ventana.

Puesto que es un método de un objeto predefinido por JS, no será necesario anteponer a la función el objeto window.

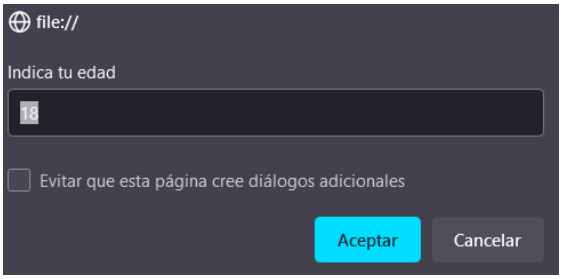
OJO!! Lo que se recoge es un **String**.

Se puede incluir un **valor por defecto**.

```
const nombre = prompt ("Indica un nombre para poder dirigirnos a tí: ");
alert ("Para nosotros eres " + nombre);
```




```
const mayorEdad = prompt ("Indica tu edad", 18);
```



6.4. window.write()

Escribe directamente en el documento HTML. Se pueden utilizar elementos HTML dentro de la instrucción.

NO es recomendable su uso a excepción de realizar ciertas pruebas.

```
document.write ("<h2>Natalia Escrivá</h2>");
document.write ("<h2>8/2</h2>");
```

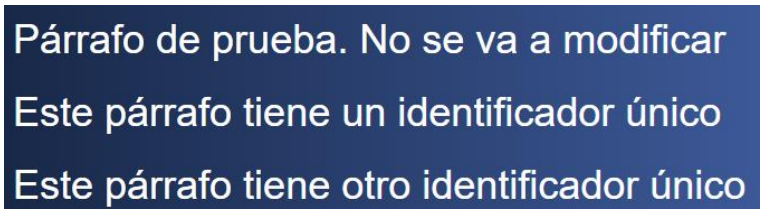
6.5. innerHTML

Se utiliza con el **DOM**. Es una instrucción que **actúa sobre un elemento del documento HTML**.

Siempre se utilizarán en elementos de etiquetas de texto, no de imágenes.

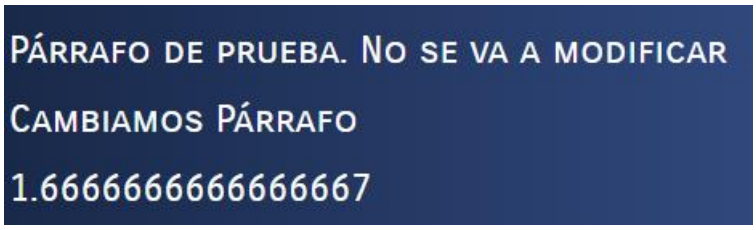
En el documento HTML tenemos estos párrafos:

```
<p>Párrafo de prueba. No se va a modificar </p>
<p id="OtroParrafo"> Este párrafo tiene un identificador único </p>
<p id="UltimParrafo"> Este párrafo tiene otro identificador único </p>
```



Este es el código del fichero JS:

```
document.querySelector('#OtroParrafo').innerHTML="Cambiamos Párrafo";
document.querySelector('#UltimParrafo').innerHTML=5/3;
```



6.6. console.log()

La información **únicamente se podrá ver desde la consola del navegador** (F12 o "Inspeccionar").

Nos podrá servir para mostrar mensajes como puntos de control, pero sin que interfieran en el documento HTML.

```
console.log("Texto visible en la consola");
console.log(2+2);
```

```
Texto visible en la consola
4
```

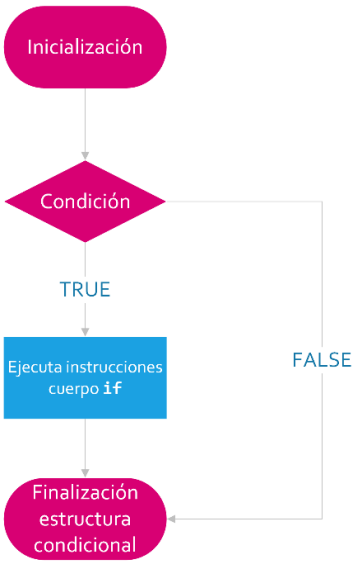
7. ESTRUCTURAS DE CONTROL DE CÓDIGO

JavaScript es un lenguaje estructurado, es decir, está basado en tres pilares o estructuras básicas:

- **Secuencia**: Una instrucción va detrás de otra.
- **Selección**: Estructuras de control condicionales.
- **Iteración**: Estructuras de control repetitivas.

Podremos ejecutar sentencias según se cumpla o no una **condición** (se evalúa una expresión lógica), **ejecutar sentencias un número determinado de veces** o **ejecutarlas hasta o mientras se cumpla una condición**.

7.1. Estructuras de control de selección



La sentencia "**if**" permite ejecutar una serie de instrucciones siempre que la **expresión lógica** que se evalúa sea **cierta**.

Si esa **expresión lógica** es **falsa**, NO se ejecutarán esas instrucciones y pasará al siguiente bloque de código.

```
const nota = prompt ("¿Qué nota has sacado en el examen?");
if (nota >= 5) {
    alert ("Has aprobado!!");
}
```

7.1.1. DETENER LA EJECUCIÓN DE UN IF

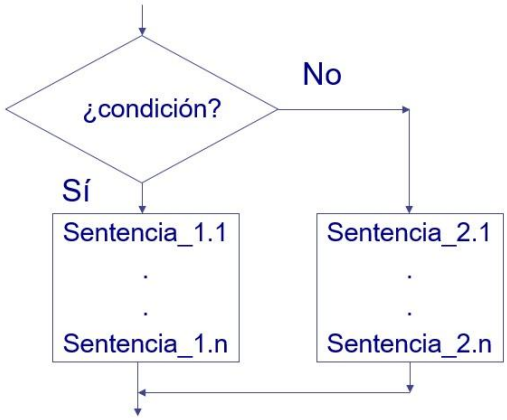
Podemos detener la ejecución de código que siga a una instrucción if. Esto favorecerá NO tener muchas sentencias de selección anidadas.

Tendremos en cuenta:

- Usaremos la palabra reservada **return**.
- El código deberá estar dentro de una **función**.

```
function aprobado(){
    const nota = prompt ("¿Qué nota has sacado en el examen?");
    if (nota >= 5) {
      alert ("Has aprobado!!");
      return;
  }
    if (nota < 5){
      alert("Has suspendido!!");
      return;
    }
}
```

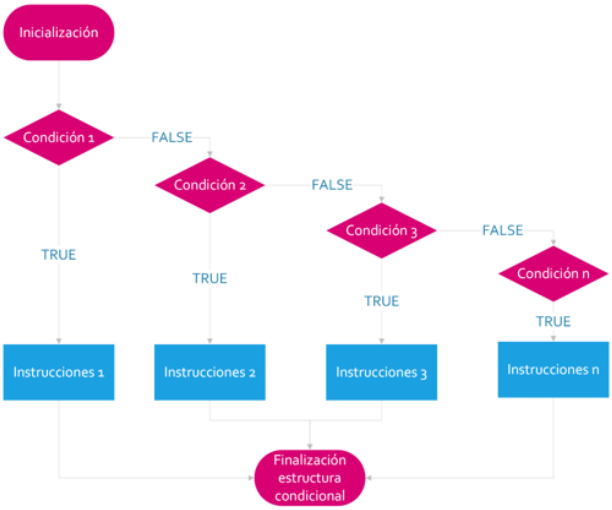
7.1.2. SECUENCIA CONDICIONAL COMPUESTA IF…ELSE



Es igual a la anterior pero añadiendo una serie de sentencias que se ejecutarán si NO se cumple la condición propuesta.

```
const nota = prompt ("¿Qué nota has sacado en el examen?");
if (nota >= 5) {
    alert ("Has aprobado!!");
} else {
    alert ("Has suspendido, suerte para la próxima!!");
}
```

7.1.3. SECUENCIA CONDICIONAL IF…ELSE…IF



Dentro de una sentencia if podemos colocar más sentencias if.

A esto lo llamamos **anidación**.

De esta forma se evalúan más condiciones que permiten afinar la complejidad del programa.

De esta forma NO necesitamos detener la ejecución de un if, ya que en cuanto la condición evaluada sea verdadera, ya no sigue evaluando más condiciones.

```
const nota = prompt ("¿Qué nota has sacado en el examen?");
if (nota < 5) {
    alert ("Has suspendido, debes prepararte mejor");
} else if ((nota >= 5) && (nota < 6)) {
    alert ("Has aprobado por los pelos!!");
} else if ((nota >= 6) && (nota < 7)) {
    alert ("Has aprobado con un bien, no está mal!!");
} else if ((nota >= 7) && (nota < 9)) {
    alert ("Has aprobado con un notable, vas bastante bien!!");
} else if ((nota >= 9) && (nota <=10)) {
    alert ("Tienes un sobresaliente, enhorabuena!!");
}
```

7.1.4. SECUENCIA SWITCH…CASE

Esta estructura es una variante de la anidación, es decir, equivaldría a hacer varios if – else if.

Lo utilizaremos cuando tengamos un número de condiciones a revisar numeroso, y de esa forma, se lee mejor.

Al igual que en el caso de if, **switch evalúa una expresión (o varias separadas por comas) comparando su valor con instancias case**, es decir, va comparando cuál de esas opciones se corresponden con la expresión, y ejecuta las instrucciones asociadas.

La expresión se evalúa una única vez y, si hay coincidencia, se ejecutan las instrucciones que correspondan a la instancia case coincidente.

En esta estructura **introducimos dos instrucciones**:

→ break: Permite salir de la estructura sin seguir comprobando casos.

→ default: Indica cualquier valor que no está recogido con anterioridad.

```
let nota = prompt ("¿Qué nota has sacado en el examen? (Sin decimales...)");
    nota = Number(nota);

switch (nota) {
    case 0:
    case 1:
    case 2:
    case 3:
    case 4:
        alert ("Has suspendido, debes prepararte mejor");
        break; //si esta es la opción, sale y no evalúa más
    case 5:
        alert ("Has aprobado por los pelos!!");
    case 9:
    case 10:
    alert ("Tienes un sobresaliente, enhorabuena!!");
    break;
    default:
        alert ("La nota debe estar entre 1 y 10");
}
```

```
function aprobadoSwitchDecimales(){
    let nota = prompt ("¿Qué nota has sacado en el examen?");
        nota = parseFloat(nota);

    switch (true) {
        case (nota < 5):
        alert ("Has suspendido, debes prepararte mejor");
        break; //si esta es la opción, sale y no evalúa más
        case ((nota >= 5) && (nota < 6)):
        alert ("Has aprobado por los pelos!!");
        break;
        case ((nota >= 6) && (nota < 7)):
        alert ("Has aprobado con un bien, no está mal!!");
        break;
        case (nota >= 7) && (nota < 9):
        alert ("Has aprobado con un notable, vas bastante bien!!");
        break;
        case (nota >= 9) && (nota <=10):
        alert ("Tienes un sobresaliente, enhorabuena!!");
        break;
        default:
            alert ("La nota debe estar entre 1 y 10");
    }
}
aprobadoSwitchDecimales();
```

Tendremos en cuenta la **limitación** de esta estructura:

Evaluar casos más complejos añadiendo expresiones lógicas.

Para solucionar la limitación, lo que haremos será **comprobar un valor booleano** directamente **en vez de comprobar el valor de la variable o expresión**.

7.1.5. OPERADOR TERNARIO

Operador que nos permite evaluar una condición y ejecutar código según sea verdadera o falsa.

```
const autenticado = true;
alert(autenticado ? "Usuario de la comunidad" : "No estás autenticado");
```

Analizando la sintaxis:

- ? → A su izquierda pondremos la condición a evaluar.
- : → Hace la función de "else". A su izquierda pondremos el código a ejecutar si la condición resulta verdadera. A su derecha el código a ejecutar si la condición resulta falsa.

Podremos evaluar más de una expresión:

```
const tieneCredito = false;
alert(autenticado && tieneCredito ? "Bienvenido a la pasarela de pagos" : "No estás autenticado o debes reponer crédito");
```

Además podremos introducir instrucciones de selección dentro del operador ternario:

```
console.log(autenticado ? tieneCredito ? "Elige la forma de pago" : "Autenticado SI, crédito NO" :"No estás autenticado :( ");
```

Analizando la sintaxis:

La primera condición a evaluar es "autenticado":

- Si es verdadera, pasará a analizarse el siguiente ternario:
  - Si "tieneCredito" es verdadera → `Elige la forma de pago`
  - Si "tieneCredito" es falsa → `Autenticado SI, crédito NO`
- Si es falsa → `No estás autenticado :(`

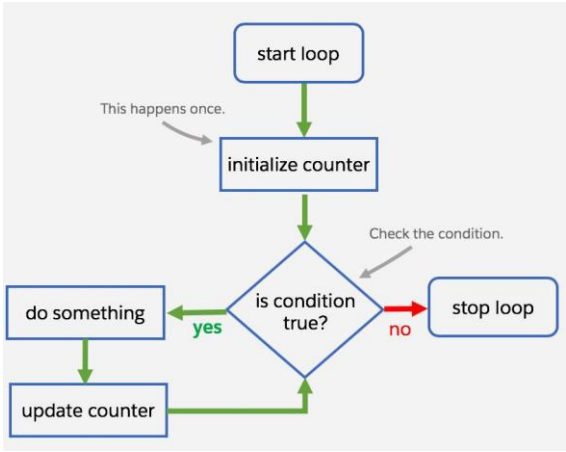
7.2. Estructuras Iterativas

Estas estructuras nos permiten repetir un bloque de código un número determinado de veces, siempre que se cumpla o no se cumpla una condición.

7.2.1. BUCLE FOR

Primera de las **estructuras de control de repetición**. Nos va a permitir repetir el código un número determinado de veces (mientras se siga cumpliendo una condición).

Consiste en tres expresiones en las que se incluye la condición que ha de cumplirse. Además incorpora un bloque de instrucciones.




```
let i;
//Con las tres expresiones
for (i = 1; i < 6; i++){
    console.log (i);
}
```

Veamos las tres expresiones que, se pueden omitir:

→ La primera es la **variable contador** que deberemos inicializar. Si se omite, debemos declarar e inicializar la variable antes.

```
//eliminamos la primera expresión
i = 1;
for (; i < 6; i++){
    console.log (i);
}
```

→ La segunda es la **condición** que se evaluará en cada vuelta del bucle. Si se omite, debemos añadir dentro del bloque de instrucciones la condición y una salida del bucle con la sentencia **break**.

```
//eliminamos la condición
for (i = 1; ;i++){
    if (i < 6){
        console.log (i);
    } else {
        break;//para salir del bucle
    }
}
```

→ La última es la forma en la que se **actualiza el contador**: incremento o decremento. Si se omite, debemos incrementar o decrementar la variable dentro del bloque de instrucciones.

→ Si quisiéramos ir de dos en dos, podríamos actualizar el contador así **i+=2**.

```
//Eliminamos la última expresión
for (i = 1; i < 6;){
    console.log (i);
    i++;
}
```

Además podremos **utilizar más de una variable con la que trabajar dentro del bucle.**

En este caso, en cada vuelta del bucle, cada variable sufre una modificación.

```
//Utilizamos dos o más valores dentro del bucle
let y, z;
for (y=1, z=5; y<6, z>=1; y++, z--){
    console.log("y vale " + y+ " y z vale "+z);
}
```

```
y vale 1 y z vale 5
y vale 2 y z vale 4
y vale 3 y z vale 3
y vale 4 y z vale 2
y vale 5 y z vale 1
```

Por último, comentamos los **bucles anidados**, es decir, por cada vuelta del bucle principal, se ejecutará el bucle interno.

```
//Uso de bucles anidados
let m, n;
for (m=0; m<4; m++){
    for (n=3; n>0 ; n--){
        console.log ("m= " + m + " y n= " + n);
    }
}
```

```
m= 0 y n= 3
m= 0 y n= 2
m= 0 y n= 1
m= 1 y n= 3
m= 1 y n= 2
m= 1 y n= 1
m= 2 y n= 3
m= 2 y n= 2
m= 2 y n= 1
m= 3 y n= 3
m= 3 y n= 2
m= 3 y n= 1
```

7.2.1.1. Break y continue en for

Estas dos sentencias nos van a permitir manipular o modificar el flujo o secuencia normal del programa.

Hemos visto en la estructura switch la sentencia **break**, que nos servía para salir del bloque de instrucciones una vez se había evaluado la correcta y así no seguir evaluando otras.

En los bucles también se puede **forzar la salida** de estos.

Por otro lado, la sentencia **continue**, nos permite interceptar un elemento, identificarlo y continuar la ejecución, es decir, **rompe el ciclo, pero NO la ejecución**.

```
for (let i=0; i<=10; i++){
    if (i === 5){
        console.log("Este es el 5");
        break;
    }
    console.log(`Número: ${i}`);
}
```

```
***BREAK*********
Número: 0
Número: 1
Número: 2
Número: 3
Número: 4
Este es el 5
```

```
for (let i=0; i<=10; i++){
    if (i === 5){
        console.log("Número: CINCO");
        continue;
    }
    console.log(`Número: ${i}`);
}
```

```
*******CONTINUE*********
Número: 0
Número: 1
Número: 2
Número: 3
Número: 4
Número: CINCO
Número: 6
Número: 7
Número: 8
Número: 9
Número: 10
```

Estas sentencias se pueden utilizar además, para romper el flujo de un programa dentro de un bloque de código etiquetado.

Las etiquetas preceden una sentencia con el nombre de la etiqueta seguido de dos puntos. Si recoge una única instrucción, la podremos poner directamente seguida de los dos puntos, pero si recoge más de una sentencia, abriremos llaves.

En el caso de **continue** la etiqueta es opcional y SIEMPRE debe estar dentro de un bucle.

```
18  //Break con etiquetas
19  //creamos un array...
20  let mascotas = ["perro", "gato", "hamster"];
21  //creamos la etiqueta...
22  ver_mascotas:
23  {
24      console.log(mascotas[0]);
25      console.log(mascotas[1]);
26      break ver_mascotas;
27      console.log(mascotas[2]);
28  }
```

```
perro
gato
```


```
//Continue con etiquetas (opcional)
let j=0;
//creamos la etiqueta...
bucle:
    for (j=0; j<10; j++){
        if(j == 3 || j == 5){
            console.log ("El número es un 3 o un 5, continuamos...");
            continue bucle;
        }
        console.log(j);
    }
```

```
0
1
2
El número es un 3 o un 5, continuamos...
4
El número es un 3 o un 5, continuamos...
6
7
8
9
```

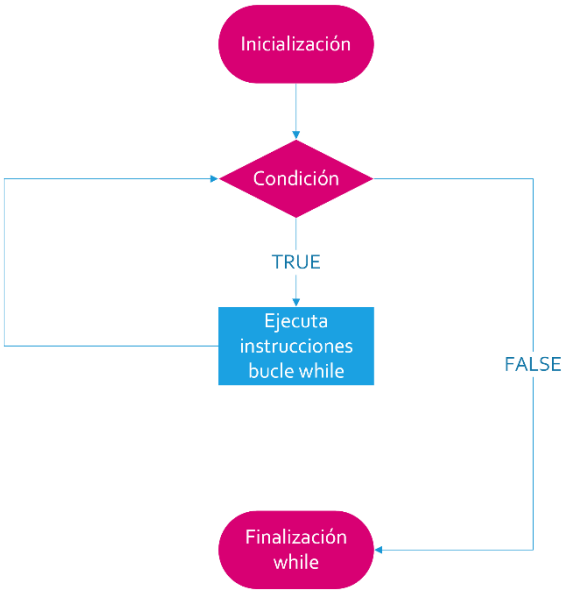
7.2.2. BUCLE WHILE CON CONTADOR

En este caso, tendremos un bloque de instrucciones que se van a ejecutar **mientras la condición que se evalúa es verdadera** → NO hay un número determinado de iteraciones.

La condición se evalúa ANTES de ejecutar las sentencias.

Si la condición que se evalúa es falsa, nunca entrará en el bucle.

Deberemos tener precaución con la condición, ya que, si siempre es cierta provocaremos un BUCLE INFINITO.




```
let i = 0;
while (i <= 50) {
    console.log (i);
    i+=10;
}
```

```
0
10
20
30
40
50
```

SI la variable i valiese 100, no saldría nada por consola.

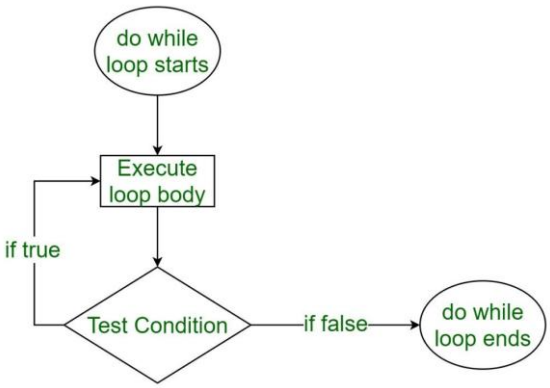
7.2.3. BUCLE DO…WHILE

Este tipo de bucle es como el anterior, con la única diferencia de que **la expresión lógica se evalúa DESPUÉS de ejecutar las sentencias**.

En estos casos, **el bucle se ejecuta como mínimo, una vez.**

```
i = 0;
do {
    console.log (i);
    i+=10;
} while (i <= 50);
```

```
0
10
20
30
40
50
```



Si ejecutamos el código, tendríamos una salida de 5 números, del 0 al 50, de 10 en 10.

Si la i valiese, por ejemplo, 100, la salida de esta ejecución sería 100 y saldría del bucle por no cumplir la condición.

7.2.4. BUCLES CON CENTINELA

Hasta ahora hemos visto **bucles con contador**, es decir, el elemento que controla cuando acaba el bucle es numérico, un incremento o decremento de una variable.

Tenemos **otra opción para decidir si el bucle se repite o no**: una **variable booleana o una expresión lógica**, a esto lo llamaremos **centinela**.

Veamos un ejemplo en el que se muestran por pantalla números aleatorios del 0 al 5 (ponemos 6 porque estamos usando parseInt que quita los decimales y la función random del objeto Math no llega al número límite, es decir, se quedaría en 4,99999999 si pusiéramos un 5).

La **variable salir inicializada a false** será que la se evalúe en cada iteración y también servirá como centinela, es decir, su valor cambiará a true si el número es par, de forma que el bucle finalizará:

```
let salir = false; //centinela
let y;
//condición del centinela para iteración
while (salir === false){
    //escribimos enteros aleatorios del 0 al 5
    y = parseInt(Math.random()*6);
    console.log(y);
    salir = (y%2==0);//centinela finaliza bucle
}
```

```
1
3
1
5
1
2
```

```
1
3
1
2
```

```
1
0
```

```
3
1
5
3
1
0
```

```
1
4
```

Podemos también **utilizar los dos métodos de control: contador y centinela**. Por **ejemplo**: Mostramos por pantalla números del 1 al 500 hasta que encuentre un número par, pero como mucho, se escribirá 6 veces:

```
//Bucle while con contador y centinela
let sale = false; //centinela
let num = 0;
let cont = 1; //contador

while (sale == false && cont <= 6){
    num = parseInt(Math.random()*501);
    console.log (num);
    cont++; //incrementamos contador
    sale = (num % 2 == 0); //centinela finaliza
}
```

```
489
118
341
491
264
117
55
223
404
```

8. FUNCIONES

Las funciones constituyen uno de los elementos más importantes de la programación estructurada.

Una función es un bloque de código que, a partir de unos datos de entrada o parámetros (o sin ellos), realiza **una tarea determinada.**

Su uso es muy importante porque nos va a **permitir dividir un problema en pequeños problemas e ir resolviéndolos por separado**.

Es importante destacar la concreción de la tarea ya que, precisamente eso es lo que la hace útil para nuestro programa o aplicación, que podremos **reutilizar el código** tantas veces como lo necesitemos. Por ejemplo, verificar si un usuario está autenticado o verificar un formulario.

La función que hemos creado deberá ser invocada o llamada para que el código se ejecute. Se puede invocar desde cualquier lugar del código y de diferentes formas: desde el código, con un evento o automáticamente (**autoinvocadas**).

Las funciones incorporan los siguientes **elementos** (aunque no siempre son necesarios):

→ Un **identificador** o nombre de la función: Cumplirán las mismas reglas que los identificadores de variables. Como convención formal, aunque no obligatoria, las funciones se identifican con nombres en minúsculas. El nombre NO siempre es necesario (funciones anónimas).

→ Uno o más **parámetros** que son variables locales a la función que sirven para almacenar los datos necesarios para que la función realice su tarea. Podemos encontrar funciones que NO utilicen parámetros.

→ Un **resultado** que es un valor que se devuelve a través de la instrucción ***return***. Es posible que una función NO devuelva ningún resultado, sino que, realice una determinada acción. Este tipo de funciones, que no usan return, en otros lenguajes se conocen como **procedimientos**.

→ Las **instrucciones** de la función, que son las sentencias que se ejecutan cuando se invoca a la función. Es el **cuerpo** de la función (entre llaves).

Debemos tener en cuenta que las funciones pueden llamarse unas a otras, es decir, en el cuerpo de la función, pueden invocar otras funciones.

8.1. Function declaration & expression

Tenemos dos formas de crear funciones:

- Function declaration
- Function expression

8.1.1. FUNCTION DECLARATION

Para declarar funciones utilizaremos la palabra reservada ***function()***.

```
function saludo(){
    alert("Hola");
}
saludo(); //invocamos la funcion
```

Aquí vemos que NO hay parámetros ni resultado, es un procedimiento.

```
function suma(a, b){
    return a+b;
}
alert(suma(7, 4)); //Invocamos la función
```

En este caso sí hay unos parámetros de entrada y devuelve un resultado.

Los resultados que devuelven un valor normalmente serán asignados a una variable.

```
const result = suma(10,20);
console.log(result);//30
```

8.1.2. FUNCTION EXPRESSION

Podemos asignar una función a una variable. En este caso tenemos funciones **anónimas**.

Ejemplo 1→ Sin parámetros

```
const despedida = function(){
    alert("Adiós");
}
despedida();
```

Ejemplo 2→ Con parámetros

```
const triple = function(x){
    return 3*x;
}
//invocamos la función
const b = prompt("Indica un número para obtener su triple");
console.log (`El triple de ${b} es: ${triple(b)}`);
```

Deberemos tener en cuenta que, NO será lo mismo lo que asignemos si ponemos en la asignación el nombre de la función con o sin paréntesis.

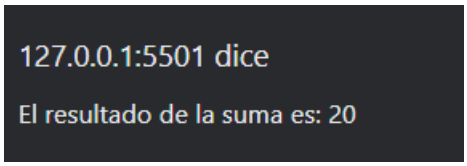
Ejemplo1:

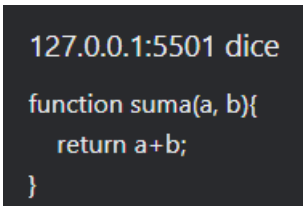
```
const resultSaludo = saludo();
const resultSaludo2 = saludo;
alert(resultSaludo2);
```

Lo que nos muestra el alert de resultSaludo2 es el código de la función

Ejemplo 2:

```
const resultSuma = suma(12, 8); //recoge el resultado del código
alert("El resultado de la suma es: " + resultSuma);
alert(suma); //Muestra el código de la función
```





Podemos **crear** una función anónima con el **constructor *Function***.

```
const producto = new Function ("i", "j", "return i*j");
```

Ejemplo 1→ Asignamos a una variable el resultado de la variable producto (función anónima).

```
const resulProducto = producto(3,5);
console.log(resulProducto);
```

```
15
```

Ejemplo 2→ Invocamos directamente la función.

```
console.log("4 x 8 son " + producto(4,8));
```

```
4 x 8 son 32
```

8.1.3. HOISTING

Hoisting o elevación es el comportamiento por defecto de JavaScript de "mover declaraciones al principio del código".

Esto quiere decir que JavaScript, por la forma en la que se ejecuta, puede declarar una función después de ser utilizada.

Esto es porque todas **las declaraciones (NO inicializaciones)** se mueven a la parte superior del ***scope*** actual.

Comparamos la llamada de dos funciones antes de ser declaradas.

Ejemplo 1→ Function declaration

```
restar(5,10);
function restar(num1, num2){
    console.log(`${num1} - ${num2} = ${num1 - num2}`)
}
```

```
5 - 10 = -5
```

Ejemplo 2→ Function expression

```
miNombre("Natalia");
const miNombre = function(nom){
    console.log(`Te llamas ${nom}`);
}
```

```
Uncaught ReferenceError: Cannot access 'miNombre' before initialization        01-app.js:64
    at 01-app.js:64:1
```

Podemos decir que **JavaScript se interpreta en dos fases**: de creación y de ejecución.

En la **primera fase** es donde se produce el ***hoisting***, se revisan todas las **declaraciones** de variables y funciones.

En la **segunda fase** es donde se produce la **ejecución**. Da error en el segundo ejemplo porque la función NO está declarada como tal. Para verlo más claro podríamos decir que JavaScript hace esto:

const miNombre;
miNombre("Natalia");

NO sabe qué ejecutar porque aún NO ha encontrado el código.

8.2. Funciones y métodos

Una **función** es un bloque de código que se puede invocar desde cualquier punto del programa.

Un **método** está ligado a un objeto. Es una propiedad de un objeto que contiene una definición de función.

Los métodos son funciones almacenadas como propiedades de objetos.

```
const num1 = 20;
const num2 = "20";
console.log(parseInt(num2));//función
console.log(num1.toString());//método
```

8.3. Funciones autoinvocadas

Las funciones se pueden invocar automáticamente.

Se ejecutarán en el momento en que el intérprete la encuentre.

Para ello, lo que debemos hacer es:

→ Poner detrás de la llave de cierre los paréntesis
→ Incluir todo el código entre paréntesis.

```
77  (function prueba(){
78      console.log("Esta función se ha invocado automáticamente!!!!!");
79  }());
```

```
Esta función se ha invocado automáticamente!!!!!                01-app.js:78
```

8.4. Parámetros y argumentos

Los **parámetros** son los valores que aparecen en la definición de la función. Una función podrá tener o no parámetros.

Los **argumentos** son los valores que le pasamos a la función cuando la invocamos.

```
function suma(a, b){//a y b son parámetros
    return a+b;
}
alert(suma(7, 4)); //7 y 4 son argumentos
```

A diferencia de otros lenguajes de programación:

- → **NO tendremos que especificar el tipo de dato de los parámetros.**
- → **NO se verifican los tipos de los argumentos**
- → **NO se verifica el número de argumentos.**

8.4.1. ARGUMENTOS POR DEFECTO Y POR EXCESO

Como ya sabemos, JavaScript es un lenguaje de tipificación débil por lo que no hay un control del número de argumentos que le pasamos a la función cuando vamos a invocarla.

En el caso en que le pasemos **menos**, dependiendo de las sentencias de la función, tendrá un comportamiento u otro.

En el caso en que le pasemos **más**, usará el mismo número de argumentos que parámetros tenga la función declarada.

Ejemplo 1:

```
function suma(a,b){
    return a + b;
}
console.log(suma(3)); //Devuelve NaN
console.log(suma (5,2,9)); //Usa los dos primeros
```

Ejemplo 2:

```
function concatena (x,y,z){
    console.log(x+y+z);
}
concatena("hola, ","soy "); //devuelve "hola soy undefined"
concatena("hola, ","soy ","Natalia", "Escrivá"); //devuelve "hola soy Natalia"
```

8.4.2. PARÁMETROS CON VALOR POR DEFECTO

Los parámetros pueden tener un valor predeterminado.

Cuando invoquemos la función, si no se le pasan los argumentos o son undefined, la función ya lo tiene contemplado y utilizará los valores predeterminados en la definición de la función.

Antes de poder utilizar los parámetros con valores predeterminados, debemos asegurar los valores que tendrían los parámetros si los argumentos no estaban bien pasados, ahora este proceso se simplifica.

Ejemplo 1→ Verificando y dando valor al argumento

```
function potencia (x,y){
    //Nos aseguramos de que tenemos y
    if (y === "undefined"); y = 1;
    return x**y;
}
alert(potencia(6));
```

Ejemplo 2 → Valor por defecto en la declaración de la función

```
function potencia2 (x,y=1){
    return x**y;
}
alert(potencia(6));
```

```
-----Parámetros con valor predeterminado----   02-app.js:40
6                                              02-app.js:48
6                                              02-app.js:54
```

8.4.3. NÚMERO DE PARÁMETROS NO DEFINIDO

En el estándar ES5, si el número de parámetros NO está definido, podemos utilizar el **método length del objeto arguments**. Este objeto está incluido en la función.

De esta forma los valores de los argumentos se podrán recoger y trabajar a través de una especie de array, aunque no se considera un array real.

Lo que tenemos es un objeto iterable tipo lista con pares clave: valor. Al ser iterable, podremos listarlos como lo haríamos con un array pero NO podremos utilizar los métodos del array para trabajar con los datos.

```
//NUM PARÁMETROS NO DEFINIDO -->  Método arguments.length
function valores (){
    alert("El número de argumentos es: " + arguments.length);
    for (let j=0; j<arguments.length; j++){
        alert ("argumento " + j +": "+ arguments[j]);
    }
}
valores(2,4,6,8,10);
```

```
El número de argumentos es: 5
argumento 0: 2
argumento 1: 4
argumento 2: 6
argumento 3: 8
argumento 4: 10
```

```
function mascotas(a, b){
    console.log("a vale: " + a);
    console.log("b vale: " + b);
    console.log ("Todos los elementos: " , arguments);
    console.log("Esta lista tiene " + arguments.length);
}
mascotas("perro", "gato", "hámster", "tortuga", "pájaro");
```

En la versión ES6, si el número de parámetros NO está definido, es decir, podemos tener un número de argumentos no concreto, podremos utilizar el **Spread Operator** o **Rest Operator** cuya sintaxis nos permite **representar un número indefinido de argumentos como un array.**

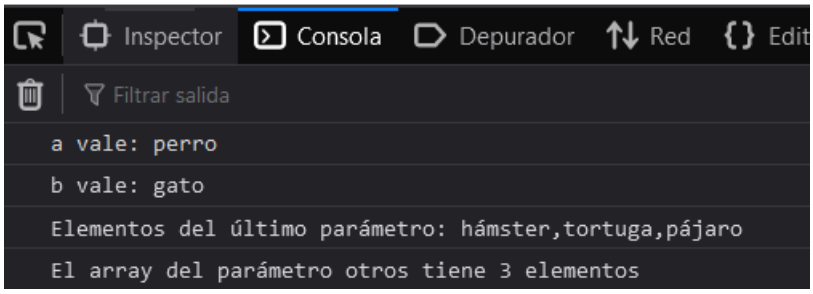
En este caso **SÍ hablamos de un array real.**

El último parámetro de una función (que recogerá los argumentos de más) debe ir precedido de **tres puntos seguidos**. Lo que hace es que todos los argumentos a partir de los tres puntos se colocan dentro de array estándar.

Únicamente el último parámetro puede ser un parámetro REST.

Vamos a comparar la última función para ver las diferencias entre arguments y REST:

```
function mascotas2(a, b,...otros){
    console.log("a vale: " + a);
    console.log("b vale: " + b);
    console.log(`Elementos del último parámetro: ${otros}`);
    console.log("El array del parámetro otros tiene " + otros.length + " elementos");
}
mascotas2("perro", "gato", "hámster", "tortuga", "pájaro");
```



Con este otro ejemplo, podemos ver que trabajamos los últimos argumentos (a partir del tercero), con un array.

```
89  function miFuncion(x,y,...resto){
90      console.log(`x = ${x}, y = ${y} y resto = ${resto}`);
91      let total=0;
92      for(let i in resto){
93          console.log(resto[i]);
94          total += i;
95      }
96      console.log("La suma del parámetro resto es: " + total);
97  }
98  miFuncion(1,2); //x=1, y=2, resto=array[]
99  miFuncion(1,2,3,4,5);
100 //x=1, y=2, resto=array[3,4,5]
```

```
x = 1, y = 2 y resto =
La suma del parámetro resto es: 0
x = 1, y = 2 y resto = 3,4,5
3
4
5
La suma del parámetro resto es: 0012
```

Por último, vemos que podemos prescindir de los primeros parámetros y utilizar únicamente el parámetro REST.

```
function media(...numeros){
    let acum = 0;
    for (let n of numeros){
        console.log(n);
        acum = acum + n;
    }
    return acum/numeros.length;
}
console.log(`La media es ${media(10,20)}`);
console.log(`La media es ${media(10,20,30)}`);
console.log(`La media es ${media(10,20,30,40)}`);
```

```
10
20
La media es 15
10
20
30
La media es 20
10
20
30
40
La media es 25
```

8.4.4. PASO DE ARGUMENTOS POR VALOR Y POR REFERENCIA

Cuando trabajamos con funciones debemos saber que los argumentos que les pasamos cuando las invocamos se pasarán de dos formas diferentes atendiendo a la naturaleza o tipos de datos de los parámetros de la función concreta:

→ **Paso por valor: Tipos primitivos o básicos (booleanos, numbers y strings)**

→ **Paso por referencia: Tipos complejos (arrays, sets, maps….)**

Cuando **pasamos una variable a una función por valor**, **lo que se recoge es una copia de su valor. La variable original NO se modifica.**

Ejemplo 1:

```
const a = 2; b = 3;
let resul = 0;
function sumaDoble(h,j){
    h = h*2;
    j = j*2;
    return h + j;
}
resul = sumaDoble(a,b);

console.log(a); //Muestra 2
console.log(b); //Muestra 3
console.log(resul); //Muestra 10
```

Cuando **pasamos una variable a una función por referencia**, **se está pasando el objeto concreto. La variable SI se modificará.**

Ejemplo 2:

```
const array = [1, 2, 3, 4, 5];
function s(v){
    v[0] = 6;
}
console.log(array); //Muestra 1,2,3,4,5
s(array);
console.log(array);//Muestra 6,2,3,4,5
```

8.5. Arrow Function

Las ***Arrow Functions*** o **Funciones Flecha** son una **variante** de las **funciones anónimas. NO** se podrá utilizar el **constructor function.**

La ventaja de este tipo de funciones es que permiten compactar una función convencional, facilitando su escritura. Siempre se **elimina** la palabra reservada **function que pasa a la derecha de los paréntesis con otra sintaxis ("=>")**y, según las características de la función, su **sintaxis** cambiará ligeramente:

Ejemplo 1 → Función anónima:

```
const fraseAnonima = function(){
    return "Ejemplo Function Expression o Función Anónima";
}
console.log(fraseAnonima());
```

Ejemplo 2 → Función flecha:

```
const fraseArrow = () =>"Transformando en Arrow function";
console.log(fraseArrow());
```

```
--------Arrow Function---------                    03-app.js:4
Ejemplo Function Expression o Función Anónima      03-app.js:10
Transformando en Arrow function                    03-app.js:12
```

Ejemplos de distintas sintaxis atendiendo a las distintas opciones de funciones flecha:

→ Si la función tiene **únicamente una sentencia que devuelve un valor**, podemos **eliminar las llaves y la palabra return**.

```
const ahora = ()=> new Date();
console.log (ahora());
```

→ Si hay **más de una sentencia**, usaremos las **llaves**.

```
const factorial = n=>{
    let acum = 1;
    for (let i=n; i>0; i--){
        acum = acum * i;
    }
    return acum;
}
let num = prompt ("Indica un número y se calculará su factorial");
alert (factorial (num));
```

→ Si **sólo** hay **un parámetro**, podremos **prescindir de los paréntesis**.

```
const tripleAnonima = function(x){
    return 3*x;
}
console.log(`El triple de 20 es ${tripleAnonima(20)}`);
const tripleArrow = x => 3*x;
console.log (`El triple de 100 es ${tripleArrow(100)}`);
```


→ Si contamos con **más de un parámetro**, usaremos el **paréntesis**.

```
const mediaAnonima = function(num1, num2, num3){
    return (`La media de los tres números introducidos es ${(num1 + num2 + num3)/3}`);
}
```

```
const mediaArrow = (x,y,z)=>(`La media de los tres números introducidos es ${(x + y + z)/3}`);
```

→ Podemos **usar una expresión como en cualquier función**.

```
const saludoFlecha = mensaje=>alert (mensaje);
saludoFlecha ("Buenas tardes a tod@s!!");
```

Si la función es compleja **NO** usaremos funciones flecha

8.6. Recursividad

La recursividad consiste en una técnica de resolución de problemas que se basa en la **capacidad que tienen las funciones de invocarse a sí mismas**.

La pila de llamadas de JavaScript admite esa posibilidad y lo que ocurre es que aparece varias veces el código de la misma función en la pila.

La idea es que cada invocación de la función resuelva parte del problema y se llame a sí mismo para resolver la parte que queda del problema, y así sucesivamente.

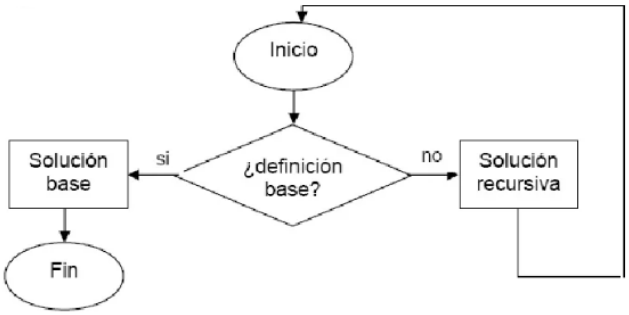
En cada llamada el problema es cada vez más sencillo hasta llegar a una llamada, en la que la función devuelve un único valor.

Esa última llamada cierra el bucle de llamadas y tras ella se irán resolviendo las anteriores hasta liberar la pila y conseguir el resultado final.

Los algoritmos recursivos tienen al menos **dos elementos**:

→ Caso base**: Evita que el algoritmo sea infinito. El problema se resolverá sin tener que hacer uso de una nueva llamada a sí mismo.**

→ Paso recursivo**: Relaciona el resultado del algoritmo en base a los resultados de casos más simples. Se hacen llamadas a la misma función, pero cada vez más próximas al caso base.**



RECURSIVIDAD	ITERACIÓN
Llamadas repetidas a los métodos	Instrucción de repetición explícita
Termina cuando se reconoce un caso base	Termina cuando falla la condición
Se aproxima poco a poco a la terminación	Repetición controlada por contador
Infinita cuando no reduce el problema	Infinita cuando la condición nunca se vuelve falsa
Sobrecarga de llamadas a métodos	

```
function fact(x){
    if (x==0){
        return 1;
    }else{
        return (x*fact(x-1));

    }
}
```

9. STRING

Los strings **son cadenas de caracteres**. Se delimitan con **comillas simples** o **dobles** y, desde la versión **ES6**, se permiten las comillas **invertidas** o *backticks*.

```
let nombre = "Natalia";
let apellido = 'Escrivá';
let nomCompleto = `Natalia Escrivá Núñez`;
let nomCompleto2 = 'Mi nombre es "Natalia Escrivá Núñez"';
```

Como se puede ver, se puede combinar el uso de las diferentes comillas en la misma asignación.

Son un **tipo de dato primitivo, siendo un tipo de objeto nativo que NO depende del navegador.**

9.1. Declaración de Strings

Podemos declarar variables de tipo String de tres formas diferentes:

- Utilizando las comillas, directamente se convierte en un dato de tipo String. Esta es la forma más común.

```
const instituto = "IES SERRA PERENXISA";
```

- Utilizando el constructor predeterminado (String).

```
const ciclo = String('DAW');
```

- Utilizando el constructor de objetos.

```
const nivel = new String("CFGS");
```

Vamos a ver cómo se ven en la consola….

Podemos ver que la 3ª opción se muestra diferente, como un objeto.

```
IES SERRA PERENXISA
DAW
String { "CFGS" }
    0: "C"
    1: "F"
    2: "G"
    3: "S"
    length: 4
  ▶ <prototype>: String { "" }
```

9.2. Template String y concatenación

Los **Template String** son unas plantillas que **admiten el uso del $ dentro del texto** delimitado por las **comillas invertidas, acento grave o *backtick* y después, entre llaves, indicar la expresión o variable de JavaScript**. El texto que está **entre llaves** no se tomará como literal, sino **que se interpreta como si estuviera fuera de las comillas.**

```
let nombre = "Natalia";
```

```
console.log (`Me llamo ${nombre}`);
console.log (`En un día hay ${24*60*60} segundos`);
```

```
Me llamo Natalia
En un día hay 86400 segundos
```

9.2.1. CONCATENAR STRINGS

Podemos **concatenar strings** de tres formas:

Utilizando el **operador de concatenación +**, siempre que estemos utilizando las **comillas dobles o simples**.

```
const nombre = "Natalia",
      apell1 = "Escrivá",
      apell2 = "Núñez";

const nomComp = 'Mi nombre es ' + nombre + ' '+ apell1 + ' ' + apell2;
```

```
Mi nombre es Natalia Escrivá Núñez
```

Utilizando los **Template String**, con las **comillas backtick**

```
const curso = "2 DAW";
const tecnicos = "DWEC, DWES, DIW, DAW";
const comunes = 'EiE e Inglés';
let modulos = "";
    modulos += `En ${curso} cursamos estos módulos:  \n`;
    modulos += ` ${tecnicos} como módulos técnicos `;
   modulos += `y ${comunes} como genéricos`;
console.log(modulos);
```

```
En 2 DAW cursamos estos módulos:
 DWEC, DWES, DIW, DAW como módulos técnicos y EiE e Inglés como genéricos
```

Podemos también utilizar el método **concat(). Se podrá usar con cualquiera de las comillas.**

```
const nombre = "Natalia",
      apell1 = "Escrivá",
      apell2 = "Núñez";
```

```
let nomComp2 = "Me llamo "
console.log(nomComp2.concat(nombre).concat(apell1).concat(apell2));
```

```
Me llamo NataliaEscriváNúñez
```


Además vamos a poder combinar los distintos métodos de concatenación:

```
const curso = "2 DAW";
const tecnicos = "DWEC, DWES, DIW, DAW";
const comunes = 'EiE e Inglés';
let modulos = "";
    modulos += "En " + curso + " cursamos estos módulos:  \n";
    modulos += ` ${tecnicos} como módulos técnicos `;
    modulos += 'y ' + comunes +  ' como genéricos';
console.log(modulos);

let modulos2 = "";
modulos2 = modulos2.concat(`En ${curso} se estudia ${tecnicos} y ${comunes}`);
```

```
En 2 DAW cursamos estos módulos:
 DWEC, DWES, DIW, DAW como módulos técnicos y EiE e Inglés como genéricos
```

```
En 2 DAW se estudia DWEC, DWES, DIW, DAW y EiE e Inglés
```

9.3. Secuencias de escape

Cuando trabajamos con cadenas de caracteres, a veces necesitamos usar algunos que no los podemos escribir.

Como ejemplos, el salto de párrafo (Intro) o el tabulador.

Podremos escribir estos caracteres usando la barra inclinada hacia la izquierda o backslash seguida del carácter concreto.

SEC. ESC	NOMBRE	EJEMPLO
\n	Salto de línea	`console.log ("Mi nombre es:\n" + nomCompleto);`
\t	Tabulador	`console.log ("NomTrabajador\tDIA\tHora Entrada\tHora Salida");`
\r	Retorno de carro	`console.log ("Mi nombre es:\rNatalia\rEscrivá\rNúñez");`
\f	Salto de página	`console.log ("Hoy es:\f24\fdiciembre\fNochebuena");`
\"	Comillas dobles	`console.log ("Y entonces dijo: \"Buena suerte\"");`

\'	Comillas simples	`console.log ('Y entonces dijo: \'Buena suerte\'');`
\\	El carácter backslash	`console.log ('Bienvenidos a: \\2º DAW\\');`

```
Mi nombre es:
Natalia Escrivá Núñez
NomTrabajador   DIA Hora Entrada    Hora Salida
Mi nombre es:NataliaEscriváNúñez
Hoy es:24diciembreNochebuena
Y entonces dijo: "Buena suerte"
Y entonces dijo: 'Buena suerte'
Bienvenidos a: \2º DAW\
```

9.4. Propiedades y métodos de los Strings

9.4.1. PROPIEDAD TAMAÑO

length() → Devuelve el tamaño de una cadena.

```
const text1 = "Supercalifragilisticoespialidoso";
console.log(`La longitud de "${text1}" es: ${text1.length}`);
```

```
La longitud de "Supercalifragilisticoespialidoso" es: 32
```

9.4.2. TYPEOF()

typeof() → Esta función devuelve el tipo de dato de una variable.

```
const text = "Ejemplo de cadena de texto";
console.log(`"${text}" es de tipo: ${typeof(text)}`);
```

```
"Ejemplo de cadena de texto" es de tipo: string
```

9.4.3. MÉTODOS DE BÚSQUEDA

indexOf(<caracter [, inicio]>) → Devuelve la **primera posición** de un carácter en la cadena

```
const text1 = "Supercalifragilisticoespialidoso";
```

```
console.log(`En "${text1}",
la primera r está en la posición:  ${text1.indexOf("r")} `);
```

```
En "Supercalifragilisticoespialidoso",
la primera r está en la posición:  4
```

lastIndexOf(<carácter [, inicio]>) → Devuelve la **última posición** de un carácter en la cadena

```
console.log(`En "${text1}",
la última r está en la posición:  ${text1.lastIndexOf("r")} `);
```

```
En "Supercalifragilisticoespialidoso",
la última r está en la posición:  10
```

El parámetro inicio es opcional, modifica la posición desde donde empieza a buscar.

```
console.log(`En "${text1}",
la segunda r está en la posición:  ${text1.indexOf("r", 15)} `);
```

```
En "Supercalifragilisticoespialidoso",
la segunda r está en la posición:  -1
```

Devolverá -1 si NO se encuentra el carácter dado.

includes(<carácter>) → Devuelve **true o false si se encuentra o no en la cadena**

```
console.log(`En "${text1}",
se encuentra el texto: 'oso' --> ${text1.includes("oso")} `);
```

```
En "Supercalifragilisticoespialidoso",
se encuentra el texto: 'oso' --> true
```

```
console.log(`¿En "${text1}",
podemos encontrar?:
super: ${text1.includes("super")},
fragil: ${text1.includes("fragil")} y
espia: ${text1.includes("espia")}`);
```

```
¿En "Supercalifragilisticoespialidoso",
podemos encontrar?:
super: false,
fragil: true y
espia: true
```

charAt(<num>) → Devuelve un **carácter concreto según la posición**

```
console.log(`El décimo carácter del texto "${text1}" es: ${text1.charAt(9)} `);
```

```
El décimo carácter del texto "Supercalifragilisticoespialidoso" es: f
```


charCodeAt(<num>), [<caracter>] → Devuelve el **código Unicode del carácter (admite posición o carácter)**

```
console.log(`El décimo carácter del texto "${text1}" según Unicode es:
 ${text1.charCodeAt(9)}`);
```

```
El décimo carácter del texto "Supercalifragilisticoespialidoso" según
Unicode es:
 102
```

```
let caracter = "h";
console.log(caracter.charCodeAt(caracter));//104
```

String.fromCharCode(<cód1,..,códn>) → Devuelve un **string formado por los caracteres correspondientes a los códigos Unicode** pasados como argumentos.

```
console.log(String.fromCharCode(65,66,67,100,101,102));
```

```
ABCdef
```

search(<cadena|expresión>) → Nos da **la posición de una cadena o expresión regular en otra cadena**

```
console.log(`En "${text1}",
encontramos 3 palabras escondidas:
Super, en la posición: ${text1.search("Super")},
fragil, en la posición: ${text1.search("fragil")}
y espia, en la posición: ${text1.search("espia")}`);
```

```
En "Supercalifragilisticoespialidoso",
encontramos 3 palabras escondidas:
Super, en la posición: 0,
fragil, en la posición: 9
y espia, en la posición: 21
```

9.4.4. MAYÚSCULAS / MINÚSCULAS

toUpperCase() → Pasa el texto a mayúsculas

```
let text2 = text1.toUpperCase();
console.log(`El texto ${text1} se ha convertido en...
${text2}`);
```

```
El texto Supercalifragilisticoespialidoso se ha convertido en...
 SUPERCALIFRAGILISTICOESPIALIDOSO
```

toLowerCase() → Pasa el texto a minúsculas

```
console.log(`"${text2}" ahora en minúsculas:
${text1.toLowerCase()}`);
```

```
"SUPERCALIFRAGILISTICOESPIALIDOSO" ahora en minúsculas:
 supercalifragilisticoespialidoso
```


9.4.5. EMPIEZA / TERMINA

startsWith(<cadena [, posicion]>) → Devuelve **true si la cadena empieza con el parámetro**

El parámetro posición es opcional, mira si el texto comienza por el primer parámetro, pero empezando a buscar desde la posición dada.

endsWith(<cadena [, tamaño]>) → Devuelve **true si la cadena termina con el parámetro**

El parámetro tamaño es opcional, mira si el texto termina por el primer parámetro, pero dentro del tamaño dado.

```
console.log(`¿"${text1}",
empieza por "Super"?: ${text1.startsWith("Super")},
¿y acaba con "natalia"?: ${text1.endsWith("natalia")}`);
```

```
¿"Supercalifragilisticoespialidoso",
 empieza por "Super"?: true,
 ¿y acaba con "natalia"?: false
```

```
console.log(`¿"${text1}",
empieza por "Super"?: ${text1.startsWith("Super")},
¿y si quitamos los 5 últimos caracteres acaba en "espia"?: ${text1.endsWith("espia", 26)}`);
```

```
¿"Supercalifragilisticoespialidoso",
 empieza por "Super"?: true,
 ¿y si quitamos los 5 últimos caracteres acaba en "espia"?: true
```

9.4.6. EXTRAER / MODIFICAR SUBCADENAS

replace (<textoBuscado, textoReemplazado>) → Busca el 1º parámetro y lo reemplaza por el 2º parámetro

```
console.log(`Sustituye  "oso" por "Super":
${text1.replace("oso","Super")} `);
```

```
Sustituye  "oso" por "Super":
 SupercalifragilisticoespialidSuper
```

trim() → Elimina los espacios en blanco a derecha e izquierda del texto

```
let text3 = "      Natalia Escrivá      ";
console.log(`Eliminamos del texto --${text3}--,
los espacios a derecha e izquierda: --${text3.trim()}--`);
```


```
Eliminamos del texto --     Natalia Escrivá     --,
los espacios a derecha e izquierda: --Natalia Escrivá--}
```

trimStart() → Elimina los espacios en blanco del inicio de la cadena.

```
console.log(`Eliminamos del texto --${text3}--,
los espacios de la izquierda: --${text3.trimStart()}--}`);
```

```
Eliminamos del texto --     Natalia Escrivá     --,
los espacios de la izquierda: --Natalia Escrivá     --}
```

trimEnd() → Elimina los espacios en blanco del final de la cadena.

```
console.log(`Eliminamos del texto --${text3}--,
los espacios de la derecha: --${text3.trimEnd()}--}`);
```

```
Eliminamos del texto --     Natalia Escrivá     --,
los espacios de la derecha: --     Natalia Escrivá--}
```

Estos métodos pueden ser muy útiles para "formatear" las direcciones de correo electrónico en los formularios. Muchas veces se ponen con algún espacio y ya no se pueden validar.

slice(<inicio [, fin]>) → Corta los caracteres desde inicio hasta fin.

Si fin es negativo, empieza a contar por el final.

```
console.log(`Eliminamos de "${text1}" caracteres por delante y por detrás:
 ${text1.slice(5,15)} `);
```

```
console.log(`Eliminamos de "${text1}" caracteres por delante y por detrás:
${text1.slice(5,-15)} `);
```

```
Eliminamos de "Supercalifragilisticoespialidoso" caracteres por delante y por detrás:
 califragil
```

```
Eliminamos de "Supercalifragilisticoespialidoso" caracteres por delante y por detrás:
 califragilis
```

substring(<inicio [, fin]>) → Como el anterior, NO admite negativos

```
console.log(`Eliminamos de "${text1}" caracteres por delante y por detrás:
 ${text1.substring(5,15)} `);
```

```
Eliminamos de "Supercalifragilisticoespialidoso" caracteres por delante y por detrás:
 califragil
```

repeat(<num>) → Repite la cadena el número de veces del parámetro. El parámetro debe ser un natural. Si es un real, JS trunca el número.

```
let texto = "REBAJAS!! ".repeat(3);
console.log(texto)
```

```
REBAJAS!! REBAJAS!! REBAJAS!!
```

```
console.log(`Repite "${text1}" 2 veces:
${text1.repeat(2)} `);
```

```
Repite "Supercalifragilisticoespialidoso" 2 veces:
 SupercalifragilisticoespialidosoSupercalifragilisticoespialidoso
```

split([textoDelim] [, límite]>) → Divide el texto en un array de textos.

El 2º parámetros, pone un límite tope de divisiones.

```
console.log(`Dividimos "${text1}":
 ${text1.split("a")} `);
 console.log(`Dividimos "${text1}":
 ${text1.split("a",2)} `);
```

```
Dividimos "Supercalifragilisticoespialidoso":
 Superc,lifr,gilisticoespi,lidoso
Dividimos "Supercalifragilisticoespialidoso":
 Superc,lifr
```

9.4.7. COMPARAR STRINGS

localeCompare(<texto>,[<"idioma">]) → Compara sin distinguir entre mayúsculas y minúsculas y considerando la forma de ordenar de cada lengua.

Si no usamos el 2º parámetro, se utilizará la configuración local del equipo que ejecuta el código.

```
const palabra1 = "Oso";
const palabra2 = "Ñu";
console.log(palabra1.localeCompare(palabra2));
```

Devuelve 1 porque "ñ" va antes que "o". En caso constrario, devolvería -1.

9.4.8. CONVERTIR A STRING

toString() → Convierte a String

```
const num = 500;
console.log(`${num.toString()}, ${typeof(num)}`);
const numStr = num.toString();
console.log(`Asignando valor con toString(), se ha convertido en ${typeof(numStr)}`);
```

```
500, number
Asignando valor con toString(), se ha convertido en string
```

9.5. Recorrer un String

En ocasiones vamos a necesitar recorrer una cadena para poder trabajar con cada uno de los caracteres que la componen.

Utilizaremos para ello una estructura iterativa → FOR…OF

Ejemplo:

```
const miNombre = "Natalia";
for (let j of miNombre){
    console.log(j)
}
```

```
N
a
t
a
l
i
a
```

10. NUMBER

Los números, al igual que los strings son también datos primitivos de JavaScript.

A diferencia de otros lenguajes, como C, C++ o Java, JavaScript utiliza un único tipo de dato para los números, ya sean valores **enteros** o en **coma flotante**.

Todos los números tienen el mismo espacio en memoria → 64 bits y su tipo de dato es ***Number***.

10.1. Declaración de Numbers

Podemos declarar variables de tipo Number de tres formas diferentes:

- Poniendo directamente el valor. Podemos declarar distintos tipos de números de la misma forma:

```
const num1 = 30;
const num2 = 30.56;
const num3 = .125;
const num4 = -20.5;
```

```
30 30.56 0.125 -20.5
```

Podemos utilizar incluso la **notación científica** para números muy grandes:

```
const grosorPeloHumano = 2e-4;
let fortuna = 7.5e+8;
console.log ("El grosor medio de un cabello humano equivale a " + grosorPeloHumano + "mm");
console.log ("La fortuna del acusado asciende a " + fortuna + "€");
```

```
El grosor medio de un cabello humano equivale a 0.0002mm
La fortuna del acusado asciende a 750000000€
```

- Utilizando el constructor predeterminado (Number).

```
const num5 = Number(-36.95);
```

```
-36.95
```

- Utilizando el constructor de objetos.

```
const num6 = new Number(100);
```

```
Number { 100 }
  ▶ <prototype>: Number { 0 }
```

Usaremos siempre el dato primitivo en vez del constructor integrado porque la ejecución es más rápida

10.2. Infinity y NaN

JavaScript permite utilizar el número **Infinity**, que es superior al **máximo número que puede representar**. Además, es el resultado de la **división de cualquier número entre 0.**

El término Infinity es **reconocido de forma independiente** y se podrá operar con él, siendo cualquier resultado Infinity.

Ejemplos:

```
const num7 = Infinity;
const num8 = 2;
const num9 = -5;
```


```
console.log(`Cualquier número más infinito es: ${num7 + 256.36}`);
console.log("Cualquier número positivo dividido entre 0 es: " + (num8/0));
console.log("Cualquier número negativo dividido entre 0 es: " + (num9/0));
```

```
Cualquier número más infinito es: Infinity
Cualquier número positivo dividido entre 0 es: Infinity
Cualquier número negativo dividido entre 0 es: -Infinity
```

Un término especial relacionado con este tipo de datos es **NaN** *(Not a Number)*. Este valor nos va a indicar que se está intentando operar como números valores que no lo son.

Estos valores nos podrán ayudar a detectar fallos rápidamente, por ejemplo en las interacciones con los usuarios.

Las indeterminaciones matemáticas también dan como resultado NaN.

```
console.log(`Una cadena no numérica operando con números es: ${"Hola"/5}`);
console.log(`Infinity - Infinity es: ${Infinity - Infinity}`);
console.log(`0/0 es: ${0/0}`);
```

```
Una cadena no numérica operando con números es: NaN
Infinity - Infinity es: NaN
0/0 es: NaN
```

10.3. Propiedades de Number

El tipo de dato primitivo Number, tiene una serie de propiedades como son:

- ✓ MAX_VALUE → Devuelve el número más grande que se puede representar o que puede almacenar una variable.
- ✓ MIN_VALUE → Devuelve el número más pequeño que se puede representar o que puede almacenar una variable, es decir, el positivo más cercano a 0.
- ✓ NEGATIVE_INFINITY → Valor para representar infinitos negativos
- ✓ POSITIVE_INFINITY → Valor para representar infinitos positivos
- ✓ NaN → Valor especial "no es número".

```
console.log(`El máximo nº representable en JS es: ${Number.MAX_VALUE}`);
console.log(`El mínimo nº representable en JS es: ${Number.MIN_VALUE}`);
console.log(`Para representar infinitos positivos en JS es: ${Number.POSITIVE_INFINITY}`);
console.log(`Para representar infinitos negativos en JS es: ${Number.NEGATIVE_INFINITY}`);
console.log(`Para representar los Not a Number en JS es: ${Number.NaN}`);
```

```
El máximo nº representable en JS es: 1.7976931348623157e+308
El mínimo nº representable en JS es: 5e-324
Para representar infinitos positivos en JS es: Infinity
Para representar infinitos negativos en JS es: -Infinity
Para representar los Not a Number en JS es: NaN
```

10.4. Métodos/Funciones de Number

METODO / FUNCIÓN	OBJETIVO
typeof(<variable>)	Método común para todos los tipos de datos de JS. Devuelve el tipo de dato de una variable.

```
const edad = 18;
const edad2 = "18";
console.log(`La primera constante es de tipo: ${typeof(edad)} y la segunda es: ${typeof(edad2)}`);
```

```
La primera constante es de tipo: number y la segunda es: string
```

Number(<valor>)	Función que devuelve el valor numérico de un dato. Cambia el tipo de dato a Number

```
console.log(Number(true));
console.log(Number(false));
console.log(Number("25"));
console.log(Number("Natalia"));
```

```
1
0
25
NaN
```

```
const cad1 = "175.36";
const cad11 = Number (cad1);
console.log(`La variable cad1 es de tipo: ${typeof(cad1)} y cad11 es: ${typeof(cad11)}`);
console.log (cad1, cad11);
```

```
La variable cad1 es de tipo: string y cad11 es: number
175.36 175.36
```

```
const cad2 = "1255HOLA";
const cad3 = Number(cad2);
console.log(typeof(cad2));
console.log(typeof(cad3));
console.log(cad3);
```

```
string
number
NaN
```

parseInt(<valor>)	Función que trunca el número parseado. Permite realizar cambios de base. Permite cambiar el tipo de dato a Number.

```
const cad4 = "12HOLA";
const n3 = parseInt(cad4);
console.log(n3);
console.log(typeof(n3));
console.log(parseInt("20.5"));
console.log(parseInt("20 5"));
console.log(parseInt("20hola"));
console.log(parseInt(" hola 5"));
```

```
12
number
20
20
20
NaN
```

parseFloat(<valor>)	Función que devuelve el valor numérico con decimales. Permite cambiar el tipo de dato a Number.

```
const cad7 = "12.25hola";
const n4 = parseFloat(cad7);
console.log(n4);
console.log(typeof(n4));
console.log(parseFloat("20.5"));
console.log(parseFloat("20.25 5.78"));
console.log(parseFloat("20.36hola"));
console.log(parseFloat(" hola.24 5.1"));
```

```
12.25
number
20.5
20.25
20.36
NaN
```

Number.toFixed(<num dec>)	Método que devuelve el valor con el número de **decimales específico**. Redondea Si se asigna un valor a una variable usando este método DEVOLVERÁ UN STRING

```
const numero = 65.9653;
```

```
console.log(`El número ${numero}...
Sin decimales es: ${numero.toFixed(0)}
Con dos decimales es: ${numero.toFixed(2)}
Con siete decimales es: ${numero.toFixed(7)}`);
console.log(`El número ${numero} es de tipo ${typeof(numero)}`);
```

```
El número 65.9653...
Sin decimales es: 66
Con dos decimales es: 65.97
Con siete decimales es: 65.9653000
El número 65.9653 es de tipo number
```

```
let numero2 = numero.toFixed(3);
console.log(`El número ${numero2} ahora es de tipo ${typeof(numero2)}`);
```

```
El número 65.965 ahora es de tipo string
```

```
const cad8 = "2223.256";
let n5 = parseFloat(cad8);
console.log(n5, typeof(n5));
    n5 = parseFloat(cad8).toFixed(2);
console.log(n5, typeof(n5));
```

```
2223.256 number
2223.26 string
```


Number.toPrecision(<num cifras>)	Método que devuelve el valor con el número de **cifras específico**. Redondea. Si se asigna un valor a una variable usando este método DEVOLVERÁ UN STRING

```
console.log(`El número ${numero}...
Con 2 dígitos: ${numero.toPrecision(2)}
Con 4 dígitos es: ${numero.toPrecision(4)}
Con 7 dígitos es: ${numero.toPrecision(7)}`);
console.log(`El número ${numero} es de tipo ${typeof(numero)}`);
```

```
El número 65.9653...
Con 2 dígitos: 66
Con 4 dígitos es: 65.97
Con 7 dígitos es: 65.96530
El número 65.9653 es de tipo number
```

```
const numero3 = numero.toPrecision(3);
console.log(`El número ${numero3} ahora es de tipo ${typeof(numero3)}`);
```

```
El número 66.0 ahora es de tipo string
```

Number.toExponential(<num_ cifras>)	Método que devuelve el valor **en notación científica** con el número de **decimales** específico. Redondea Si se asigna un valor a una variable usando este método DEVOLVERÁ UN STRING

```
let numeroX = 987654321;
console.log(`El número ${numeroX} equivale a ${numeroX.toExponential()}`);
console.log(`El número ${numeroX} equivale a ${numeroX.toExponential(3)}`);
console.log(`El número ${numeroX} es de tipo ${typeof(numeroX)}`);
```

```
El número 987654321 equivale a 9.87654321e+8
El número 987654321 equivale a 9.877e+8
El número 987654321 es de tipo number
```

```
const numeroY = numeroX.toExponential(5);
console.log(`El número ${numeroY} ahora es de tipo ${typeof(numeroY)}`);
```

```
El número 9.87654e+8 ahora es de tipo string
```

Number.isInteger(<valor>)	Método que devuelve true si es entero.

```
const number1 = "15";
const number2 = 1500;
const number3 = -33;
const number4 = 33.5;
console.log(`"${number1}" es un entero??: ${Number.isInteger(number1)}`);
console.log(`${number2} es un entero??: ${Number.isInteger(number2)}`);
console.log(`${number3} es un entero??: ${Number.isInteger(number3)}`);
console.log(`${number4} es un entero??: ${Number.isInteger(number4)}`);
```

```
"15" es un entero??: false
1500 es un entero??: true
-33 es un entero??: true
33.5 es un entero??: false
```

isNaN(<valor>) Number.isNaN(<valor>)	Función que devuelve false o true si el tipo de dato del parámetro es o no un número. Método del objeto Number que devuelve true o false si el parámetro contiene el valor NaN (tipo de dato debe ser Number)

```
let cad9 = "NaN";
console.log(`La cadena "NaN" contiene NaN?? ${Number.isNaN("NaN")}`);//false
console.log(`La cadena "NaN" no es un número?? ${isNaN("NaN")}`);//true
cad9 = Number(cad9);
console.log(`Pasamos "NaN" a Number. Contiene NaN?? ${Number.isNaN(cad9)}`);//true
console.log(`La cadena "NaN" no es un número?? ${isNaN(cad9)}`);//true
```

```
console.log(`${"12hola12"} no es nº: ${isNaN("12hola12")}`);//true
console.log(`${"12hola12"} no es nº: ${Number.isNaN("12hola12")}`);//false
console.log(`${"12hola12"} no es nº: ${Number.isNaN(Number("12hola12"))}`);//true
console.log(`${"12hola12"} no es nº: ${isNaN(parseInt("12hola12"))}`);//false
console.log(`${"12hola12"} no es nº: ${Number.isNaN(parseInt("12hola12"))}`);//false
```

10.5. Números en diferentes bases

JavaScript acepta no sólo los números en notación decimal, también se aceptan **números hexadecimales**. Se les pondrá el **prefijo 0x**

También acepta **números en octal**. Se les pondrá el **prefijo 0o**

Por último, inluimos en este bloque los **números binarios**. Se les pondrá el **prefijo 0b.**

Ejemplos:

```
const base8 = 0o136;
const base2 = 0b1001;
const base16 = 0xABC;
```


```
console.log (`El número expresado en octal es: ${base8}`);
console.log (`El número expresado en binario es: ${base2}`);
console.log (`El número expresado en hexadecimal es: ${base16}`);
```

```
El número expresado en octal es: 94
El número expresado en binario es: 9
El número expresado en hexadecimal es: 2748
```

10.5.1. CAMBIOS DE BASE ENTRE NÚMEROS

Para objetos Number, **el método toString() nos permite cambiar de base 10 a cualquier base.**

Si no se da a toString() una base entre 2 y 36, se lanza una excepción.

Si no se especifica la base, JavaScript asume la predefinida, que es 10.

Ejemplo:

```
const base10 = 136.5;
```

Realizamos las conversiones:

```
console.log (`El número ${base10} corresponde al número decimal: ${base10.toString()}`);
console.log (`El número ${base10} corresponde al número binario: ${base10.toString(2)}`);
console.log (`El número ${base10} corresponde al número en base 9: ${base10.toString(9)}`);
console.log (`El número ${base10} corresponde al número octal: ${base10.toString(8)}`);
console.log (`El número ${base10} corresponde al número hexadecimal: ${base10.toString(16)}`);
```

```
El número 136.5 corresponde al número decimal: 136.5
El número 136.5 corresponde al número binario: 10001000.1
El número 136.5 corresponde al número en base 9: 161.444444444444444
El número 136.5 corresponde al número octal: 210.4
El número 136.5 corresponde al número hexadecimal: 88.8
```

Para objetos Number, **el método parseInt() nos permite cambiar de cualquier base a decimal.**

Ejemplos:

```
console.log(`El número "BC8" en hexadecimal es:      ${parseInt("BC8",16)} en base 10`);
console.log(`El número "5710" en octal es:           ${parseInt(5710,8)} en base 10`);
console.log(`El número "101111001000" en binario es: ${parseInt(101111001000,2)} en base 10`);
console.log(`El número "161.44" en base 9 es: ${parseInt(161.44,9)} en base 10`);
```

```
El número "BC8" en hexadecimal es:      3016 en base 10
El número "5710" en octal es:           3016 en base 10
El número "101111001000" en binario es: 3016 en base 10
El número "161.44" en base 9 es:        136 en base 10
```


11. BOOLEAN

Valores que únicamente almacenan dos valores → verdadero o falso (*true o false*).

Estos valores son palabras reservadas que pueden asignarse directamente o bien estos valores se producen al evaluar una expresión lógica.

```
const bool1 = true;
const bool2 = false;
const bool3 = "true";
console.log(`La variable bool1 es de tipo ${typeof(bool1)}`);
console.log(`La variable bool2 es de tipo ${typeof(bool2)}`);
console.log(`La variable bool3 es de tipo ${typeof(bool3)}`);
```

```
La variable bool1 es de tipo boolean
La variable bool2 es de tipo boolean
La variable bool3 es de tipo string
```

También podemos crear elementos de tipo Boolean a través del constructor. **Ejemplo**:

```
const bool4  = new Boolean(false);
console.log(`La variable bool4 es de tipo ${typeof(bool4)}`);
```

```
La variable bool4 es de tipo object
```

Para comprobar **que toda expresión y valor en JavaScript se asocia con valores booleanos**, usaremos la **función Boolean**:

```
const x = 3, y = 8, w = true, z = (x > y);
```

```
console.log (Boolean(z));//false
console.log (Boolean(w));//true
console.log (Boolean(1));//true
console.log (Boolean(0));//false
console.log (Boolean("cadena"));//true
console.log (Boolean(""));//false
console.log (Boolean(NaN));//false
console.log (Boolean(undefined));//false
console.log (Boolean(Infinity));//true
console.log (Boolean(null));//false
```

Si lo ejecutamos, veremos que únicamente **se considera falso** el cero, las comillas vacías, NaN, undefined y null.

A esta lista le añadiremos las expresiones cuya valoración lógica NO sea verdadera.

12. OPERADORES

Los operadores son elementos que nos van a permitir realizar estructuras más complejas y cálculos. Existen de diferentes tipos para poder utilizarlos con diferentes tipos de datos.

12.1. Operadores aritméticos

Los operadores aritméticos permiten realizar cálculos mátemáticos sobre variables de tipo Number.

Los **operadores de números** en JavaScript se dividen en dos grupos:

→ **Binarios**: Requieren de dos operandos (suma, resta, producto, potencia, división y módulo o resto).

→ **Unarios**: Se aplican sobre un único operando (incremento y decremento)

```
let x = 5;
let y = 2;
```

OPERADOR	NOMBRE	EJEMPLO
+	Suma	`console.log ("La suma de x e y es " + (x+y));` `La suma de x e y es 7`
-	Resta	`console.log ("7 menos y es " + (7-y));` `7 menos y es 5`
*	Multiplicación	`console.log ("x multiplicado por 4 es " + (x*4));` `x multiplicado por 4 es 20`
**	Potencia	`console.log ("Hallamos la potencia de 3^2: " + (3**2));` `Hallamos la potencia de 3^2: 9`
/	División	`console.log ("Dividimos x entre y: " + (x/y));` `Dividimos x entre y: 2.5`
%	Módulo o resto	`console.log ("El resto de 10 entre 2 es: " + (10%2));` `El resto de 10 entre 2 es: 0`

12.2. Operadores de asignación

Estos operadores nos sirven para operar a la vez que asignan el valor de la operación.

Podemos ver que los **operadores de incremento y decremento** se pueden usar antes o después de la variable: pre o post incremento/decremento.

El comportamiento NO es el mismo, por lo que deberemos prestar atención a esto: Si **mostramos** variables con el incremento o decremento en el modo POST, DEBEREMOS LLAMAR A LA VARIABLE DE NUEVO PARA OPERAR CON SU NUEVO VALOR.

OPERADOR	SIGNIFICADO	EJEMPLO

var++	Post Incremento	`console.log(x); //5`
		`console.log(x++); //5`
		`console.log(x); //x vale 6`
		`//Solución`
		`console.log(q); //5`
		`q++; //equivalente a q = q + 1`
		`console.log(q); //6`
		Salida: 5, 5, 6, 5, 6
		`let x = 5;`
		`let y = 2;`

++var	Pre incremento	`console.log(w); //5`
		`console.log(++w); //6`
		`console.log(w); //6`
		Salida: 5, 6, 6

var--	Post decremento	`console.log(x);`
		`console.log(x--);`
		`console.log("Post Decremento --> x vale: " + x);`
		Salida: 12, 12, Post Decremento --> x vale: 11

--var	Pre decremento	`console.log(x);`
		`console.log(--x);`
		`console.log("Pre Decremento --> x vale: " + x);`
		Salida: 11, 10, Pre Decremento --> x vale: 10

+=	Suma y asignación	`let z = 20;` `console.log (z+=5); // z vale 25. Es equivalente a z = z+5;`

-=	Resta y asignación	`console.log (z-=5); // z vale 20. Es equivalente a z = z-5;`
***=**	Producto y asignación	`console.log (z*=2); // z vale 40. Es equivalente a z = z*2;`
/=	División y asignación	`console.log (z/=2); // z vale 20. Es equivalente a z = z/2;`
%=	Resto y asignación	`console.log (z%=3); // z vale 2. Es equivalente a z = z%3;`
****=**	Potencia y asignación	`console.log (z**5); // z vale 32. Es equivalente a z = z**5;`

12.3. Operadores Relacionales

Estos operadores nos van a permitir comparar dos expresiones y devolverán un valor booleano.

En cuanto a la comparación de **strings** debemos saber que distinque entre mayúsculas y minúsculas y se rige por el orden que tienen los caracteres en la table Unicode.

OPERADOR	SIGNIFICADO	EJEMPLO

>	**Mayor que**	`console.log("Analizamos 5 mayor que 1: " + (5 > 1) );`
		`Analizamos 5 mayor que 1: true`
		`console.log ("perro" > "perra"); // true`
		`console.log ("Perro" > "perra"); // false`

<	**Menor que**	`console.log("Analizamos '15' menor que 7: " + ("15" < 7) );` `Analizamos '15' menor que 7: false`
>=	**Mayor o igual que**	`console.log('Analizamos "33.33" mayor o igual a 50: ' + ("33.33" >= 50) );` `Analizamos "33.33" mayor o igual a 50: false`

<=	Menor o igual que	`console.log('Analizamos 150 menor o igual a 160: ' + (150 <= 160) );` `Analizamos 150 menor o igual a 160: true`

==	Comparador de Igualdad	`console.log('Analizamos 65 igual a "65": ' + (65 == "65") );`
		`Analizamos 65 igual a "65": true`
		`console.log ("Comparamos 1 es igual a true: " + (true == 1));`
		`console.log ("Comparamos undefined es igual a null: " + (undefined == null));`
		`Comparamos número (1 o 0) con booleano: true`
		`Comparamos undefined y null: true`
		`console.log ("perro" == "perra"); // false`
		`console.log("true" == true);//false`

===	Comparador ESTRICTO** de Igualdad	`console.log (`Comparación estricta de una cadena y un número: ${"8" === 8}`);`
		`console.log (`Comparación estricta de dos valores numéricos: ${7+1 === 8}`);`
		`console.log ("Comparación estricta undefined y null: " + (undefined === null));`
		`Comparación estricta de una cadena y un número: false`
		`Comparación estricta de dos valores numéricos: true`
		`Comparación estricta undefined y null: false`

!=	Comparador de Desigualdad	`console.log ("Analizamos -36 distinto de 36: " + (-36 != 36));` `Analizamos -36 distinto de 36: true`
!==	Comparador ESTRICTO** de Desigualdad	`console.log (`Verificamos la desigualdad estricta entre una cadena y un número: ${"8" !== 8}`);` `Verificamos la desigualdad estricta entre una cadena y un número: true`

****La diferencia entre la igualdad y la igualdad estricta es que la estricta**

TIENE EN CUENTA EL TIPO DE DATO, NO SOLO EL VALOR!!

12.4. Operadores Lógicos

Los operadores lógicos permiten evaluar dos condiciones o expresiones lógicas para poder tomar decisiones en la ejecución de nuestro programa.

Ejemplos:

```
const a = 5, b = 2;
```

OPERADOR	SIGNIFICADO	EJEMPLO

AND **&&**	**True si las dos expresiones son verdaderas**	`console.log(a == 5 && b == 2); //true`
		`console.log((a == 5) && (b == 8)); //false`
		`const edad = 17;`
		`const conducir = (edad >= 18 && carnet == true);`
		`console.log (conducir); //false`

OR **||**	**True si cualquiera de las dos expresiones es verdadera**	`console.log(a == 5 || b == 2); //true`
		`console.log((a == 5) || (b == 8)); //true`
		`console.log((a == 8) || (b == 3)); //false`

NOT **!**	**Niega la expresión que tenga a su derecha**	`console.log(!(a == 5 && b == 2)); //false`
		`console.log( a == 5 && !(b == 2)); //false`
		`console.log((!(a == 3) && (b == 2))); //true`

13. CONVERSIÓN DE TIPOS

Sabemos que JavaScript **es un lenguaje de tipado débil**. Esto implica que no tiene por qué dar error cuando se trabaja con distintos tipos de valores.

Ya hemos visto que, dependiendo de los tipos o sus valores, se pueden utilizar operadores de comparación entre ellos. También podríamos operar con ellos.

13.1. Conversión Automática

Para poder trabajar con valores de diferentes tipos, lo que hace JavaScript es realizar conversiones automáticas de tipos de datos, siempre que sea posible.

Si utilizamos operadores aritméticos con números y strings (cuya cadena esté formada por números), JavaScript será capaz de operar con esos valores.

Hay combinaciones que no se pueden convertir y, aunque **no veamos un error**, el intérprete lo califica como NaN porque no puede hacer la conversión y trabajar con esos valores.

Ejemplos:

```
console.log ('5' * 5); // 25
console.log ('5' + 5); // 55
console.log ('Natalia' / 2); // Nan
console.log (null * 25); //  0
console.log (true + 6); // 7
console.log (false * 2);// 0
console.log (undefined ** 2); // NaN
console.log (7 + 3 + " Natalia "); //10 Natalia
console.log ("Natalia " + 7 + 3); //Natalia 73
```

Si el operador es una suma y el intérprete detecta un string, lo que hará será concatenar y no sumar.

OJO!! Según la precedencia de los operadores, el intérprete convertirá de una manera u otra los datos.

Fíjate en los dos ejemplos finales!!!!

13.2. Conversión con métodos/funciones

De la misma forma que hemos visto que hay maneras de realizar comparaciones o analizar desigualdades de forma estricta, también tenemos la posibilidad de forzar la conversión de los números.

Estas conversiones se podrán utilizar en el momento concreto según sea la necesidad de la aplicación o bien definitiva, a través de la asignación.

Recordamos distintos métodos y funciones para realizar estas conversiones de tipos:

→ **Number**: Convertiremos cadenas cuyo contenido son números en tipos de datos de tipo numérico.

```
//Forzar conversiones --> Number
let x = 5; y = "2.25";
let z = Number(y);
console.log (x + y); // 52.25
console.log (Number(x) + Number(y)); // 7.25
console.log(typeof (y));//String
console.log(typeof(z));//Number
```

→ **String /toString**: La usaremos para convertir a String

```
//Forzar conversiones --> String
let w = 8; z = 1;
console.log (w + z); // 9
console.log (String(w) + String(z)); // 81
console.log(typeof (w));//Number
```

→ **ParseInt**: La podremos utilizar para realizar cambios a enteros. Permite cambiar de base y pasar a decimal. Además, si el string empieza por un número, lo puede convertir. Trunca el número si éste tiene decimales.

```
//Forzar conversiones --> ParseInt
console.log (parseInt("1001",2)); //El 2 es la base --> Da 9
console.log (parseInt("2987jhjklhjh2211")); // Si está en base 10, no
    //hace falta el segundo parámetro --> Da 2987
console.log (parseInt("10.25veces")); // 10 pq convierte a entero
```

→ **ParseFloat**: Para convertir a decimales.

```
//Forzar conversiones --> ParseFloat
console.log (parseFloat("10.25veces")); // 10.25
```