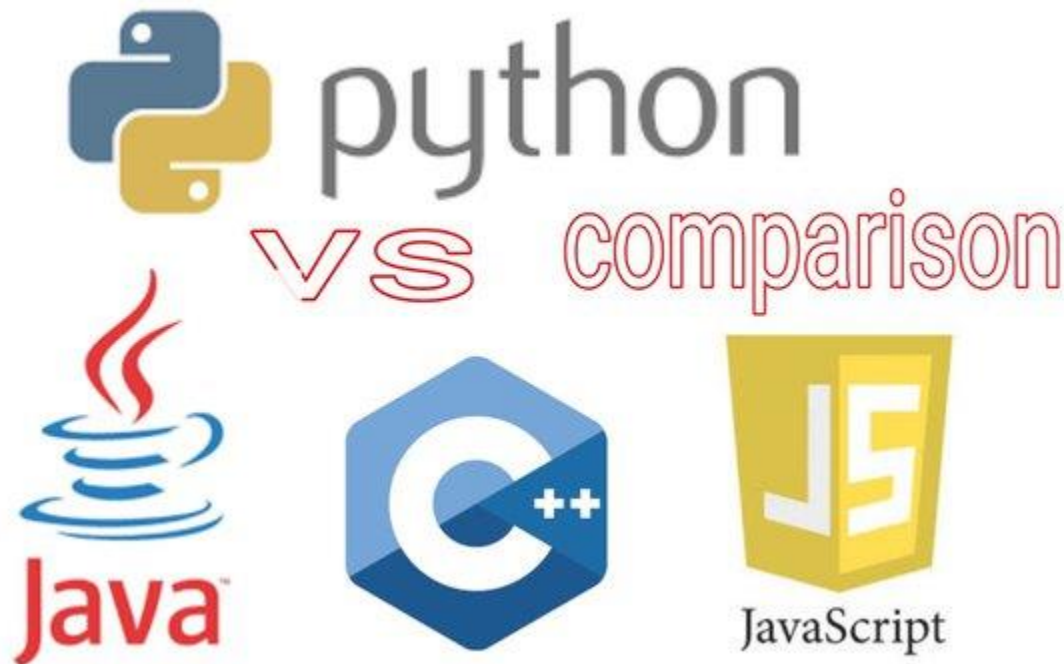


Unidad 2

Introducción a la programación en JavaScript



La sintaxis de JavaScript es muy parecida a Java y C++. Cualquier programador que sepa programar en Java, PHP u otro lenguaje con sintaxis similar será capaz de comprender la sintaxis de JavaScript de una manera rápida. No obstante, al ser JavaScript un lenguaje de programación del lado del cliente, es importante conocer sus aspectos básicos como pueden ser los eventos, el control de los elementos HTML, etc.



2. 10 reglas básicas del JavaScript

- *Regla 1.* Las instrucciones en JavaScript terminan en un punto y coma. Ejemplo:

```
var s = "hola";
```

- *Regla 2.* Uso de decimales en JavaScript. Los números en JavaScript que tengan decimales utilizarán el punto como separador de las unidades con la parte decimal. Ejemplos de números:

```
var x = 4;  
var pi = 3.14;
```

- *Regla 3.* Los literales se pueden escribir entre comillas dobles o simples. Ejemplo:

```
var s1 = "hola";  
var s2 = 'hola';
```

- *Regla 4.* Cuando sea necesario declarar una variable, se utilizará la palabra reservada *var*.
- *Regla 5.* El operador de asignación, al igual que en la mayoría de lenguajes, es el símbolo igual (=).

2. 10 reglas básicas del JavaScript

- *Regla 6.* Se pueden utilizar los siguientes operadores aritméticos: (+ - * /). Ejemplo:

```
var x = (5*4)/2+1;
```

- *Regla 7.* En las expresiones, también se pueden utilizar variables. Ejemplo:

```
var t = 4;  
var x = (5*t)/2+1;  
var y;  
y = x * 2;
```

- *Regla 8.* Comentarios en JavaScript.
- *Regla 9.* Los identificadores en JavaScript comienzan por una letra o la barra baja (_) o el símbolo del dólar (\$).
- *Regla 10.* JavaScript es sensible a las mayúsculas y minúsculas (case-sensitive). Ejemplo:

```
var nombre = "Julio";  
var Nombre = "Ramón";
```

- *Nombre y nombre* son dos variables diferentes.
- Ten cuidado al escribir la palabra reservada *var*. Si escribes *Var* o *VAR*, tu código no funcionará.

3. Palabras reservadas de JavaScript

Las más utilizadas son:

Palabra	Descripción
<code>var</code>	Utilizada para declarar una variable.
<code>if ... else</code>	Estructura condicional.
<code>for</code>	Estructura de repetición. Se ejecutará mientras la condición sea verdadera.
<code>do ... while</code>	Estructura de repetición. Se ejecutará mientras la condición sea verdadera.
<code>switch</code>	Serie de sentencias que van a ser ejecutadas dependiendo de diferentes circunstancias.
<code>break</code>	Termina un switch o un bucle.
<code>continue</code>	Sale del bucle y se coloca al comienzo de este.
<code>function</code>	Declara una función.
<code>return</code>	Sale de una función.
<code>try ... catch</code>	Utilizadas para el manejo de excepciones.

Listado completo de palabras reservadas en JS

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Lexical_grammar#palabras_clave

Para definir identificadores en JavaScript hay que asegurar de no saltarse ninguna de las siguientes reglas:

- Debe empezar obligatoriamente por una letra, guion bajo (_) o el símbolo del dólar (\$).
- Pueden seguirle más letras, números o guiones bajos.
- Distingue entre mayúsculas y minúsculas.
- Puede usar todas las letras que están definidas en UNICODE.

Es un error muy común no tener en cuenta que JavaScript distingue entre mayúsculas y minúsculas en los identificadores. No hay que olvidar que **Mielemento**, **miElemento**, **Mielemento** y **MIELEMENTO** son cuatro identificadores completamente distintos para JavaScript.

Existen 3 tipos de comentarios en JavaScript:

1. **COMENTARIOS EN LÍNEA** – se utiliza `//` para iniciar el comentario. Se considera comentario todo lo que aparezca a continuación y hasta el final de la línea.

```
// Controla el primer bucle (número de iteraciones)
var contador = 0;
// Indica si el recurso está libre (true) u ocupado (false)
var recurso = true;
```

2. **COMENTARIOS DE VARIAS LÍNEAS o COMENTARIOS DE BLOQUES** – se utiliza `/*comentario*/` se utiliza para comentarios más extensos

```
/*
    Almacena la cantidad de elementos actuales de la estructura.
    Solo se incluyen los positivos.
    Debe coincidir con los dispositivos válidos.
*/
let numElementos = 0;
```

3. **COMENTARIOS HASHBANG** – comentarios especiales, se utilizan para indicar la ruta a un motor de JavaScript específico que debe ejecutar el script. Se utiliza `#!`

```
#!/usr/bin/env node
```

VARIABLE – posición de memoria a la que se le asigna un nombre y en la que se guarda un valor.

El nombre de la variable permitirá a JS poder ubicar y localizar dicho espacio cuando interprete los scripts.

En JavaScript, se pueden ejecutar las siguientes órdenes:

```
var x;  
x = 2 * x + 1;  
var pi = 3.141592;  
var paginaweb = "Myfpschool";  
var pregunta = '¿cuantos años tienes?', respuesta="veinte";  
paginaweb="Myfpschool.com";  
paginaweb="Myfpschool" + "." + "com";
```

El resultado a la derecha de una asignación se almacena en la variable del lado izquierdo de esta.

Todas las variables son plenamente funcionales cuando se les asocia un valor y todos los valores pertenecen a un tipo concreto.

En JavaScript tenemos los siguientes tipos de datos:

1. STRING
2. NUMBER
3. BOOLEAN
4. ARRAY (veremos posteriormente)
5. OBJECT (veremos posteriormente)

```
var edad = 25; // Number
var nombre = "Dimas"; // String
var asignatura = ["lengua", "mate", "cono"]; // Array
var persona = {nombre:"Dimas", apellido:"Moreno"}; // Objeto
```

1. TIPO DE DATOS STRING

Este tipo de datos es una secuencia de caracteres (texto). Hay 3 formas de escribir una cadena de texto en JS:

```
miString_1 = "con comillas dobles";
miString_2 = 'con comillas simples';
miString_3 = `con comillas invertidas o backticks`;
```

Además, se puede utilizar el operador de concatenación (+) para unir fragmentos de cadenas o incluir el contenido de unas variables dentro de ellas:

```
subcadena_1 = "Primera parte, ";  
subcadena_2 = "parte concatenada";  
subcadena_3 = ". Última parte añadida";  
cadenaFinal = "Cadena: " + subcadena_1 + subcadena_2 + subcadena_3 + ".";
```

En este ejemplo **cadenaFinal** contendría la cadena:

```
"Cadena: Primera parte, parte concatenada. Última parte añadida. "
```

La tercera forma de expresar las cadenas de caracteres simplifica el proceso de concatenación de cadenas. Los *backticks* permiten incluir expresiones sin tener que utilizar el operador +. Para ello, se incluye la expresión dentro de las llaves de **\${}.**

Por ejemplo, podría obtenerse la misma **cadenaFinal** anterior al hacer esto:

```
cadenaFinal = `Cadena: ${subcadena_1}${subcadena_2}${subcadena_3}.`;
```

SECUENCIAS DE ESCAPE - conjunto de caracteres que no se pueden escribir, pero que son fundamentales en algunas situaciones.

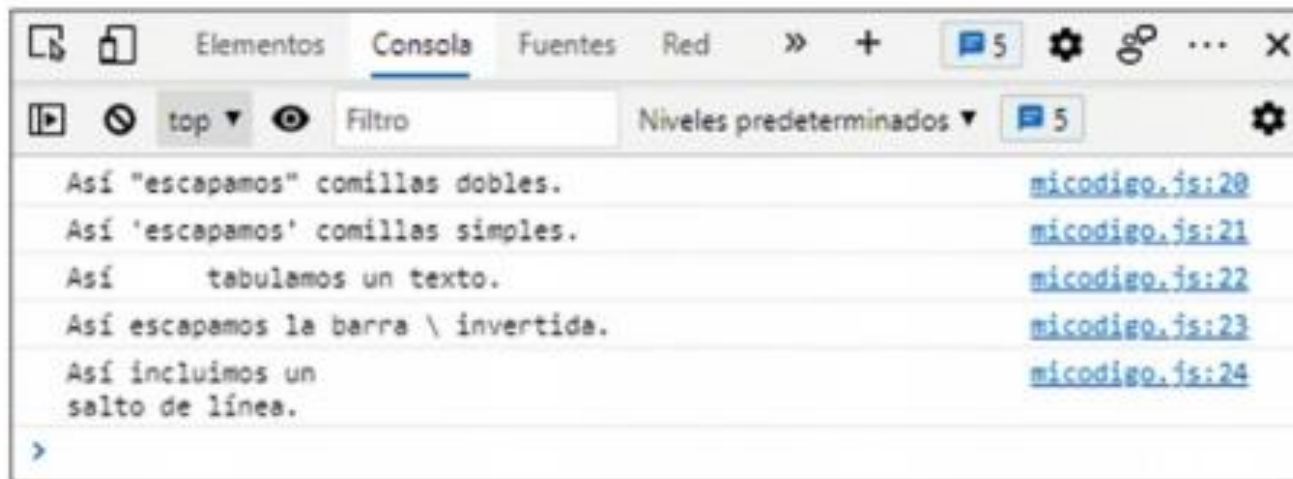
Las secuencias de escape más utilizadas son:

Secuencia	Uso
\'	Comillas simples
\"	Comillas dobles
\\	Barra invertida o <i>backslash</i>
\b	Retroceso
\f	Salto de página
\n	Salto de línea
\r	Retorno de carro
\t	Tabulador horizontal
\v	Tabulador vertical

Ejemplos de uso de las secuencias de escape:

```
console.log("Así \"escapamos\" comillas dobles.");  
console.log('Así \'escapamos\' comillas simples.');
```

```
console.log("Así \t tabulamos un texto.");  
console.log("Así escapamos la barra \\ invertida.");  
console.log("Así incluimos un\\nsalto de línea. ");
```



Además, utilizando el patrón `\u{código}`, siendo *código* un valor hexadecimal de hasta ocho dígitos (32 bits), se puede representar cualquier carácter o símbolo de la tabla UNICODE. Así, si se quiere incluir en la cadena el símbolo del Bitcoin (₿) solo habría que escribir `\u{20BF}`.

2. TIPO DE DATOS NUMBER

Este tipo de datos hace referencia a un único tipo de dato numérico que recoge cualquier formato de número con o sin decimales.

Para expresar un número, se escribe sin indicar ningún tipo de símbolo adicional. Si quiere indicarse que se trata de un número decimal se utiliza el punto (.) como separador decimal. Y si quiere indicarse un exponente se utiliza la **e** y el signo del exponente:

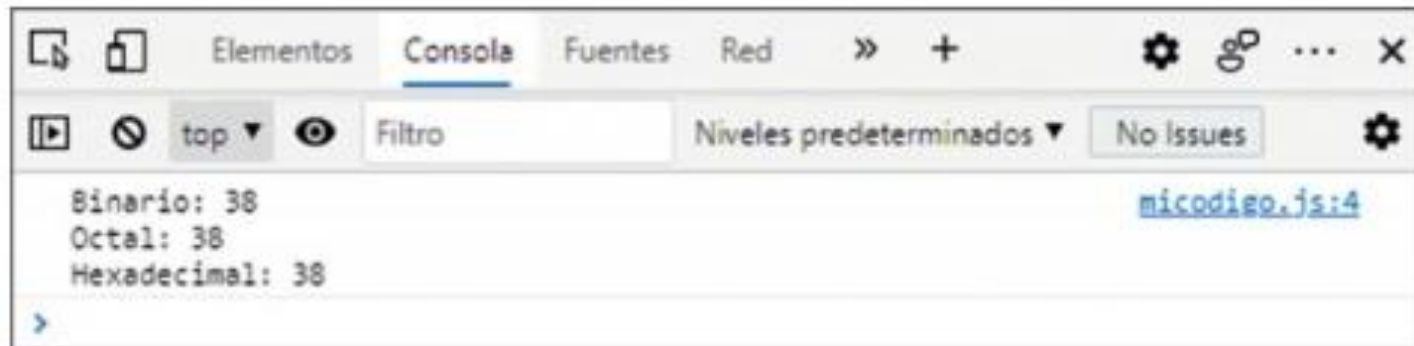
```
alturaCm = 182;  
pesoKg = 71.6;  
diametroTransistor = 2e-9;
```

Así, se estaría almacenando 182 cm, 71'6 kg y $2 \cdot 10^{-9}$ m.

7. Tipos de datos

Si queremos decirle al motor de JS que queremos utilizar un sistema diferente del decimal, hay que escribir, delante del número el dígito 0 seguido por un carácter de identificación: binario (**b**), octal (**o**) o hexadecimal (**x**).

```
numeroBinario = 0b100110;  
numeroOctal = 0o46;  
numeroHexadecimal = 0x26;  
console.log(`Binario: ${numeroBinario}\nOctal: ${numeroOctal}\nHexadecimal: ${numeroHexadecimal}`);
```

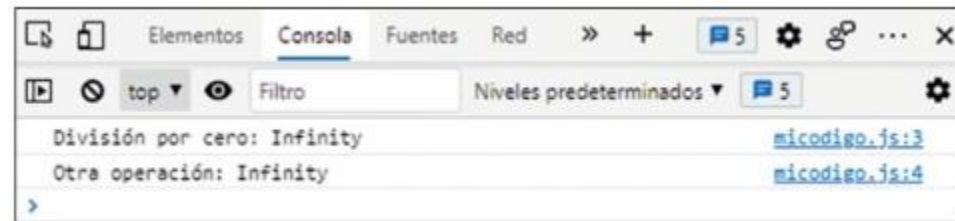


Al observar la salida, vemos que la consola ha convertido todos los números al sistema decimal, puesto que está configurada por defecto para trabajar con ese sistema.

CURIOSIDADES DE JAVASCRIPT:

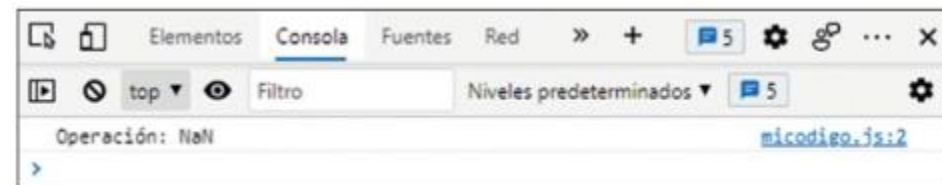
- 1) Es capaz de entender y operar con el valor infinito (por ejemplo, con una división por 0) devolviendo **Infinity**

```
divisionPorCero = 23/0;  
otraOperacion = divisionPorCero + 12;  
console.log(`División por cero: ${divisionPorCero}`);  
console.log(`Otra operación: ${otraOperacion}`);
```



- 2) Es capaz de operar con tipos de datos distintos, devolviendo **NaN** (Not a Number)

```
operacion = 54 * "cadena de texto";  
console.log(`Operación: ${operacion}`);
```



Estos dos casos, facilitan notablemente la validación de errores y su localización.

3. TIPO DE DATOS BOOLEANO

BOOLEANO – dato lógico que solo puede tomar 2 valores true (verdadero) o false (falso).

Al evaluar expresiones lógicas se considera:

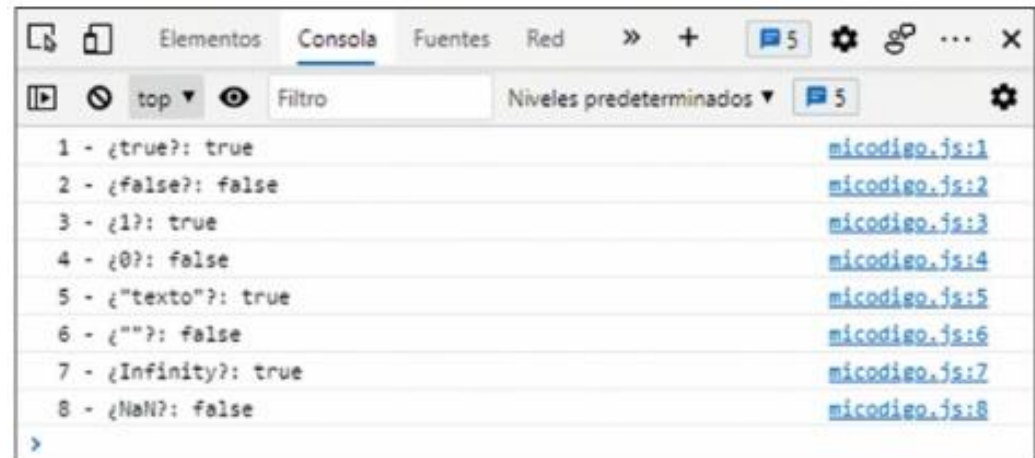
- Todo lo que sea **igual a cero** es **FALSO**
- Todo lo que sea **distinto de cero** es **VERDADERO**

Usando la función Boolean() podemos hacer las comprobaciones:

```
console.log(`1 - ¿true?: ${Boolean(true)}`);  
console.log(`2 - ¿false?: ${Boolean(false)}`);  
console.log(`3 - ¿1?: ${Boolean(1)}`);  
console.log(`4 - ¿0?: ${Boolean(0)}`);  
console.log(`5 - ¿"texto"?: ${Boolean("texto")}`);  
console.log(`6 - ¿"": ${Boolean("")}`);  
console.log(`7 - ¿Infinity?: ${Boolean(Infinity)}`);  
console.log(`8 - ¿NaN?: ${Boolean(NaN)}`);
```

Función Boolean()

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Boolean



```
1 - ¿true?: true micodigo.js:1  
2 - ¿false?: false micodigo.js:2  
3 - ¿1?: true micodigo.js:3  
4 - ¿0?: false micodigo.js:4  
5 - ¿"texto"? : true micodigo.js:5  
6 - ¿"": false micodigo.js:6  
7 - ¿Infinity?: true micodigo.js:7  
8 - ¿NaN?: false micodigo.js:8
```

OTROS TIPOS DE DATOS

UNDEFINED – valor que se obtiene cuando todavía no se le ha asignado un valor a una variable y cuando no es capaz de evaluar una expresión.

`Boolean(undefined)` devuelve `false`.

NULL – en JS representa el valor vacío o nulo, de forma intencionada.

`Boolean(null)` devuelve `false`.

BigInt – para números enteros muy grandes

Tipos de datos en JS

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

8. Conversión entre tipos de datos

JS es un lenguaje débilmente tipado, es decir, que es capaz de realizar operaciones donde se mezclan tipos de datos distintos sin lanzar errores constantemente.

Esto se debe a que tiene una potente capacidad para convertir unos tipos de datos en otros de forma dinámica.

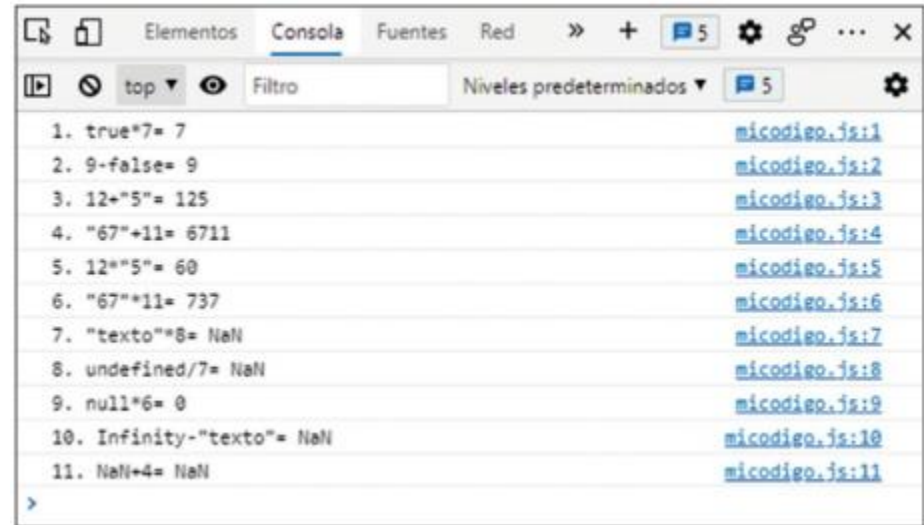
La conversión de tipos en JS se puede realizar de dos formas: Automáticamente o Manualmente.



8. Conversión entre tipos de datos

CONVERSIÓN DE TIPOS AUTOMÁTICA

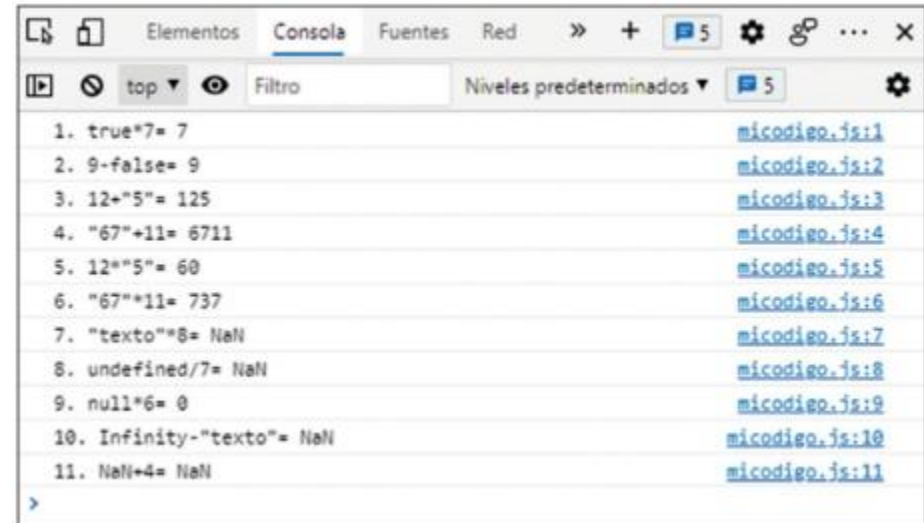
```
console.log(`1. true*7= ${true*7}`);  
console.log(`2. 9-false= ${9-false}`);  
console.log(`3. 12+"5"= ${12+"5"}`);  
console.log(`4. "67"+11= ${"67"+11}`);  
console.log(`5. 12*"5"= ${12*"5"}`);  
console.log(`6. "67"*11= ${"67"*11}`);  
console.log(`7. "texto"*8= ${"texto"*8}`);  
console.log(`8. undefined/7= ${undefined/7}`);  
console.log(`9. null*6= ${null*6}`);  
console.log(`10. Infinity-"texto"= ${Infinity-"texto"}`);  
console.log(`11. NaN+4= ${NaN+4}`);
```



1. El valor booleano **true** se considera que es toda aquella expresión que se evalúa a distinto de cero. Desde el punto de vista numérico se entiende que **true** es 1 y **false** es 0. Por tanto, se sustituye **true** por 1 y se multiplica.
2. Siguiendo el mismo razonamiento anterior, se sustituye **false** por 0 y se resta.
3. Se intenta sumar dos números, pero uno de ellos está encerrado entre comillas dobles, por tanto, es un *string*. En el contexto de los *strings* se había estudiado que se utiliza el operador **+** para concatenar cadenas. Y esa es precisamente la lógica que se ha seguido, convertir el 12 a cadena de caracteres y concatenarlas.
4. Siguiendo el mismo razonamiento anterior, se ha convertido el 11 en una cadena de caracteres y los ha concatenado.
5. En este caso el operador es *****; por tanto, y a diferencia de los dos casos anteriores, solo cabría pensar que lo que se desea es multiplicar. Se convierte la cadena de texto "5" en el número 5 y se multiplica.

8. Conversión entre tipos de datos

```
console.log(`1. true*7= ${true*7}`);  
console.log(`2. 9-false= ${9-false}`);  
console.log(`3. 12+"5"= ${12+"5"}`);  
console.log(`4. "67"+11= ${"67"+11}`);  
console.log(`5. 12*"5"= ${12*"5"}`);  
console.log(`6. "67"*11= ${"67"*11}`);  
console.log(`7. "texto"*8= ${"texto"*8}`);  
console.log(`8. undefined/7= ${undefined/7}`);  
console.log(`9. null*6= ${null*6}`);  
console.log(`10. Infinity-"texto"= ${Infinity-"texto"}`);  
console.log(`11. NaN+4= ${NaN+4}`);
```



6. Siguiendo el mismo razonamiento anterior, se convierte "67" en 67 y se multiplica.
7. Una vez más, el mismo caso que los dos anteriores, salvo por el detalle de que la cadena de texto no es un número. No es posible realizar operaciones matemáticas cuando una de las partes no es un número. Y así se indica en el resultado.
8. Siguiendo el mismo razonamiento anterior, no puede hacerse una operación matemática cuando una de las partes no está definida.
9. **Null** se habría definido como la ausencia intencionada de valor por parte del programador. Parece razonable que se considere 0 cuando interviene en una operación matemática.
10. Se trata del mismo caso que el explicado en el punto 7; se intenta operar un número muy grande con una cadena de caracteres.
11. Resultado evidente, toda vez que se intenta realizar una operación matemática sabiendo que una de las partes indica explícitamente que no es un número.

8. Conversión entre tipos de datos

CONVERSIÓN MANUAL DE TIPOS

- 1) Forzar para que + sume y no concatene – Convirtiendo el string a número con la función Number()

```
console.log(`3. 12+"5"= ${12+Number("5")}`);  
console.log(`4. "67"+11= ${Number("67")+11}`);
```

- 2) Para que dos números se concatenen, en vez de que se sumen – Convirtiendo el número a string con la función String()

```
console.log(`6+7 Sin forzar: ${6+7}`);  
console.log(`6+7 Forzando: ${String(6)+String(7)}`);
```

8. Conversión entre tipos de datos

FUNCIONES DE CONVERSIÓN:

- 1) **String()** – convierte un número a string (cadena)
- 2) **Number()** – convierte una cadena en número
- 3) **parseInt()** – convierte una cadena a un número entero
- 4) **parseFloat()** – convierte una cadena a un número decimal

```
alert("Entero: " + parseInt("9.99")); //mostrará Entero: 9  
alert("Float: " + parseFloat("9.99")); //mostrará Float: 9.99
```

- 5) **toDateString()** – convierte fechas a string

```
var fecha = new Date();  
var cadena = fecha.toDateString(); // Ahora cadena contendrá la fecha  
actual "Sun, 13 Jan 2019".
```

- 6) **toUTCString()** – convertir la hora a formato UTC (hora universal coordinada)

```
var fecha = new Date();  
var cadena = fecha.toUTCString(); // Ahora cadena contendrá la fecha ac-  
tual en formato UTC "Sun, 13 Jan 2019 07:45:49 GMT".
```

9. Declaración e inicialización de variables

En JS se pueden declarar las variables de 3 formas, con **var**, **let** y **const**.

1) VAR

Es la forma tradicional de declarar una variable:

```
var miVariable;
```

Si queremos asignar un valor a dicha variable:

```
miVariable = 8;
```

Sólo es necesario declarar las variables una vez. Después pueden utilizarse tantas veces como se quiera simplemente usando su identificador.

El ámbito de uso de estas variables es el contexto de ejecución en el que se encuentra la palabra **var**.

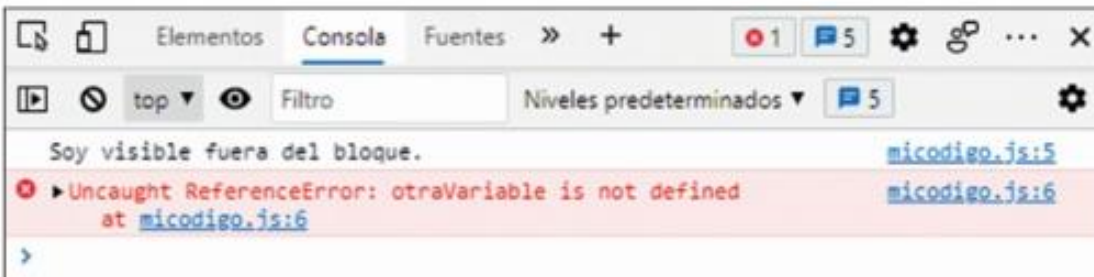
9. Declaración e inicialización de variables

2) LET

La gran diferencia entre `let` y `var` es que el ámbito de `let` es más local que el de `var`. El ámbito de las variables declaradas con `let` es el bloque donde fue declarada y no puede utilizarse fuera.

```
{  
    var miVariable = "Soy visible fuera del bloque.";  
    let otraVariable = "No soy visible fuera del bloque.";  
}  
console.log(miVariable);  
console.log(otraVariable);
```

```
var a = 10; // x vale 10  
{  
    let a = 5; // a vale 5  
}  
// aquí a vale 10  
{  
    let b=3;  
}  
// aquí no se puede utilizar b  
{  
    var c=7;  
}  
// aquí sí se puede utilizar c y valdrá 7.
```



9. Declaración e inicialización de variables

3) CONST

Se utiliza para definir constantes.

CONSTANTE – variable a la que se le asigna un valor que no cambiará nunca, es decir, es una variable que no varía.

Se suelen escribir en mayúsculas, para distinguirlas del resto de las variables.

Constantes en JavaScript

En JavaScript, a partir de ES6 (ES2015), se puede utilizar la palabra reservada *const*. Por lo tanto, en vez de utilizar:

```
var pi = 3.141592;
```

se aconseja emplear:

```
const PI = 3.141592;
```

dado que pi es una constante (valor invariable). Por convención, se suele utilizar mayúsculas cuando se definen constantes.

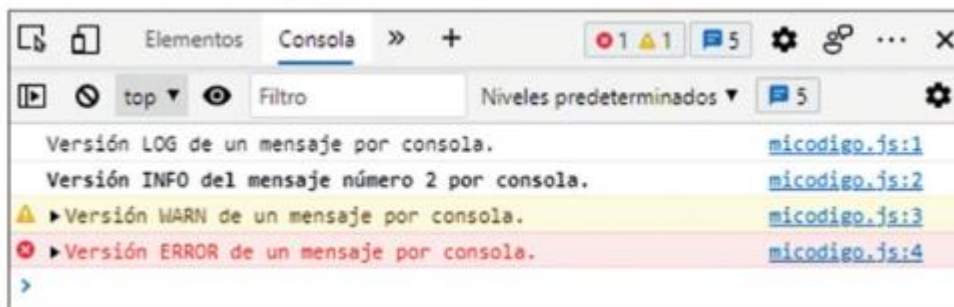
10. Entrada y salida del navegador

1) MENSAJES EN CONSOLA

Un recurso muy habitual que utilizan los programadores durante el desarrollo y despliegue de sus aplicaciones es enviar texto a la consola. Es bastante útil porque permite obtener mucha información de depuración de los programas sin «ensuciar» el aspecto de esos programas y, al mismo tiempo, son fácilmente localizables para retirarlos antes de poner las aplicaciones en entornos de producción.

Existen cuatro tipos de mensajes que se pueden generar para sacarlos en la consola: `console.log()`, `console.info()`, `console.warn()` y `console.error()`. Todos ellos muestran un texto con una estética diferente (que puede cambiar según el navegador) y se pueden utilizar los filtros que proporciona el navegador para ver solo aquellos del tipo que interese.

```
console.log("Versión LOG de un mensaje por consola.");  
console.info("%cVersión INFO del mensaje número %d por consola.,"font-weight: bold;";2);  
console.warn("Versión WARN de un mensaje por consola.");  
console.error("Versión ERROR de un mensaje por consola.");
```



Cada tipo de mensaje se muestra con una apariencia diferente.

El segundo mensaje, como comienza por `%c` puede establecerse un CSS personalizado.

Si en cualquier parte de la cadena se coloca `%d` permite sustituirse dinámicamente por un valor numérico y si se utiliza `%s` por una cadena de caracteres.

10. Entrada y salida del navegador

Lista adicional de métodos útiles para trabajar con la consola:

Método	Descripción
<code>console.dir()</code>	Muestra un listado interactivo de las propiedades de un objeto JavaScript específico.
<code>console.group()</code>	Crea un nuevo grupo que tabula los mensajes de la consola. Para retroceder un nivel de tabulación se utiliza <code>groupEnd()</code> .
<code>console.groupCollapsed()</code>	Igual que <code>group()</code> pero aparecen todos los mensajes colapsados en uno, siendo necesario pulsar un botón para desplegarlos.
<code>console.table()</code>	Muestra los datos en forma de tabla.
<code>console.time()</code>	Inicia un temporizador con un nombre personalizado (pueden crearse hasta 10 000 simultáneamente en la misma página). El temporizador se detiene con <code>timeEnd()</code> .
<code>console.trace()</code>	Muestra una traza del error.

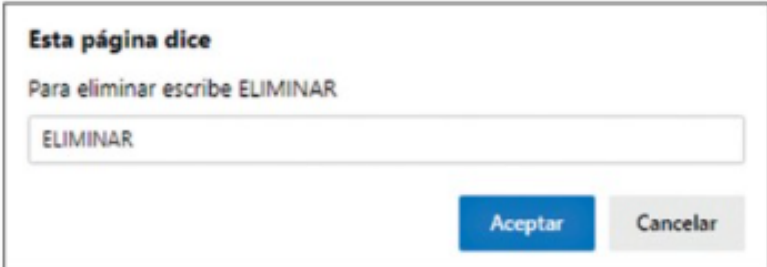
10. Entrada y salida del navegador

2) MENSAJES DE ENTRADA

Existe una variante al cuadro de diálogo anterior que además de mostrar un mensaje y dos botones para aceptar y rechazar la acción propuesta, proporciona una caja de texto para que el usuario pueda escribir. Esto se hace a través de la herramienta **prompt()**.

Si el usuario pulsa el botón **Cancelar** se obtiene **null** en la variable que controla la respuesta. Si el usuario pulsa el botón **Aceptar** pero no ha escrito nada, se obtiene una cadena vacía. Si escribe algo y pulsa **Aceptar**, se recibirá la cadena introducida.

```
let mensaje = "Para eliminar escriba ELIMINAR";  
let respuesta = prompt(mensaje);  
console.log(`El usuario escribió: ${respuesta}.`);
```



Cuadro de entrada de texto relleno por el usuario.



Salida del valor devuelto por el cuadro de texto tras pulsar Aceptar.

10. Entrada y salida del navegador

Además, puede establecerse un valor predeterminado que aparecerá dentro de la caja de texto. En ocasiones es útil para dirigir al usuario. Se consigue pasando un segundo parámetro a **prompt()**.

```
let mensaje = "¿Qué IVA desea aplicar?";  
let respuesta = prompt(mensaje, "21%");  
console.log(respuesta);
```

A screenshot of a browser's prompt dialog box. The title bar reads "Esta página dice". The main text inside the dialog is "¿Qué IVA desea aplicar?". Below this text is a text input field containing the value "21%". At the bottom right of the dialog are two buttons: "Aceptar" (Accept) in a blue button and "Cancelar" (Cancel) in a grey button.

Esta página dice

¿Qué IVA desea aplicar?

21%

Aceptar Cancelar

Cuadro de entrada de texto con valor por defecto.

10. Entrada y salida del navegador

3) MENSAJES DE ALERTA

Consiste en lanzar una pequeña ventana donde se mostrará algún mensaje de interés para el usuario. Para cerrarlo basta con pulsar **Aceptar**. Esto se realiza mediante la herramienta **alert()**.

```
const PI = 3.14159;  
alert(`Recuerda usar esta aproximación de  $\pi$  en tus cálculos: ${PI}`);
```

Esta página dice

Recuerda usar esta aproximación de π en tus cálculos: 3.14159

Aceptar

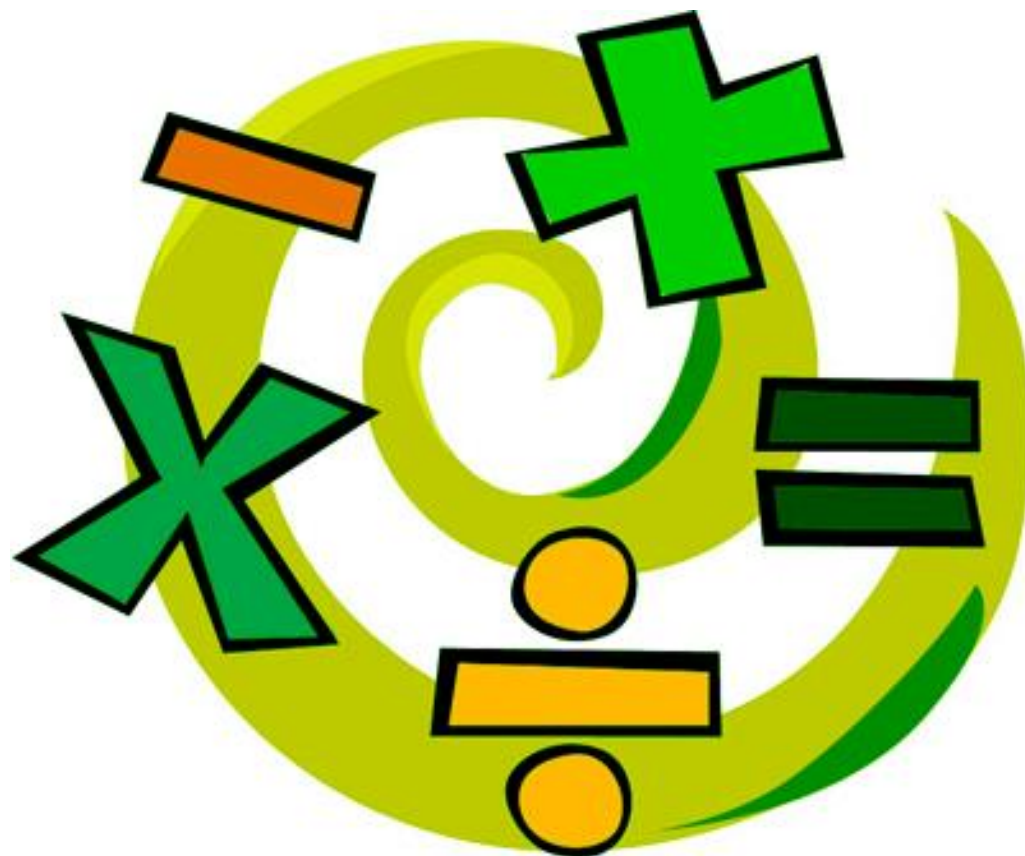
Importante



Si bien la utilización de mensajes de alerta, cuadros de confirmación y cuadros de entrada de texto son muy útiles desde el punto de vista del aprendizaje, es importante destacar que son cosas del pasado. En la actualidad ninguna aplicación web sería utiliza estas herramientas del navegador para interactuar con el usuario. Existen multitud de soluciones mucho más profesionales para este cometido.

Tipos de operadores en JS:

1. Operadores aritméticos
2. Operadores de asignación
3. Operadores de comparación
4. Operadores lógicos
5. Operadores bit a bit
6. Operadores de cadena
7. Operador condicional



Operadores JavaScript

https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Expressions_and_Operators

1) OPERADORES ARITMÉTICOS

Operador	Significado	Ejemplo
Módulo (%)	Devuelve el resto de la división entera.	9 % 5 devuelve 4
Incremento (++)	Operador unario que suma uno a su operando. Preincremento (++a): incrementa el valor de a en 1 y devuelve su contenido. Posincremento (a++): devuelve su contenido y después incrementa el valor de a en 1.	let a = 9; b = ++a; b vale 10 let a = 9; b = a++; b vale 9
Decremento (--)	Operador unario que resta uno a su operando. Predecremento (--a): decrementa el valor de a en 1 y devuelve su contenido. Posdecremento (a--): devuelve su contenido y después decrementa el valor de a en 1.	let a = 9; b = --a; b vale 8 let a = 9; b = a--; b vale 9
Potencia (**)	Calcula la potencia de un número elevando el operando izquierdo al exponente del operando derecho.	2 ** 3 devuelve 8
Negativo unario (-)	Devuelve la negación de su operando.	Si a es 5 entonces devuelve -5
Positivo unario (+)	Intenta convertir el operando en un número, si aún no lo es.	+"3" devuelve 3 +true devuelve 1

Operadores aritméticos

Operador	Descripción
+	Suma
-	Resta
*	Multiplicación
/	División
%	Módulo
++	Incremento
--	Decremento

2) OPERADORES DE ASIGNACIÓN

Asignan al operando izquierdo el valor del operando derecho por medio del símbolo =.

Nombre	Aplicación	Simplificación
Asignación	$a = b$	$a = b$
Asignación de adición	$a = a + b$	$a += b$
Asignación de resta	$a = a - b$	$a -= b$
Asignación de multiplicación	$a = a * b$	$a *= b$
Asignación de división	$a = a / b$	$a /= b$
Asignación de módulo	$a = a \% b$	$a \% = b$
Asignación de exponenciación	$a = a ** b$	$a ** = b$
Asignación de desplazamiento a la izquierda	$a = a << b$	$a << = b$
Asignación de desplazamiento a la derecha	$a = a >> b$	$a >> = b$
Asignación de desplazamiento a la derecha sin signo	$a = a >>> b$	$a >>> = b$
Asignación AND bit a bit	$a = a \& b$	$a \& = b$
Asignación XOR bit a bit	$a = a \wedge b$	$a \wedge = b$
Asignación OR bit a bit	$a = a b$	$a = b$
Asignación AND lógico	$a = a \&\& (a = b)$	$a \&\& = b$
Asignación OR lógico	$a = a (a = b)$	$a = b$
Asignación de anulación lógica	$a = a ?? (a = b)$	$a ?? = b$

Operadores de asignación

Operador	Ejemplo de uso
=	$x = y;$
+=	$x += y;$ // igual que $(x = x + y)$
-=	$x -= y;$ // igual que $(x = x - y)$
*=	$x *= y;$ // igual que $(x = x * y)$
/=	$x /= y;$ // igual que $(x = x / y)$
%=	$x \% = y;$ // igual que $(x = x \% y)$

Lista completa con los operadores de asignación compuesta de JavaScript

3) OPERADORES DE COMPARACIÓN

Su función es determinar si una comparación entre dos operandos es verdadera (**true**) o falsa (**false**). El resultado de la operación es siempre un valor booleano.

Operador	Significado	Resultado true
Igualdad (==)	Devuelve true si los operandos son iguales.	5 == 5 "5" == 5 5 == '5'
Distinto (!=)	Devuelve true si los operandos no son iguales.	5 != 9 9 != "5"
Igualdad estricta (===)	Devuelve true si los operandos son iguales y del mismo tipo.	5 === 5
Desigualdad estricta (!==)	Devuelve true si los operandos son del mismo tipo pero no iguales, o son de diferente tipo.	5 !== "5" 5 !== '5'
Mayor que (>)	Devuelve true si el operando izquierdo es mayor que el operando derecho.	9 > 5 "11" > 9
Mayor o igual que (>=)	Devuelve true si el operando izquierdo es mayor o igual que el operando derecho.	9 >= 3 9 >= 9
Menor que (<)	Devuelve true si el operando izquierdo es menor que el operando derecho.	3 < 9 "9" < 11
Menor o igual que (<=)	Devuelve true si el operando izquierdo es menor o igual que el operando derecho.	5 <= 9 9 <= 9

4) OPERADORES LÓGICOS

Permiten evaluar condiciones que permitirán tomar decisiones a lo largo del flujo de ejecución del programa.

Se suelen utilizar con valores booleanos y devuelven otro valor booleano.

Operador	Uso	Descripción
AND lógico (&&)	<code>expr1 && expr2</code>	Devuelve expr1 si se puede convertir a false , de lo contrario devuelve expr2 . Por tanto, cuando se usa con valores booleanos, devuelve true si ambos operandos son true y false en caso contrario.
OR lógico ()	<code>expr1 expr2</code>	Devuelve expr1 si se puede convertir a true , de lo contrario devuelve expr2 . Por tanto, cuando se usa con valores booleanos, devuelve true si alguno de los operandos es true o false si ambos son false .
NOT lógico (!)	<code>!expr</code>	Devuelve false si su único operando se puede convertir a true ; en caso contrario devuelve true .

Las expresiones que pueden convertirse a **false** son aquellas que se evalúan a **0**, **""**, **null**, **NaN** o **undefined**.

Las tablas de verdad de los operados lógicos son:

a	b	a && b
0	0	0
0	1	0
1	0	0
1	1	1

a	b	a b
0	0	0
0	1	1
1	0	1
1	1	1

```
// OPERACIONES AND
console.log(`1 = ${true && true}`);
console.log(`2 = ${true && false}`);
console.log(`3 = ${false && true}`);
console.log(`4 = ${false && (3 == 4)}`);
console.log(`5 = ${'Gato' && 'Perro'}`);
console.log(`6 = ${false && 'Gato'}`);
console.log(`7 = ${'Gato' && false}`);
```

```
// OPERACIONES OR
console.log(`8 = ${true || true}`);
console.log(`9 = ${false || true}`);
console.log(`10 = ${true || false}`);
console.log(`11 = ${false || (3 == 4)}`);
console.log(`12 = ${'Gato' || 'Perro'}`);
console.log(`13 = ${false || 'Gato'}`);
console.log(`14 = ${'Gato' || false}`);
```

a	!a
0	1
1	0

```
// OPERACIONES NOT
console.log(`15 = ${!true}`);
console.log(`16 = ${!false}`);
console.log(`17 = ${!'Gato'}`);
```

```
1 = true
2 = false
3 = false
4 = false
5 = Perro
6 = false
7 = false
8 = true
9 = true
10 = true
11 = false
12 = Gato
13 = Gato
14 = Gato
15 = false
16 = true
17 = false
```

Resultado de la ejecución de las instrucciones con operadores lógicos

5) OPERADORES BIT A BIT

Tratan a sus operandos como una ristra de 32 bits, es decir, operan en binario bit a bit, una vez operado, el intérprete JS devolverá el resultado convertido de nuevo a decimal.

Operador	Uso	Descripción
AND a nivel de bits	a & b	Devuelve un 1 en cada posición del bit donde ambos operandos son 1.
OR a nivel de bits	a b	Devuelve un 0 en cada posición del bit donde ambos operandos son 0.
XOR a nivel de bits	a ^ b	Devuelve un 0 en cada posición del bit donde ambos operandos son iguales, o 1 si ambos operandos son distintos.
NOT a nivel de bits	~ a	Invierte los bits de su operando.
Desplazamiento a la izquierda	a << b	Desplaza el primer operando a a la izquierda tantos bits como indique el operando b . Los bits en exceso desplazados a la izquierda se descartan. Los bits cero se desplazan desde la derecha.
Desplazamiento a la derecha (con propagación de signo)	a >> b	Desplaza el primer operando a a la derecha tantos bits como indique el operando b . Los bits en exceso desplazados a la derecha se descartan. Las copias del bit más a la izquierda se desplazan desde la izquierda. Dado que el nuevo bit más a la izquierda tiene el mismo valor que el bit anterior a la izquierda, el bit de signo (el bit más a la izquierda) no cambia.
Desplazamiento a la derecha sin signo (o desplazamiento a la derecha de relleno cero)	a >>> b	Desplaza a la derecha el primer operando el número de bits indicado. Los bits en exceso desplazados hacia la derecha se descartan. Los bits cero se desplazan desde la izquierda. El bit de signo se convierte en 0, por lo que el resultado nunca es positivo. A diferencia de los otros operadores a nivel de bits, el desplazamiento a la derecha de relleno cero devuelve un entero de 32 bits sin signo.

6) OPERADORES DE CADENA

En JS se utilizan los operadores + y += para concatenar cadenas.

```
console.log("Así" + " concatenamos " + 4 + " cadenas"); // "Así concatenamos 4 cadenas"
```

```
let cadena = "nombre";  
cadena += " y apellidos";  
console.log(cadena); // "nombre y apellidos"
```

7) OPERADOR CONDICIONAL o TERNARIO

Actúan tres operadores. La sintaxis es:

condición ? valor1: valor2

Si la condición se evalúa a **true**, la operación devuelve el **valor1**, en caso contrario devuelve **valor2**

```
let edad = 21;  
let mensaje = (edad >= 18) ? "mayor" : "menor";  
console.log(`El usuario es ${mensaje} de edad.`); // "El usuario es mayor de edad."
```

PRECEDENCIA DE LOS OPERADORES

Establece el orden que se aplica a los operadores al evaluar una expresión.

Tipo de operador	Operadores
Miembro	. []
Llamar / crear instancia	() new
Negación / incremento	! ~ - + ++ -- typeof void delete
Multiplicar / dividir	* / %
Adición / sustracción	+ -
Desplazamiento bit a bit	<< >> >>>
Relacional	< <= > >= in instanceof
Igualdad	== != === !==
AND bit a bit	&
XOR bit a bit	^
OR bit a bit	
AND lógico	&&
OR lógico	
Condicional	?:
Asignación	= += -= *= /= %= <<= >>= >>>= &= ^= = &&= = ??=
Coma	,

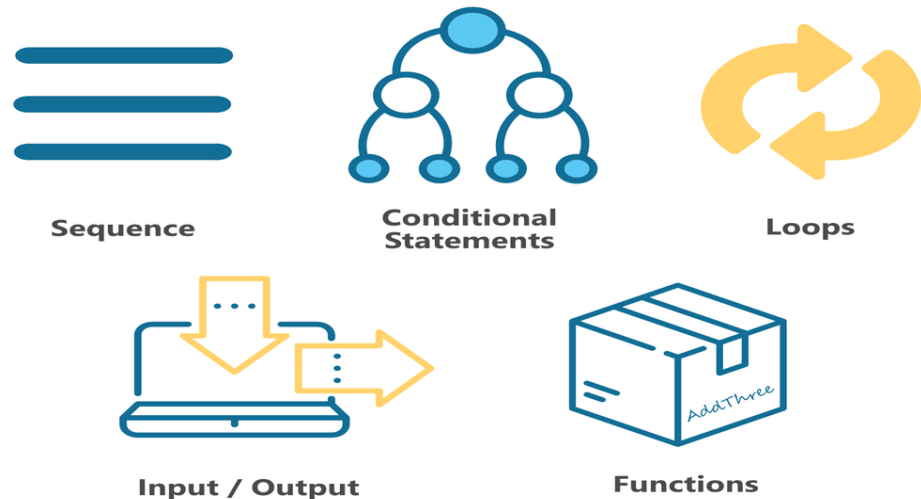
12. Estructuras de control

Son instrucciones que permiten elegir alternativas de ejecución, plantear bifurcaciones o realizar tareas repetitivas en un punto dado, antes de continuar con la siguiente instrucción, es decir, son instrucciones que permiten alterar el flujo de ejecución de un programa.

El elemento que determina estas trayectorias es la **condición**, que normalmente está construida con expresiones lógicas.

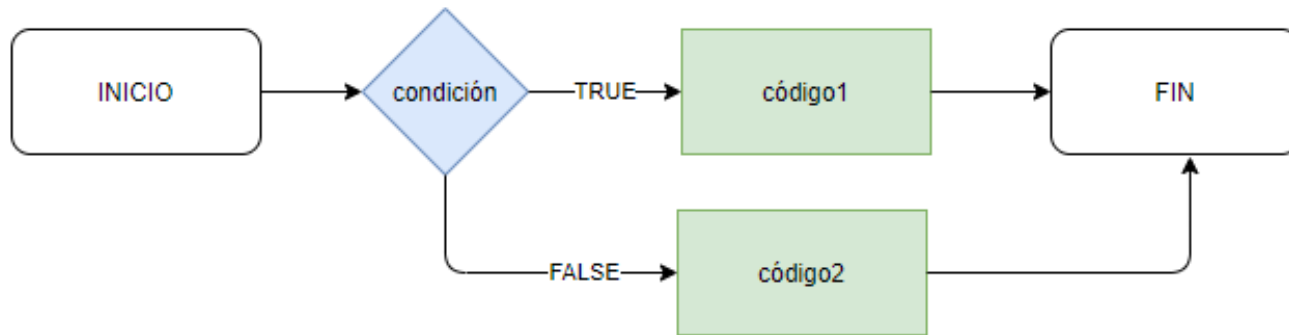
Las estructuras de control se clasifican en 3 grandes grupos:

- 1) Estructuras condicionales
- 2) Estructuras repetitivas
- 3) Estructuras de salto



1) ESTRUCTURAS CONDICIONALES

En función de una condición, permiten elegir un camino u otro.



Las estructuras condicionales que tiene JS son:

1.1) **IF**

1.2) **SWITCH**

1.1) IF

Es la estructura condicional más simple.

Funcionamiento: si se cumple la condición se ejecutan unas instrucciones que contiene, y si no se cumple, se ejecutan las instrucciones del bloque **else**.

Sintaxis:

IF SIMPLE	IF CON ELSE	IF ANIDADOS
<pre>if (condición){ /* código a ejecutar si la condición es verdad */ }</pre>	<pre>if (condición){ /* código a ejecutar si la condición es verdad */ } else{ /* código a ejecutar si la condición es falsa */ }</pre>	<pre>if (condición1){ /* código a ejecutar si la condición1 es verdad */ } else if (condición2){ /* código a ejecutar si la condición1 es falsa y la condición2 es verdad */ } else{ /* código a ejecutar si la condición1 y la condición2 son falsas */ }</pre>

Importante

En las estructuras **if**, y en general en todas las estructuras que utilicen bloques, si el contenido del bloque lo forma una única instrucción, no es necesario escribir las llaves.



Ejemplo 1: IF

Crear un programa que diga si podemos o no comprar un coche que vale 25000 €. Para ello, sale una ventana que pregunta qué cantidad tenemos ahorrada y si esa cantidad es mayor o igual que el precio del coche mostrará que puedes comprarlo, si no, mostrará que no puedes comprarlo.

```
1  <!DOCTYPE html>
2  <html lang="es-ES">
3    <head>
4      <meta charset="utf-8">
5      <title>Ejemplo HTML</title>
6      <script>
7        var precioCoche = 25000;
8        var dineroAhorrado = prompt('Introduce la cantidad que tienes ahorrada');
9        document.write('El precio del coche es ' + precioCoche + ' y tus ahorros son ' +
10          dineroAhorrado + ' por tanto: ');
11        if(dineroAhorrado>=precioCoche){
12          document.write("Puedes comprar el coche");
13        } else{
14          document.write("No puedes comprar el coche");
15        }
16      </script>
17    </head>
18    <body>
19  </body>
20 </html>
```

Ejemplo 2: IF

Modificar el ejemplo anterior añadiendo una condición nueva, además de tener el suficiente dinero ahorrado, debe ser mayor de 18 años, por tanto, hay que evaluar ambas condiciones.

```
1  <!DOCTYPE html>
2  <html lang="es-ES">
3    <head>
4      <meta charset="utf-8">
5      <title>Ejemplo HTML</title>
6      <script>
7        var precioCoche = 25000;
8        var dineroAhorrado = prompt('Introduce la cantidad que tienes ahorrada');
9        var edad = prompt('Introduce tu edad');
10       if(dineroAhorrado>=precioCoche && edad>=18){
11         document.write('El precio del coche es ' + precioCoche + ', tus ahorros son ' +
12           dineroAhorrado+' y tu edad '+edad+ ' por tanto: Puedes comparar el coche');
13       } else{
14         document.write('El precio del coche es ' + precioCoche + ', tus ahorros son ' +
15           dineroAhorrado+' y tu edad '+edad+ ' por tanto: No puedes comparar el coche');
16       }
17     </script>
18   </head>
19   <body>
20     </body>
21 </html>
```

1.2) SWITCH

Se utiliza para expresiones cuyo resultado puede ser variado y necesita distintas vías distintas de ejecución. Es una solución mucho más sencilla que los if anidados.

Sintaxis:

```
switch (expresión) {  
    case valor_1:  
        instrucciones_1  
        [break;]  
    case valor_2:  
        instrucciones_2  
        [break;]  
    ...  
    default:  
        instrucciones_predeterminadas  
        [break;]  
}
```

- Se utiliza la sentencia break para salir de cada bloque del switch.
- La sentencia default se ejecutará si ninguna de las opciones anteriores ha sido ejecutada con las opciones case.

Tras evaluar **expresión** se ejecutan las instrucciones del bloque **case** cuyo valor coincida con el resultado de la evaluación. Si el valor de ningún **case** coincide, entonces se ejecuta la sección **default**. Las instrucciones **break** aparecen entre corchetes porque son opcionales. Si se pone, ahí terminará la ejecución del **switch**. Si no se pone, se ejecuta también «todo» lo que hay desde ahí hasta el final, o hasta que se encuentre un **break**.

Ejemplo 3: SWITCH

Crear un programa que pida una vocal y que nos diga que vocal es la que hemos introducido.

```
1  <!DOCTYPE html>
2  <html lang="es-ES">
3    <head>
4      <meta charset="utf-8">
5      <title>Ejemplo HTML</title>
6      <script>
7        vocal = prompt('Introduce una vocal');
8        switch(vocal)
9          {
10           case 'a':
11             document.write('La vocal introducida es una a');
12             break;
13           case 'e':
14             document.write('La vocal introducida es una e');
15             break;
16           case 'i':
17             document.write('La vocal introducida es una i');
18             break;
19           case 'o':
20             document.write('La vocal introducida es una o');
21             break;
22           case 'u':
23             document.write('La vocal introducida es una u');
24             break;
25           default:
26             document.write(vocal + ' No es una vocal');
27         }
28      </script>
29    </head>
30    <body>
31
32  </body>
```

2) ESTRUCTURAS REPETITIVAS

Permiten la ejecución repetitiva de una o varias instrucciones mientras se cumpla una condición.

Las estructuras repetitivas que tiene JS son:

2.1) **WHILE**

2.2) **DO WHILE**

2.3) **FOR**



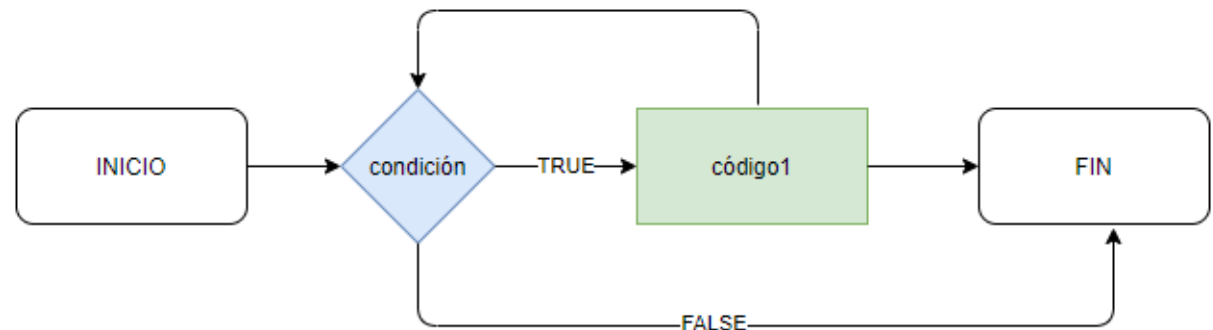
2.1) WHILE

El cuerpo del bucle while se repite mientras la condición sea verdadera, en el momento que la condición sea falsa, se sale del bucle.

Si no se cumple la condición, el bucle no se ejecuta nunca.

Sintaxis:

```
while (condición) {  
    instrucciones;  
}
```



Asegurarse de que la variable que controla la condición de continuación se modifique dentro del bucle es crucial. De no hacerlo, es muy probable que nunca se dé una condición de salida y se produzca un bucle infinito.

Recuerda



Las consecuencias de que se produzca un bucle infinito en un programa van mucho más allá de los problemas derivados de que el programa deje de funcionar. Provocará un consumo de recursos desproporcionado, puede colgar el navegador e incluso la propia máquina en la que se ejecuta.

Ejemplo 4: WHILE

Crear un programa con el bucle WHILE que cuente del 1 al 10 y lo muestre por pantalla.

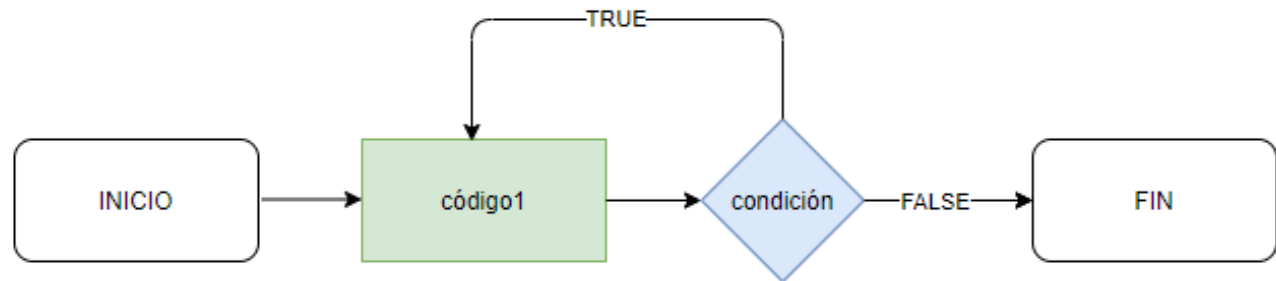
```
1  <!DOCTYPE html>
2  <html lang="es-ES">
3    <head>
4      <meta charset="utf-8">
5      <title>Ejemplo HTML</title>
6      <script>
7        var num=1;
8        document.write('CONTAR DEL 1 AL 10<BR>');
9        while (num<=10){
10         document.write(num + '<br>');
11         num++;
12       }
13     </script>
14   </head>
15   <body>
16
17   </body>
18 </html>
```

2.2) DO WHILE

Variante del bucle while. La condición se sitúa al final del bucle, en lugar de al principio. Por tanto, el bucle do while siempre se ejecuta al menos una vez.

Sintaxis:

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



Asegurarse de que la variable que controla la condición de continuación se modifique dentro del bucle es crucial. De no hacerlo, es muy probable que nunca se dé una condición de salida y se produzca un bucle infinito.

Recuerda



Las consecuencias de que se produzca un bucle infinito en un programa van mucho más allá de los problemas derivados de que el programa deje de funcionar. Provocará un consumo de recursos desproporcionado, puede colgar el navegador e incluso la propia máquina en la que se ejecuta.

Ejemplo 5: DO WHILE I

Crear un programa que pida por pantalla el nombre, la edad y la población. Debe hacer las siguientes comparaciones:

- Para el nombre, repite la petición y no continúa, si se introduce un número.
- Para la edad, repite la petición y no continúa, si lo que se introduce es un número menor que 0 o mayor que 105, o si se introduce un texto.
- Para la población, repite la petición y no continúa, si se introduce un número.

NOTA: para determinar si es un número utilizar la función **isNaN()**

isNaN

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/isNaN

```
1  <!DOCTYPE html>
2  <html lang="es-ES">
3    <head>
4      <meta charset="utf-8">
5      <title>Ejemplo HTML</title>
6      <script>
7        var nombre, edad, poblacion;
8        do{
9          nombre = prompt('Introduce tu nombre',
10            'Escribir aquí el nombre');
11        }while (!isNaN(nombre));
12
13        do{
14          edad = prompt('Introduce tu edad','Escribir
15            aquí la edad');
16        }while (edad<=0 || edad>105 || isNaN(edad));
17
18        do{
19          poblacion = prompt('Introduce tu población',
20            'Escribir aquí la población');
21        }while (!isNaN(poblacion));
22
23        document.write('Nombre: ' + nombre);
24        document.write('<br>Edad: ' + edad);
25        document.write('<br>Población: ' + poblacion);
26      </script>
27    </head>
28    <body>
29
30    </body>
31  </html>
```

Ejemplo 5: DO WHILE II

ACLARACIONES:

En el código hay varias novedades:

- nombre = prompt('Introduce tu nombre','Escribir aquí el nombre'); - si al prompt le añadimos una segunda cadena esta se mostrará dentro de la casilla donde hay que introducir los datos.
- isNaN(); - función que se utiliza para averiguar si una variable es de tipo número.
- !isNaN() – si a la función anterior se le añade el operador !, lo que averigua es si una variable no es de tipo número.

```
1  <!DOCTYPE html>
2  <html lang="es-ES">
3    <head>
4      <meta charset="utf-8">
5      <title>Ejemplo HTML</title>
6      <script>
7        var nombre, edad, poblacion;
8        do{
9          nombre = prompt('Introduce tu nombre',
10            'Escribir aquí el nombre');
11        }while (!isNaN(nombre));
12
13        do{
14          edad = prompt('Introduce tu edad','Escribir
15            aquí la edad');
16        }while (edad<=0 || edad>105 || isNaN(edad));
17
18        do{
19          poblacion = prompt('Introduce tu población',
20            'Escribir aquí la población');
21        }while (!isNaN(poblacion));
22
23        document.write('Nombre: ' + nombre);
24        document.write('<br>Edad: ' + edad);
25        document.write('<br>Población: ' + poblacion);
26      </script>
27    </head>
28    <body>
29
30    </body>
31  </html>
```


2.3) FOR

El bucle se repite hasta que la condición especificada sea falsa. El bucle for es determinado, es decir, que antes de la ejecución ya sabemos cuántas veces se va a ejecutar.

Sintaxis:

```
for(inicio; condición; incremento/decremento){  
    //código a repetir mientras la condición sea cierta  
}
```

Dentro del paréntesis del FOR tenemos tres zonas separadas por ; (punto y coma):

- 1) Inicio del bucle – valor inicial de la variable de control del bucle.
- 2) Condición del bucle – lo que hace que el bucle se ejecute, es decir la condición.
- 3) Incremento/decremento – cómo cambia la variable de control para que el bucle llegue a finalizar.

Ejemplo 5: FOR

Crear un programa utilizando el bucle FOR que cuente del 1 al 10 y lo muestre por pantalla.

```
1  <!DOCTYPE html>
2  <html lang="es-ES">
3  <head>
4  <meta charset="utf-8">
5  <title>Ejemplo HTML</title>
6  <script>
7  |   var num=1;
8  |   document.write('CONTAR DEL 1 AL 10<BR>');
9  |   for(i=1; i<=10; i++){
10 |   |   document.write(i + '<br>');
11 |   }
12 |   </script>
13 </head>
14 <body>
15
16 </body>
17 </html>
```

3) ESTRUCTURAS DE SALTO

Permiten romper el flujo de ejecución establecido por los bucles cuando se dan situaciones excepcionales que quieren controlarse.

Las estructuras de salto que tiene JS son:

3.1) Break

3.2) Continue

3.3) Labeled



Break



Continue

Para saber más

Las instrucciones de salto son muy útiles en situaciones muy concretas, pero no debe abusarse de ellas. Un exceso de instrucciones de salto termina convirtiendo el programa en **código espagueti**, o lo que es lo mismo, un galimatías en el que nadie es capaz de seguir el flujo de ejecución del programa.



3.1) BREAK

Se utiliza para terminar un bucle **while**, **do while** o **for** o una sentencia **switch** y transferir el control a la siguiente instrucción.

En general, se usa para romper el bucle envolvente más cercano al break.

```
<!DOCTYPE html>
<html>
<body>
<p>Se supone que el bucle genera de 0 a 4, pero la instrucción break sale del bucle cuando i == 3:</p>
<p id="demo"></p>

<script>
let text = "";
for (let i = 0; i < 5; i++) {
  if (i == 3) break;
  text += i + "<br>";
}
document.getElementById("demo").innerHTML = text;
</script>

</body>
</html>
```

break

https://www.w3schools.com/jsref/jsref_break.asp

Se supone que el bucle genera de 0 a 4, pero la instrucción break sale del bucle cuando i == 3:

0
1
2

3.2) CONTINUE

Se utiliza para, estando en medio de una iteración de un bucle, se desea descartar el resto de las instrucciones del bloque y volver a evaluar la condición.

```
<!DOCTYPE html>
<html>
<body>
<p>Mostrar valores:</p>
<p id="demo"></p>
<script>
let text = "";
for (let i = 0; i < 5; i++) {

    text += i + "<br>";
}
document.getElementById("demo").innerHTML = text;
</script>
</body>
</html>
```

Mostrar valores:

0
1
2
3
4

```
<!DOCTYPE html>
<html>
<body>
<p>Mostrar valores:</p>
<p id="demo"></p>
<script>
let text = "";
for (let i = 0; i < 5; i++) {
    if (i === 3) continue;
    text += i + "<br>";
}
document.getElementById("demo").innerHTML = text;
</script>
</body>
</html>
```

Omite el 3

Mostrar valores:

0
1
2
4

continue

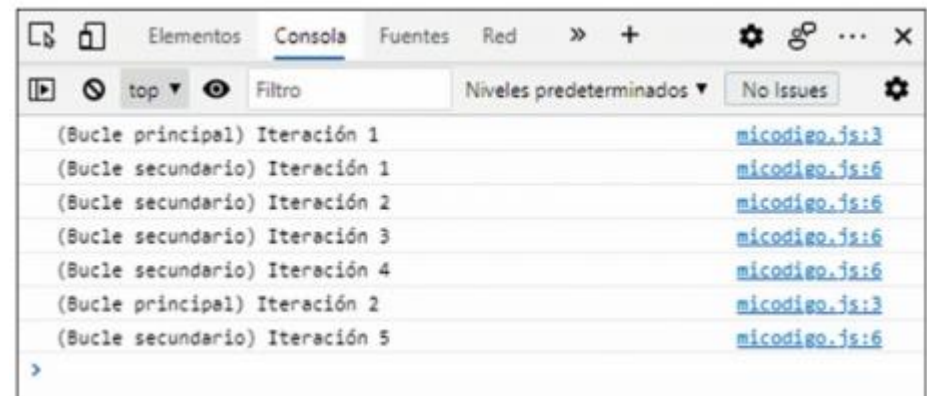
https://www.w3schools.com/jsref/jsref_continue.asp

3.3) LABELED

Completa la funcionalidad de las dos instrucciones anteriores, es decir, establece puntos en el programa, a los que se les asigna un nombre (etiqueta) al que hacer referencia cuando se desea efectuar un salto.

Esto permite saltar a cualquier punto del programa, previamente etiquetado.

```
let primero = segundo = 1;
buclePrincipal: while (true) {
  console.log('(Bucle principal) Iteración ${primero}');
  primero++;
  while (true) {
    console.log('(Bucle secundario) Iteración ${segundo}');
    segundo++;
    if (segundo == 5)
      break;
    else if (primero == 3)
      break buclePrincipal;
  }
}
```



En este código hay dos bucles **while** anidados. Como puede verse, el principal se ha etiquetado como **buclePrincipal**. Ninguno de los dos bucles tiene criterio de terminación en su condición principal (**true**). Están diseñados para que se ejecuten indefinidamente, o hasta que otra instrucción los rompa, como en este caso. Lo que se desea es que cuando las variables alcancen ciertos valores se rompa un bucle u otro en función de ciertos intereses. Tal y como se observa en el resultado de la ejecución, hay dos **break** situados dentro del **while** anidado. Llegado el momento, **break** rompe el bucle más próximo que lo envuelve, y **break buclePrincipal** rompe el bucle etiquetado con ese identificador.

Gracias

