CONTENIDOS

2.1 EJECUTAR CÓDIGO JAVASCRIPT

En esta unidad vamos a ver las bases del lenguaje JavaScript. Por lo tanto, lo que se ve en esta unidad es vital para poder progresar en el resto de unidades.

Lo primero que hemos de saber es cómo podemos ejecutar el código JavaScript. En este módulo principalmente lo usaremos para ser ejecutado en un navegador. Pero también es cierto que podemos ejecutar JavaScript fuera del navegador (con ayuda de node.js) para aprender más rápidamente los fundamentos del lenguaje.

2.1.1 EJECUCIÓN DE JAVASCRIPT EN EL NAVEGADOR

JavaScript nació para ser un lenguaje interpretado por un navegador. Por ello, ejecutar JavaScript desde el código de una página web, sigue siendo considerado como el método principal de ejecución de este lenguaje. Los navegadores interpretan todo el código JavaScript que se encuentre dentro de una etiqueta script. Ejemplo:

```
<!DOCTYPE html>
<html lang="es">
<head>
	<meta charset="UTF-8">
	<title>Prueba Javascript</title>
</head>
<body>
<script>
	document.write("Hola desde Javascript");
</script>
</body>
</html>
```

La etiqueta script contiene el código JavaScript, que en el ejemplo anterior provoca (gracias al método document.write) que se escriba el texto Hola desde Javascript en el cuerpo de nuestra página web.

La etiqueta script se puede colocar tanto en la cabecera del documento (dentro de head) como dentro de la propia etiqueta body. Normalmente, se coloca en la cabecera si es código para cargar librerías general o código que define funciones y objetos globales, pero que no interacciona con el documento. Se coloca en el cuerpo y justo antes del body, el código JavaScript que se comunica con los objetos del documento.

Ahora bien, el código JavaScript, aun siendo colocado al final de body puede ejecutar su código antes de que se carguen todos los elementos de la página, ya que la carga de una página web se realiza de forma asíncrona.

ESCRITURA POR CONSOLA

Todos los navegadores actuales poseen una ventana de depuración para poder examinar, modificar y testear el código de las aplicaciones web a fin de verificar su eficacia y de poder hacer pruebas y ensayos en él. El lenguaje JavaScript posee un objeto llamado **console** que permite utilizar la consola de depuración desde el propio

código. El método log se utiliza para escribir en la consola, de modo que esta página web:

```
<!DOCTYPE html>
<html lang="es">
<head>
      <meta charset="UTF-8">
      <title>Document</title>
</head>
<body>
<script>
      consolé.log("Hola");
</script>
</body>
</html>
```

Si ejecutamos este código en un navegador, no muestra nada en la ventana, pero en el panel de depuración veremos escrito el texto **Hola**.

Podemos ver la consola, dentro del panel de depuración, en prácticamente todos los navegadores pulsando F12 (algunos navegadores usan otras teclas). El panel de depuración sirve para hacer pruebas y detectar fallos en nuestras aplicaciones web. En este panel siempre hay una pestaña llamada Consola.

CARGAR ARCHIVOS EXTERNOS JAVASCRIPT

Si queremos, podemos escribir el código en otro archivo separado (al que se le debe colocar la extensión **js)** y, desde el código HTML, cargar su código. Para ello, la etiqueta **script** dispone de un atributo llamado **src** al que se le indica la ruta (URL) del archivo que queremos cargar:

```
<script src="carpeta/fichero.js" > </script>
```

Esta etiqueta carga y ejecuta el código JavaScript del archivo ***fichero.js*** que se encuentra dentro del directorio ***carpeta*** que estará, a su vez, en el mismo directorio en el que se encuentre la aplicación web.

Atributos de la etiqueta <script>:

La etiqueta script, en HTML 4, requería, obligatoriamente, utilizar un atributo llamado type cuyo valor era el lenguaje concreto que se usa en el script (normalmente type="JavaScrípt").

Actualmente, desde HTML 5, este atributo es opcional porque se da por hecho que lo normal es que el código sea JavaScript.

Cuando se cargan archivos externos, es decir, si se usa el atributo src, disponemos de más atributos como :

El atributo **async** que puede tomar los valores true (verdadero) o false (falso). Con valor true hace que la ejecución del código JavaScript sea asíncrona. Es decir, se carga el código de forma independiente a la carga de elementos de la página.

El atributo **defer** (con los posibles valores true o false,) en caso de valer true, indica que la carga del archivo se realizará cuando los componentes de la página web se hayan ya cargados. Es útil (si el navegador entiende este atributo) cuando el código que queremos cargar sirve para manipular el contenido de la página web.

2.1.2 EJECUCIÓN DE JAVASCRIPT FUERA DEL NAVEGADOR

Si deseamos ejecutar JavaScript fuera del navegador, necesitamos la ayuda de **node.js**.

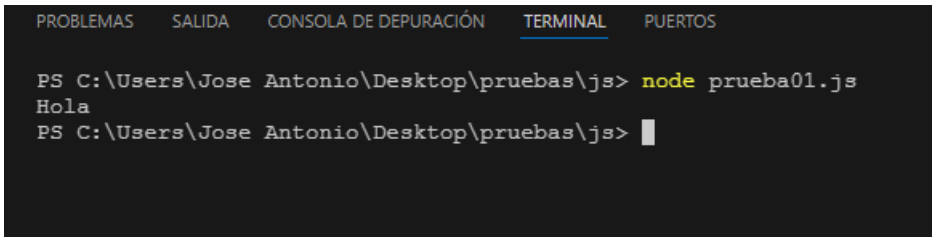
Podemos ejecutar directamente código JavaScript de forma rápida si, desde un terminal del sistema, invocamos a node y (tras la aparición del símbolo ">") escribimos el código deseado. Podemos seguir escribiendo código JavaScript, hasta que decidamos salir del entorno pulsando dos veces la combinación de teclas Ctrl+C.

```
C:\Users\Jose Antonio\Desktop\pruebas\js>node
Welcome to Node.js v20.17.0.
Type ".help" for more information.
> console.log("Hola");
Hola
undefined
>
```

También podemos escribir código JavaScript en un archivo (es recomendable que tenga extensión js) y después ejecutar el código de ese archivo utilizando node. Supongamos que hemos grabado un archivo que se llama prueba01.js, entonces desde el terminal nos debemos situar en el directorio en el que está el archivo y ejecutar este comando:

```
C:\Users\Jose Antonio\Desktop\pruebas\js>node prueba01
Hola
```

Ejecutar node desde Visual Studio Code



2.2 FUNDAMENTOS BÁSICOS

2.2.1 CONCEPTOS FUNDAMENTALES

Desde hace muchos años se ha hecho una división de los distintos lenguajes de programación entre los que son **compilados** y los que son **interpretados**.

Antes de la década de los noventa del siglo XX, la mayoría de lenguajes que florecían eran compilados. La razón es que el código de un lenguaje compilado se convierte a código máquina, analizando todas las líneas del mismo antes de empezar a convertir en binario el código. De esta forma, se consigue un código más eficiente. Si hay un error, por ejemplo, en la línea de código número 500, en el proceso de compilación se nos indicará el error de esa línea y no veremos el resultado que producen las primeras 499 líneas (que son correctas) si no arreglamos el problema de la línea 500.

En los lenguajes interpretados se realiza la conversión a código máquina línea a línea. Usando el mismo ejemplo, si tenemos un error en la línea 500 veríamos con normalidad el resultado de las primeras 499 líneas. Sería al intentar interpretar la línea 500 cuando se notificaría el error.

Sabiendo que la depuración es más compleja y que el código final de los lenguajes interpretados es menos eficiente (al no optimizar el código a través del análisis de todas las líneas) no parece haber razón alguna para utilizar lenguajes interpretados.

Sin embargo, el código interpretado ofrece ventajas cuando se debe ejecutar en un entorno donde la carga del programa es lenta, como ocurre en Internet (aunque cada vez menos). El hecho de que sea interpretado permite ir ejecutando las primeras líneas que se carguen antes incluso de que se carguen las siguientes, lo cual es mucho más eficiente en el caso de código incrustado en el lado del cliente de las aplicaciones web.

Tradicionalmente, se considera a JavaScript un lenguaje interpretado. Pero la realidad es más compleja. Inicialmente JavaScript solo era apto para realizar acciones no muy complejas, hoy en día sus capacidades no le diferencian de la mayoría de los lenguajes existentes, y eso ha implicado que los navegadores ***precompilen*** el código JavaScript para que sea más veloz su rendimiento.

Actualmente, JavaScript utiliza un compilador **JIT** (Just In Time) gracias a una máquina virtual de JavaScript (JVM) que proporcionan los navegadores. Sin extendernos demasiado, JavaScript es un lenguaje con una compilación semejante a la del lenguaje Java. El código deJavaScript se compila al vuelo (ya que se debe ejecutar lo más rápido posible) para que se pueda servir lo más rápido posible, pero habiendo optimizado el código.

No se consigue la rapidez de lenguajes totalmente compilados como el lenguaje C, pero tampoco adolece de la falta de eficiencia del lenguaje JavaScript original. Al final, la velocidad de este proceso de conversión de código depende de los trucos y habilidades que proporcionan los motores de ejecución de JavaScript. El

famoso V8 del navegador Chrome de Google, consiguió acelerar la ejecución de código JavaScript de forma notable.

En todo caso, sí conviene pensar que JavaScript tiene mucho todavía de lenguaje interpretado.

SENTENCIA O INSTRUCCIÓN

En JavaScript las instrucciones simples se pueden finalizar con el símbolo de punto y coma (;). Sin embargo, no es obligatorio y podemos escribir instrucciones sin indicar el final. Es recomendable finalizar las instrucciones simples siempre con punto y coma, ya que eso posibilita un mejor aprendizaje del lenguaje si entendemos cuándo debemos indicar el punto y coma.

Un ejemplo de sentencia es:

var nombre="Jose";

IDENTIFICADORES Y PALABRAS RESERVADAS

La palabra var del código anterior es lo que se conoce como palabra reservada. Un término especial que, dentro del lenguaje, tiene un significado especial. No podemos utilizar las palabras reservadas para identificar a otros elementos del lenguaje.

Es decir, las palabras reservadas son términos que ofrece el lenguaje para poder programar. Mientras que los nombres que damos a variables y otros elementos del lenguaje se conocen como identificadores, ya que su labor es precisamente identificar de forma inequívoca a un elemento del lenguaje.

Los identificadores deben de cumplir las siguientes reglas:

- Deben comenzar por una letra, un guión bajo (_) o el símbolo dólar ($).
- Después se pueden usar letras, números o el guión bajo.
- Se pueden usar letras Unicode, aunque estén fuera del código ASCII, como á o é.
- Se distingue entre mayúsculas y minúsculas. salarioFinal es un identificador distinto de SalarioFinal.

REGLAS FUNDAMENTALES

JavaScript es un ***lenguaje sensible a mayúsculas***. Es decir, se distingue entre mayúsculas y minúsculas. No es lo mismo var que VAR, por eso este código produce un error:

VAR x=7; //¡Error!

Los comentarios de una línea comienzan con los símbolos //. Un comentario es un texto explicativo que no se tendrá en cuenta al interpretar el código, y por ello, se usa para documentar el código. De esa forma podemos explicar mejor lo que hacemos en el código especialmente si trabajamos en equipo, pero también

incluso para nosotros mismos ya que, con el paso del tiempo, se nos olvidará el funcionamiento de muchas de las instrucciones que escribamos:

var tam = nombre.length(); //tam recoge el tamaño del nombre

Los comentarios de varias líneas comienzan con los símbolos /* y terminan con */

```
/* Comienzo de la función principal
de proceso de datos */
```

Se pueden agrupar sentencias en bloques de sentencias. Los bloques son sentencias contenidas entre llaves (símbolos **{** y **}**). Normalmente los bloques forman parte de sentencias complejas. Ejemplo:

```
var x=9;
if (x==9) {
  x++;
  console.log(x);
}
```

En este ejemplo, la sentencia if contiene un bloque de sentencias, las cuales se ejecutan si se cumple la condición x==9 (x igual a 9). En este caso, si probamos el código veremos que, por consola, se escribe el número 10.

2.3 VARIABLES Y TIPOS DE DATOS

2.3.1 VALORES

Los programas en JavaScript utilizan **valores**. Los valores son datos que el programa necesita almacenar, manipular o mostrar.

En JavaScript hay tres tipos de valores básicos: **Textos** (llamados Strings), **Números** y Valores **booleanos**. A estos tres tipos hay que añadir el tipo complejo **Objeto** y los valores **indefinido** y **nulo.**

Existe un operador llamado **typeof** que permite conocer el tipo JavaScript de un valor concreto. Para ello podemos invocar a node desde la línea de comandos, y pedir que nos escriba los tipos correspondientes a varios valores:

```
> typeof 12
' number'
> typeof 12.3
' number'
> typeof(12*3)
' number'
> typeof "Hola"
'string'
> typeof 'Hola'
'string'
> typeof Hola
'undefmed'
> typeof (1/0)
'number'
```

```
> typeof [1,2,3,4,5]
' object'
> typeof true
' boolean'
> typeof (12>3)
' boolean'
```

Algunos detalles:

- El operador **typeof** valora la expresión que escribamos a su derecha. Si es compleja, se debe indicar entre paréntesis para asegurar que se evalúa el conjunto completo. La respuesta de este operador es un texto que indica el tipo de datos al que pertenece la expresión.

- Todos los números tienen el mismo tipo (**number**) sin importar si son decimales o no. Incluso se devuelve el tipo number para la expresión 1/0 que realmente no se puede evaluar. Sin embargo, se considera que uno dividido entre cero produce infinito, y en JavaScript, el infinito es un número válido por eso typeof (1/0) devuelve number.

- Los textos se entrecomillan. Cualquier expresión entrecomillada (sin importar si usamos comillas simples o dobles) se entiende que es un texto (**string**).

- El tipo **undefined** se emplea cuando JavaScript no puede evaluar una expresión. La expresión ***typeof hola*** provoca un resultado indefinido porque se busca la variable hola (al no ser un texto entrecomillado) y, como no existe, se decide que es una expresión indefinida.

- Los tipos complejos (**object**) se utilizan mucho. Incluso el valor null se considera un objeto.

2.3.2 DECLARACIÓN DE VARIABLES

En los lenguajes de programación, ***los valores se almacenan en variables***. Las variables son uno de los elementos fundamentales en cualquier lenguaje de programación.

Cada **variable tiene un identificador**. A ese identificador se le asocian los valores que deseemos. Lo interesante es que podemos operar con las variables para realizar los cálculos que deseemos.

En JavaScript las variables deben de ser declaradas antes de poderse usar. Declarar una variable es asignarla un identificador. Tradicionalmente la palabra reservada var es la que se utiliza para declarar una variable:

var x;

De esta forma estamos avisando de que se podrá utilizar, a partir de ese momento, en el código una variable que se llama x.

Es muy habitual asignar un valor al declarar una variable:

var x=9;

Con esto declaramos la existencia de una variable que se llama x. Además, estamos indicando que, inicialmente, vale nueve.

Las variables solo se declaran una vez. Después la variable se usa con normalidad:

```
var x = 9;
x = 25;
console.log(x); //escribe 25
```

DIFERENCIA ENTRE LET Y VAR

Desde la versión 6 del estándar ECMAScript (conocido como ES2O15 o ES6), se permite el uso de las palabras reservadas let y const para declarar variables (además de var).

let se utiliza para declarar variables de forma más local que var. Para entender la diferencia veamos este código:

```
{
      let x=9;
}
consolé.log(x);
```

Si ejecutamos este código, aparecerá un error de que x no está definida. La razón es que la variable x se ha definido dentro de un bloque mediante la palabra clave let y no se puede utilizar fuera. Digamos que solo existe esa variable en el ámbito del bloque en el que se declaró.

Sin embargo, este otro código funciona sin errores:

```
{
      var x=9;
}
consolé.log(x);
```

Por lo tanto, la diferencia es que let restringe el uso de la variable al bloque concreto en el que se encuentra. En el caso de var, el ámbito de aplicación es mayor, la variable sobrevive fuera del bloque.

Realmente las variables declaradas con var también tienen restricciones en cuanto a su ámbito de uso. Vamos a ver este código:

```
function f(){
      var x=9;
}
consolé.log(x);
```

Al ejecutar el código, volvemos a producir un error de variable indefinida. Aunque las funciones serán objeto de nuestro estudio más adelante, podemos entender que las funciones también utilizan bloques de código (como siempre, delimitados entre llaves). La diferencia, ahora, es que esos bloques son más restrictivos: incluso las variables definidas por var no pueden utilizarse fuera del bloque definido por una función.

Siendo más formales, se dice que el ámbito de uso de las variables declaradas con let es el bloque, y el ámbito de uso de las variables declaradas con var es el ***contexto de ejecución*** en el que se encuentra la palabra var. Este último ámbito puede ser ***global***, si se declara fuera de toda función, y abarca todo el archivo, o puede ser relativo a una función si se declara dentro de ella.

```
var x=7; //x se declara en ámbito global
function f(){
        var x=9; //x se de declara en el ámbito de la función
}
consolé.log(x);
```

El resultado de este código es la escritura por pantalla del número 7. Realmente hay dos variables llamadas x, la primera tiene ámbito global y la segunda se restringe a la función. Por eso, como console.log(x) está fuera de la función, se usa la x global, que vale siete.

DECLARACIÓN CON CONST

Este término se usa para definir constantes. Funciona bajo las mismas condiciones de ámbito que let. La diferencia es que una variable declarada con ***const*** no puede modificar su valor:

```
const PI=3.14592;
PI=9;
```

La ejecución de este código produce un error:

```
TypeError: Assignment to constant variable
```

No podemos modificar el valor de una variable declarada mediante la palabra const. Aunque se pueden usar tanto minúsculas como mayúsculas para declarar constantes puras, es recomendable usar mayúsculas, ya que así las distinguimos más fácilmente de las variables.

2.3.3 TIPOS DE DATOS PRIMITIVOS

NÚMEROS

A diferencia de otros lenguajes (como C, C++ o Java), JavaScript utiliza un solo tipo de datos para los números, sin importar si el número tiene decimales o no. Los números se escriben tal cual, sin comillas ni símbolos extra. Si tienen decimales, se usa el punto como separador decimal.

let entero=1980;
let decimal=0.21;

El hecho de no diferenciar enteros de números decimales, hace que todos los números ocupen realmente el mismo espacio en memoria: 64 bits.

Otras formas de indicar números

JavaScript admite formatos especiales numéricos. Por ejemplo, admite la notación científica, ideal para indicar números muy grandes o muy pequeños:

const AVOGADRO=6.022e+23; Lo cual significa: $6{,}022 \times 10^{23}$

Por otro lado, no solo se aceptan números decimales, también se aceptan ***números hexadecimales*** si se les antepone el prefijo ***Ox***.

```
let hexa=0xAB12;
console.log(hexa);
```

El resultado es el 43794, resultado de convertir a decimal el número hexadecimal AB12.

También es posible usar ***números octales***. Éstos usan el cero como prefijo seguido de la letra o minúscula (**0o**):

```
let octal=0o27652;
console.log(octal);
```

Ahora aparece el número 12202, resultado de convertir a decimal el octal 27652.

Por si fuera poco, podemos también indicar **números en binario** mediante el prefijo **0b** (cero seguido de b).

```
let binario=0bl0111011;
console.log(binario);
```

El número binario se convertirá al decimal correspondiente (187).

Números especiales
JavaScript permite utilizar el número infinito (**Infinity**). Ejemplo:

```
var x=1/0;
console.log(x);
```

Por pantalla aparece lnfinity, ya que cualquier número dividido por cero provoca ese resultado. El término lnfinity es reconocido de forma independiente, así:

```
var y=Infinity;
console.log(y+1);
```

Este código también produce la escritura de lnfinity, porque sumar uno al número infinito, sigue resultando el propio infinito. Lo único que demuestran estos resultados es que el infinito se maneja perfectamente por parte de JavaScript.

Hay otro término especial. Veamos un ejemplo para entenderlo:

```
var x="Hola" * 3;
console.log(x);
```

Lo lógico sería pensar que la primera línea provoca un error de ejecución de código, al intentar operar de forma numérica con un texto. Pero no es así. Lo que resulta de ese

código es que, por pantalla, aparezca el texto **NaN**. Es más, podemos afirmar que x vale NaN.

NaN es el acrónimo de Not a Number (No es un Número), que se provoca cuando una expresión intenta operar de forma numérica un valor que no es un número. Es muy interesante que JavaScript aporte este término directamente, ya que facilita mucho la validación de errores.

Las indeterminaciones matemáticas también provocan este valor. Por ejemplo, la expresión 0/0 (cero partido por cero) o Infinity - Infinity. Estas habilidades matemáticas de JavaScript permiten resolver de forma muy notable problemas complejos sobre números.

STRINGS

Los strings son cadenas de caracteres. En definitiva, textos. Desde su primera versión, JavaScript permite delimitar textos con comillas dobles o simples. Así, estas expresiones son equivalentes:

texto="Hola a todo el mundo";
texto='Hola a todo el mundo';

Y esto permite que dentro del texto delimitado puedan aparecer comillas dobles o simples.

frase="Mi apellido es O'Donnell";
frase2= 'José se acercó y dijo "Hola" ';

Desde la versión 6 se permiten también las comillas invertidas (conocidas como backticks en inglés):

texto= `Hola a todo el mundo`;

Uso de plantillas de string

La tercera forma de delimitar textos, con el uso de comillas invertidas, permite, de forma muy sencilla, colocar expresiones dentro de las comillas. Vamos a explicar la idea:

Si hacemos algo como esto:

var nombre='Jose';
console.log("Me llamo nombre");

Por pantalla, la ejecución del código escribe Me llamo nombre. Para escribir ***Me llamo Jose***, ya que la variable nombre contiene el texto Jose. Pero, entre comillas, el texto se toma de forma literal. Hay una solución gracias al operador de concatenación (+).

var nombre="Jose";
console.log("Me llamo " + nombre);

Ahora sí se escribe Me llamo Jose.

Aunque el operador de concatenación es muy interesante, si el texto a escribir es complejo, es pesado tener que encadenar expresiones con el operador +.

Las plantillas de string aportan otra solución. Se llaman plantillas de string (***String templates***) a los textos delimitados con comillas invertidas. La cuestión es que se admite incluir el signo $ (dólar) dentro del texto delimitado y después, entre llaves, indicar una expresión JavaScript. El texto dentro de las llaves no se tomará como texto literal, se interpretará literalmente como si estuviera fuera de las comillas. Ejemplo:
var nombre="Jose";
console.log(`Me llamo ${nombre}`);

Nuevamente veremos Me llamo Jose.

Podemos incluso indicar expresiones más complejas:
console.log(`Cada día hay ${24*60*60} segundos`);

Escribe:
Cada día hay 86400 segundos.

Secuencias de escape

Hay caracteres de no se pueden escribir y que, sin embargo, son muy necesarios. Es el caso del salto de párrafo (relacionado con la tecla Intro) o el fabulador. Se consiguen escribir esos caracteres usando un código especial que comienza con la barra inclinada a la izquierda (backslash) seguida del código concreto. Por ejemplo:
let texto="Una línea\nOtra línea";

Por consola, aparece el texto **Una línea** en la primera línea y debajo el texto **Otra línea**.
Algunas de las secuencias más utilizadas son:

SECUENCIA	SIGNIFICADO
\n	Salto de línea
\t	Tabulador
\r	Retorno de carro
\f	Salto de página
\v	Tabulador vertical
\"	Comillas dobles
\'	Comillas simples
\b	Retroceso
\\	El propio carácter ***backslash***

Ejemplo de uso de comillas usando secuencias de escape:
let texto="Paseando por la calle O'Donell dijo \" ¡Qué calor!\" ";

Es necesario usar las comillas dobles del texto "¡Qué calor!" como secuencia de escape, para distinguirlas de las comillas dobles que sirven para delimitar el texto.

BOOLEANOS

Los valores booleanos solo pueden tomar los valores true (verdadero) o false (falso). Estos valores son palabras reservadas que se pueden asignar directamente:
let b=true;

Sin embargo, suele ser más habitual que esos valores se produzcan al evaluar una expresión lógica:

```
let x=9;
let y=10;
let b=(x>y);          //b vale false porque x no es mayor que y
```

Incluso podemos decidir que toda expresión y valor en JavaScript se asocia con valores verdaderos o falsos. Para poder comprobar que esto es cierto, podemos usar la función Boolean (escrita así con la primera letra mayúscula) que escribe el valor booleano equivalente para cualquier valor:

```
consolé.log(Boolean(true));                  //Escribe      true
consolé.log(Boolean(1));                     //Escribe      true
consolé.log(Boolean(0));                     //Escribe      false
consolé.log(Boolean( 'Hola"));               //Escribe      true
consolé.log(Boolean( ));       " "           //Escribe      false
consolé.log(Boolean(NaN));                   //Escribe      false
consolé.log(Boolean(undefined));                          //Escribe false
consolé.log(Boolean(Infinity));                           //Escribe true
consolé.log(Boolean(nul1)); //Escribe false
```

En realidad, solo los textos vacíos (comillas sin contenido), el valor cero, el valor false, el valor NaN, el valor undefined y el valor nulo (null), se consideran falsos.

2.4 OPERADORES

2.4.1 OPERADORES ARITMÉTICOS

Los operadores aritméticos permiten realizar operaciones matemáticas sobre los números. Son los siguientes:

OPERADOR	SIGNIFICADO	EJEMPLO
+	Suma	consolé.log(5 + 4.5); //Escribe 9.5
-	Resta	consolé.log(5 - 4.5); //Escribe 0.5
*	Multiplicación	consolé.log(5 * 4.5); //Escribe 22.5
/	División	consolé.log(7 / 3); //Escribe 2.333333
%	Resto	consolé.log(7 ***% 3);*** //Escribe 1
**	Exponente	consolé.log(3 ** 2); //Escribe 9

2.4.2 OPERADORES RELACIONALES

Los operadores relaciónales permiten comparar expresiones. El resultado de la operación es siempre un valor booleano. Son:

OPERADOR	SIGNIFICADO	EJEMPLOS

>	Mayor	consolé.log(5 > 4); //Escribe true
		consolé.log(5 > 5); //Escribe false

<	Menor	consolé.log(5 < 4); //Escribe false
		consolé.log(5 < 5); //Escribe false

>=	Mayor o igual	consolé.log(5 >= 4); //Escribe true
		consolé.log(5 >= 5); //Escribe true

<=	Menor o igual	consolé.log(5 <= 4); //Escribe false
		consolé.log(5 <= 5); //Escribe true

==	Igual	consolé.log(5 == 4); //Escribe false
		consolé.log(5 == 5); //Escribe true

!=	Distinto	consolé.log(5 != 4); //Escribe true
		consolé.log(5 != 5); //Escribe false

Un error muy habitual es confundir el operador de asignación (=) con el operador de valoración de igualdad (==). El primero permite asignar un valor a una variable, el segundo compara y devuelve verdadero o falso en función de si se cumple o no la igualdad. Veamos este ejemplo:
console.log(5=4);

Esa expresión provoca un error de ReferenceError: ***Invalid left-hand side in assignment***, referido a que el lado izquierdo de la asignación (el número 5) no permite realizar una asignación. Sin embargo:
console.log(5==4);

Ahora se mostrará el valor false porque 5 no es igual a 4.

Podemos comparar strings:
console.log("saco">"saca");

Escribirá true, porque, en orden alfabético, saco es mayor que saca. Sin embargo esta expresión:
console.log("Saco' >"saca");

Escribe, para nuestra sorpresa, false. El cambio es que, ahora, hemos usado la "S" mayúscula y las mayúsculas son menores que las minúsculas según el código que se asigna en la tabla Unicode.

El mismo problema ocurre con algunos símbolos:
console.log("ñ" <"o"); //Escribe false

A la eñe se le asigna un código mucho mayor que el que se le asigna a la letra o, lo que invalida estos operadores para ser utilizados con textos en español. La comparación correcta se debe realizar con el método l***ocalCompare***

COMPARACIONES ESTRICTAS

JavaScript posee un operador de comparación estricta que utiliza tres signos de igual (===). La razón procede de la capacidad de JavaScript de comparar datos de diferente tipo:
console.log("2"==2);

Esa expresión escribe true, porque aunque "2" es un texto, JavaScript puede comparar de forma numérica el contenido de ese texto. Sin embargo:
console.log("true"==true); //Escribe false

No funciona con todo tipo de datos, el texto "true" no se puede convertir a su equivalente booleano. Pero hay más sorpresas:
console.log(1==true); //Escribe true

Estamos comparando un número con un valor booleano, el resultado es true porque se considera que 1 es perfectamente convertible al valor true. Algo que ocurre con el cero también:
console.log(0==false); //Escribe true

Pero no con otros números:
console.log(2==true); //Escribe false
console.log(2==false); //También escribe false

Se debe esta situación a compatibilizar el código con la forma de usar true y false con uno y cero en algunos lenguajes (como el lenguaje C). Incluso funciona esta comparación:
console.log("1"==true); //Escribe true

Y también comparaciones curiosas como:
console.log(undefined==null); //Escribe true

Pero no:
console.log(únelefined==false); //Escribe false

Dejando de lado tantas posibilidades, hay que tener en cuenta que en JavaScript es muy habitual que recojamos números que el usuario escribe y estos números se reciben como strings. Por eso viene bien que esta expresión devuelva verdadero:
console.log("2"==2); //Escribe true

Pero puede ser que necesitemos comparar de forma más estricta, y eso es lo que consigue el operador **===**. Este operador compara el valor y el tipo de dos expresiones, ambas condiciones tienen que coincidir para que el operador devuelva true:

console.log("2"===2); //Escribe false
console.log(1+1===2); //Escribe true

1+1 suma dos números, por eso el resultado es el número 2. Sin embargo "2" en realidad es un texto (un string) por eso, al compararlo con el número 2, devuelve false. Hay también un operador estricto para valorar si una expresión es diferente de otra. Se trata del operador **!==**
console.log("2"!=2); //Escribe false
console.log("2"!==2); //Escribe true

La primera sentencia escribe false porque considera iguales las expresiones "2" y 2. Sin embargo, el operador de diferencia estricta escribe true porque entiende que el dos no es igual como texto que como número.

2.4.3 OPERADOR DE ENCADENAMIENTO

El signo más (+) en JavaScript tiene dos posibilidades. Ya hemos visto que sirve, como es lógico, para sumar números. Sin embargo, también sirve para unir textos en una operación que se conoce como encadenar. Ejemplo:

```
let hoy="Hoy es  ";
let dia='Viernes";
console.log(hoy + dia);
```

Por pantalla aparece Hoy es viernes. El signo más, en este caso, sirve para unir los textos. El único problema, como veremos más adelante, es tener cuidado de delimitar muy bien cuándo sumamos y cuándo encadenamos.

2.4.4 OPERADORES LÓGICOS

Las expresiones lógicas permiten valorar condiciones para tomar decisiones en nuestro programa. Los operadores lógicos permiten poder utilizar expresiones lógicas complejas. Estos operadores son:

OPERADOR	SIGNIFICADO
!	NOT (no)
&&	AND ("y" lógico)
||	OR ("o" lógico)

El **operador AND (&&)** permite comparar dos expresiones lógicas. El resultado será verdadero si las dos expresiones que se comparan lo son:

```
let x=9;
let y=10;
console.log(x==9 && y==10); //true
```

Este código escribe por consola el valor true, ya que se cumple que x vale 9 y que y vale 10. Si cualquiera de esas expresiones cambiara el resultado sería falso:

let x=9;
let y=12;
console.log(x==9 && y==10); //false

El **operador OR (||)** también compara dos expresiones pero, en este caso, devuelve true si cualquiera de las dos expresiones es verdadera:
let x=9;
let y=12;
console.log(x==9 || y==10); //true

Finalmente, el **operador NOT (!)** niega la expresión que tenga a su derecha: si es falsa devuelve true y si es verdadera, devuelve false:
let x=9;
let y=12;
console.log(!(x==9 || y==10)); //false

Hay que tener cuidado con este operador, ya que tiene más prioridad que los operadores AND y OR, por lo que si en el código anterior retiramos los paréntesis:
let x=11;
let y=12;
console.log(!x==9 || y==12); //true
La expresión, en este caso, x==9 devolvería false, pero como está matizada con el operador NOT, devuelve true, luego se evalúa la expresión y==12 (que es verdadera) y luego se une ambas con el operador OR, que resulta otra vez true.

Un hecho interesante del operador AND es que la segunda expresión solo se evalúa si la primera es verdadera, ya que si la primera es falsa ya sabemos que el resultado será falso. Es decir:
let edad=17;
let conduce=(edad>=18 && carnet==true);

La variable ***conduce*** será falsa porque la ***edad*** (en la línea de arriba) es de 17. Es más, a lo mejor ni hemos declarado la variable carnet, lo que, en principio, provocaría un error por indefinición de variable. Pero la realidad es que no se va a evaluar la segunda expresión porque la edad no es mayor o igual a 18, y sabemos que el resultado de la expresión es falso independientemente del valor de la segunda parte de la expresión.

2.4.5 OPERADORES DE ASIGNACIÓN

En JavaScript (al igual que ocurre con otros lenguajes), es muy habitual tener que realizar esta operación:
x=x+1;
Por ello, se dispone de esta forma más rápida:
x++;

De tal manera que conseguimos rápidamente este resultado:
let x=8;
x++;
console.log(x); //escribe 9

El operador ++ se llama incremento. También disponemos del operador -- para el decremento:
let x=8;
x--;
console.log(x); //escribe 7

Estos operadores se pueden colocar por detrás de la variable (x++) o por delante (++x).

```
let valor=50, incremento, decremento;
//Ejemplo de postincremento, primero se asigna y luego se
//incrementa la variable
incremento=valor++; //incremento vale 50 y valor vale 51
de□remento=valor—; //decremento vale 51, valor vale 50
valor=50;
//Ejemplo de preincremento, primero se incrementa
//y luego se asigna la variable
incremento=++valor; //incremento vale 51 y valor también
decremento=—valor; //decremento y valorl valen 50
```

En el ejemplo se puede observar la diferencia. En el caso del ***postincremento (x++)*** o ***postdecremento (x--)*** si es parte de otra expresión el incremento o decremento se hace después. Es decir, en la expresión y=x++ la variable y tomaría el valor que tenga x (sin incrementar) y luego se incrementaría x, pero la variable "y", ya no se modifica.
En el caso de los ***preincrementos (++x)*** o ***predecrementos (--x)*** siempre se realiza primero la operación de incrementar o decrementar. De ahí los resultados en el código anterior

OTROS OPERADORES DE ASIGNACIÓN

Hay otros operadores que permiten abreviar las operaciones de asignación:
var x=18;
x+=9; //x vale ahora 27

La expresión x+=9 sirve para abreviar la expresión x=x+9. Es un operador de asignación y suma. La lista completa de operadores de asignación es:

OPERADOR	SIGNIFICADO	EJEMPLO	EQUIVALENTE A...
+=	Suma y asignación	x+=5	x=x+5
-=	Resta y asignación	x-=5	x=x-5
*=	Multiplicación y asignación	x*=5	x=x*5
/=	División y asignación	x/=5	x=x/5
%=	Resto y asignación	x%=5	x=x%5
**=	Exponente y asignación	x**=2	x=x**2
&=	AND de bit y asignación	x&=65	x=x&65
|=	OR de bit y asignación	x | =65	x=x|65
^=	XOR de bit y asignación	x^=65	x=x^65
>>=	Desplazamiento derecho y asignación	x>>=3	x=x> >3
<<=	Desplazamiento izquierdo y asignación	x< <=3	x=x< <3

OPERADOR CONDICIONAL

Se trata de un operador capaz de evaluar una expresión booleana, y si ésta es verdadera dar como resultado un valor y si es falsa dar otro. A este operador se le conoce también como **inline if** porque su labor se asemeja mucho a la que realiza la estructura if. La sintaxis de uso de este operador es la siguiente:

```
condición ? valorSiVerdadera : valorSiFalsa
```

Ejemplo de uso:

```
let tipo = (numero%2==0) ? "par" : "impar";
```

2.4.6 OPERADORES DE BIT

La realidad de la computación es que solo maneja números binarios. Por lo que, cualquier número, internamente, se almacena de forma binaria. JavaScript aporta operadores para trabajar a nivel de BIT.

OPERADOR	SIGNIFICADO
~	NOT
&	AND
|	OR
^	XOR
>>	Desplazamiento derecho
<<	Desplazamiento izquierdo

El operador **NOT (~)**:
console.log(~25); //Escribe -26
En binario (con 32 bits), el 25 se escribe: 00000000000000000000000000011001

El operador NOT invierte los números: 11111111111111111111111111100110 y eso, en decimal, se interpreta como -26 (ya que se usa complemento a dos como representación binaria).

El operador **AND (*&*)**:
console.log(26 & 17); //Escribe 16

```
26 binario:          00000000000000000000000011010
17 binario:          00000000000000000000000010001
-----------------------------------------------
Resultado AND:       00000000000000000000000010000        (16 en decimal)
```

El operador AND de bits opera de manera individual con cada bit de modo que el resultado es 1 si ambos bits valen 1.

El operador **OR (|)**:
console.log(26 | 17); //Escribe 27

```
26 binario:          00000000000000000000000011010
17 binario:          00000000000000000000000010001
-----------------------------------------------
Resultado OR:        00000000000000000000000011011        (27 en decimal)
```

OR opera con cada bit y asigna un uno en cada bit como resultado si cualquiera de los bits con los que opera es un uno.

El operador **OR exclusivo o XOR (^)**:
console.log(26 ^ 17); //Escribe 11

```
26 binario:          00000000000000000000000011010
17 binario:          00000000000000000000000010001
-----------------------------------------------
Resultado XOR: 00000000000000000000000001011        (11 en decimal)
```

XOR retorna en cada bit un uno si uno de los dos bits es un uno, pero no hay dos unos a la vez.

El operador **desplazamiento de bit a la izquierda (<<)**:
console.log(9<<3); //Escribe 72

```
9 binario: 00000000000000000000000001001
---------------------------------------
9 << 3:          00000000000000000000001001000        (72 en decimal)
```

EL operador << añade el número de bits con valor cero a la derecha que se indiquen. En este ejemplo se han añadido tres ceros a la derecha y eso es como multiplicar tres veces por 3 al número.

El operador **desplazamiento de bit a la derecha (>>)**:
console.log(9>>3); //Escribe 1

```
9 binario: 00000000000000000000000001001
---------------------------------------
9 >> 3:          00000000000000000000000000100T        (1 en decimal)
```

Funciona al revés. En la práctica el resultado es equivalente a dividir tres veces por dos. Estos operadores de desplazamiento cada vez se usan menos.

2.5 CONVERSIÓN DE TIPOS

2.5.1 CONVERSIÓN AUTOMÁTICA

JavaScript no es un lenguaje muy **tipado**. Lo que significa que, a diferencia de lenguajes más fuertemente tipados como Java, no se producen errores con facilidad al mezclar tipos diferentes de datos. El lenguaje JavaScript se basa en buscar la lógica más razonable de cada expresión.
Ejemplo:
console.log('2' * 3);

Esta expresión produciría un error en muchos lenguajes de programación al intentar multiplicar un número por un string. Sin embargo, JavaScript realiza conversiones automáticas de tipos de datos, siempre que sea posible. Gracias a ello, el resultado de esa operación es 6.

Pero observemos este código:
console.log('2' + 3);
Escribirá 23. Ahora tenemos un operador capaz de sumar, cuando opera con números, y de encadenar, cuando opera con strings. En este caso, la conversión automática opera con el número 3, que se convertirá en un string para poder aplicar el encadenamiento. El resultado sería el mismo en este caso:
console.log(2+'3');

Cuando el signo más tiene un string alrededor, JavaScript entenderá que debe encadenar y no sumar.

Por supuesto no todo es convertible:
console.log('Hola' * 3);

No se produce error, pero se escribe el valor NaN porque no se puede conseguir un valor numérico de esa expresión. Pero vemos claramente que JavaScript se esfuerza por conseguir un resultado coherente sin provocar un error. Y por ello, las siguientes expresiones no provocan error:
console.log(null * 3); //Escribe 0
console.log(true * 3); //Escribe 3 (true se asocia al número 1)
console.log(false * 3); //Escribe 0 (false se asocia al número cero)
console.log(undefined * 3); //Escribe NaN

2.5.2 FORZAR CONVERSIONES

Algunas veces deberemos forzar la conversión de los números. En este ejemplo:
let x='2';
let y='3l;
console.log(x + y);

Como ya hemos visto, se escribe 23 por pantalla. Pero ¿y si quisiéramos sumar de forma numérica? Para eso disponemos de la **función Number**:

```
let x='2';
let y=’3l;
console.log(Number(x) + Number(y)); //Ahora escribe 5
```

Por supuesto no todo se puede pasar a número:

```
console.log(Number("Hola' )); //Escribe NaN
```

Disponemos de una función para convertir a strings. Se trata de la **función String**.

```
let x=2;
let y=3;
console.log(String(x) + String(y)); //Escribe 23
```

Hay una **función** muy poderosa llamada **parseInt** para convertir a números enteros. En este sentido es como Number, pero con capacidad solo de convertir a números no decimales. Sin embargo, permite cambiar de base y así pasar, por ejemplo, de números binarios a decimales:

```
console.log(parseInt("101101",2));
```

Tras el número a convertir (101101) se coloca una coma (para separar el segundo parámetro de la función) y el número 2, indicando que el texto original contiene un número en base dos (binaria).

parseInt además tiene la capacidad de extraer el número con el que comienza un texto (el método Number no tiene esta capacidad):

```
console.log(parseInt( 22veces',10));
```

Escribe 22 en lugar de NaN (que es lo que haría la función Number).

Cuando el número está en base decimal, no hace falta explicitar el número 10:

```
consolé.log(parseInt( 22veces')); //Sigue escribiendo 22
```

parseInt no es capaz de coger decimales:

```
console.log(parseInt( 22.5veces")); //Sigue escribiendo 22
```

Pero hay una función llamada parseFloat que sí que tiene esta capacidad:

```
console.log(parseFloat("22.5veces )); //Ahora escribe 22.5
```

2.5.3 FUNCION ISNAN

Hay problemas en JavaScript para determinar si una expresión no es numérica. Aunque el valor NaN es un valor válido en JavaScript, y sabemos que, por ejemplo, "Hola"*4 produce un resultado no numérico. La comparación ("Hola"*4)==NaN produce un resultado falso.

Por ello se utiliza una función llamada isNaN. Esta función recibe una expresión y devuelve verdadero si la expresión no es numérica:

console.log(isNaN(NaN)); //escribe true
console.log(isNaN("Hola"*5)); //escribe true
console.log(isNaN("3"*5)); //escribe false, "3"*5 retorna 15, un número

2.6 CUADROS DE DIÁLOGO DEL NAVEGADOR

Los navegadores permiten utilizar un objeto del sistema llamado window. Vamos a ver los métodos que proporciona este objeto para poder comunicarse con el usuario e incluso requerirle información.

Actualmente en una aplicación web profesional, no se usan estas técnicas al ser mucho más eficiente y versátil el uso de controles de formulario y manejo de componentes para informar al usuario y pedirle datos, pero es un método muy fácil para poder empezar a crear pequeños programas en JavaScript y probar las funcionalidades básicas del lenguaje.

2.6.1 MENSAJES DE ALERTA

Los mensajes de alerta son cuadros que muestran información al usuario. Lo que muestran es un texto que aparece en un pequeño cuadro de diálogo:

```
<!DOCTYPE html>
<html lang="es">
<head>
        <meta charset="UTF-8">
        <title>Mensajes de alerta</title>
</head>
<body>
        <script>
                alert("Hola");
        </script>
</body>
</html>
```

Podríamos imprimir todo tipo de mensajes de texto:

```
<script>
  let nombre=”Jose”;
  alert(` Mi nombre es ${nombre}`);
</script>
```

En la ventana aparecerá el texto Mi nombre es Jose.

2.6.2 CUADROS DE CONFIRMACIÓN

El método encargado de esta tarea es **confirm**. Al igual que ocurre con alert, confirm requiere de un texto que será el mensaje que mostramos al usuario. El cuadro es distinto porque admite aceptar o no el mensaje. Para saber lo que ha elegido el usuario, el resultado de este método es el valor true o false. Ejemplo:

```
<script>
    let resultado=confirm("¿Aceptas nuestras condiciones?");
    consolé.log(resultado);
</script>
```

Se mostrará este cuadro:

Después se mostrará por consola el valor true si el usuario acepta el cuadro y false si cancela el mensaje.

2.6.3 CUADROS DE ENTRADA DE TEXTO

El cuadro más completo es **prompt**. Es un cuadro que muestra un mensaje y, además, recoge lo que el usuario escriba en él (el cuadro muestra un control de texto para que el usuario escriba, si lo desea, en él). Ejemplo:

```
<script>
    let resultado=prompt("Escribe tu nombre");
    alert(Tu nombre es $¡resultado ')
</script>
```

Inicialmente aparecerá este cuadro:

El método prompt admite indicar un valor por defecto. Por ejemplo:

```
let resultado=prompt("Escribe tu edad",18);
```

Se mostrará un cuadro en el que, por defecto, ya aparece el valor 18.

2.7 CONTROL DEL FLUJO DEL PROGRAMA

2.7.1 INTRODUCCIÓN

Hasta ahora las instrucciones, que hemos visto, son instrucciones que se ejecutan secuencialmente. Es decir, podemos saber lo que hace el programa leyendo las líneas de izquierda a derecha y de arriba abajo.

Las **instrucciones de control de flujo** permiten alterar esta forma de ejecución. El uso de esas instrucciones propicia que haya líneas en el código que se ejecutarán o no dependiendo de una condición. Incluso lograremos que se puedan repetir continuamente una serie de instrucciones mientras se siga cumpliendo la condición que deseemos. Esa *condición* se construye utilizando lo que se conoce como expresión lógica. Una **expresión lógica** es cualquier tipo de expresión válida en JavaScript que proporcione un resultado booleano (verdadero o falso). Las expresiones lógicas se construyen usando variables booleanas, o bien escribiendo expresiones que combinen los operadores relaciónales (==, >, <,...) y lógicos (&&, || , !) vistos anteriormente.

2.7.2 NÚMEROS ALEATORIOS

Los números aleatorios en JavaScript requieren del uso del objeto Math. Para generar números aleatorios en JavaScript se usa el método random del objeto Math de esta forma:

Math.random();

Esa expresión da como resultado un número decimal entre cero y uno. Si queremos números más grandes, esta otra expresión:

Math.random()*2;

Obtiene como resultado un número entre el cero y el dos (siendo casi imposible que salga el número 2). Si deseamos un número decimal entre 6 y 10, la expresión correcta es:

Math.random()*4+6;

Se multiplica por el número 4 porque el rango entre 6 y 10 es 4. Se suma el número 6 para que el rango devuelto empiece por el número 6.

Si deseamos números enteros, debemos ayudarnos de la ***función parselnt***. Esta función elimina los decimales. Calcular un número aleatorio del 0 al 10 (incluidos ambos) sería:

parselnt(Math.random()*11);

No multiplicamos por diez, sino por once. La razón es que Math. random( )*10 devuelve números entre cero y diez, pero sin pasar de 10. Es imposible que aparezca el 10. Por ello, hay que sumar uno más para que los números vayan de 0 a 10,999999.... y así al retirar los decimales con parselnt aparezca el rango deseado.

Si deseamos números enteros entre el 6 y el 10, la expresión sería:

parseInt(Math.random()*5)+6;

Math.random()*5 devuelve números entre el cero y el cinco (sin llegar a cinco). Usando parselnt son enteros del cero al cuatro (al quitar los decimales ya no llegamos a cinco). Si sumamos seis a este resultado, tenemos enteros del seis al diez.

2.7.3 INSTRUCCIÓN IF

Se trata de una sentencia que, tras evaluar una expresión lógica, ejecuta una serie de instrucciones en caso de que la expresión lógica sea verdadera. Si la expresión tiene un resultado falso, no se ejecutarán dichas instrucciones. Las llaves se requieren solo si va a haber varias instrucciones. En otro caso se puede crear el if sin llaves.

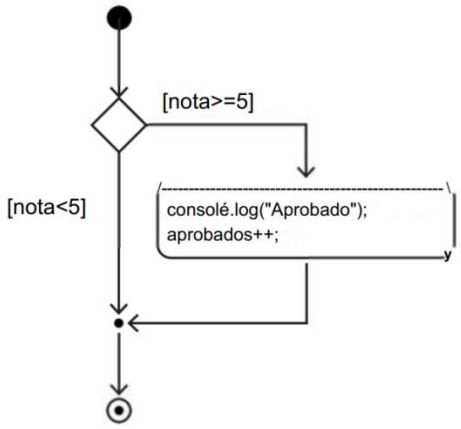
Sintaxis:

```
if(expresión lógica) {
    instrucciones
    ...
}
```

```
if (x==l) {
    instrucciones
...
}
else {
    if(x==2) {
        instrucciones
        ...
    }
    else {
        if(x==3) {
            instrucciones
            ...
        }
    }
}
```

Ejemplo de sentencia **if-else:**

```
i f(nota>=5){
    consolé.log("Aprobado");
    aprobados++;
}
else {
    consolé.log("Suspenso");
    suspensos++;
}
```

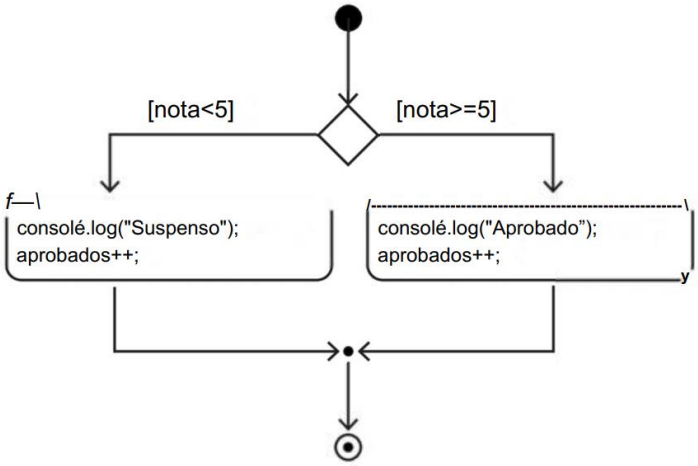


SENTENCIA CONDICIONAL COMPUESTA

Es igual que la anterior, solo que se añade un apartado else que contiene instrucciones que se ejecutarán si la expresión evaluada por el if es falsa. Sintaxis:

```
if(expresión lógica){
   instrucciones
   ….
}
else {
   instrucciones
   …
}
```

```
i f(nota>=5){
    consolé.log("Aprobado");
    aprobados++;
}
else {
    consolé.log("Suspenso");
    suspensos++;
}
```



ANIDACIÓN

Dentro de una sentencia if se puede colocar otra sentencia if. A esto se le llama anidación y permite crear programas donde se valoren expresiones complejas. La nueva sentencia puede irtanto en la parte if como en la parte else**.**

```
if (x==l) {
    instrucciones
…
}
else {
    if(x==2) {
        instrucciones
        …
    }
    else {
        if(x==3) {
            instrucciones
            …
        }
    }
}
```

```
if (x==l) {
    instrucciones que se ejecutan si x vale 1
    …
}
else if (x==2) {
    instrucciones que se ejecutan si x no vale 1 y vale 2
    …
}
else if (x==3) {
    instrucciones que se ejecutan si x no vale ni 1 ni 2 y vale 3
    …
}
else{
    instrucciones que se ejecutan si x no vale ni 1 ni 2 ni 3
}
```

SENTENCIA CASE O SWITCH

Hay momentos en JavaScript en los que podrías considerar usar una sentencia switch en lugar de una sentencia if else. Las sentencias switch pueden tener una sintaxis más limpia que las sentencias if else complicadas. Su sixtaxis es:

```
switch (expresion) {
  case 1:
   //este código se ejecutará si el caso coincide con la expresión
    break;
  case 2:
   //este código se ejecutará si el caso coincide con la expresión
    break;
  case 3:
   //este código se ejecutará si el caso coincide con la expresión
    break;
  default:
    //este código se ejecutará si ninguno de los casos coincide con la expresión
    break;
}
```

La computadora revisará la sentencia switch y verificará la igualdad estricta === entre el caso ***case*** y la expresión ***expression***. Si uno de los casos coincide con la expresión ***expression***, se ejecutará el código dentro de la cláusula del caso case.

Si ninguno de los casos coincide con la expresión, se ejecutará la cláusula predeterminada ***default***.

Si los casos múltiples coinciden con la instrucción switch, se utilizará el primer caso case que coincida con la expresión ***expression***.

Las instrucciones ***break*** se desprenderán del interruptor switch cuando el caso case coincida. Si las sentencias break no están presentes, el equipo continuará a través de las sentencias switch incluso si se encuentra una coincidencia.

Si las sentencias ***return*** están presentes en el switch, no necesita una sentencia break.

Ejemplo:

```
const mascota = "perro";

switch (mascota) {
  case "lagarto":
    console.log("Tengo un lagarto");
    break;
  case "perro":
    console.log("Tengo un perro");
    break;
  case "gato":
    console.log("Tengo un gato");
    break;
  case "serpiente":
    console.log("Tengo una serpiente");
    break;
  case "loro":
    console.log("Tengo un loro");
    break;
  default:
    console.log("No tengo mascota");
    break;
}
```

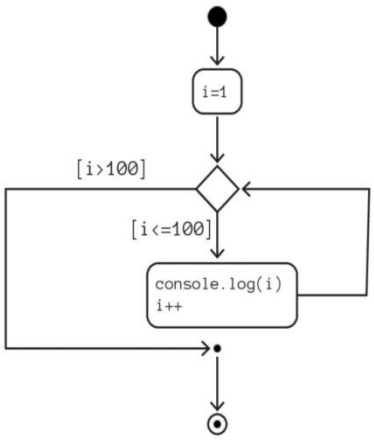
2.7.4 INSTRUCCIÓN WHILE

La instrucción **while** permite crear bucles. Un bucle es un conjunto de sentencias que se repiten mientras se cumpla una determinada condición. Los bucles while agrupan instrucciones las cuales se ejecutan continuamente hasta que una condición que se evalúa sea falsa.
La condición se mira antes de entrar dentro del while y cada vez que se termina de ejecutar las instrucciones del while.

Sintaxis:

```
while (expresión lógica) {
   sentencias que se ejecutan si la condición es verdadera
}
```

```
var i=1;
while (i<=100){
    consolé.log(i);
    i++;
}
```



El programa se ejecuta siguiendo estos pasos:
[1] Se evalúa la expresión lógica.

[2] Si la expresión es verdadera ejecuta las sentencias, si no el programa abandona la sentencia while.

[3] Tras ejecutar las sentencias, volvemos al paso 1.

BUCLES CON CONTADOR

Se llaman así a los bucles que se repiten una serie determinada de veces. Dichos bucles están controlados por un contador (o incluso más de uno). El contador es una variable que va variando su valor (de uno en uno, de dos en dos,...o como queramos) en cada vuelta del bucle. Cuando el contador alcanza un límite determinado, entonces el bucle termina.

En todos los bucles de contador necesitamos saber:
[1] Lo que vale la variable contadora al principio. Antes de entrar en el bucle.

[2] Lo que varía (lo que se incrementa o decrementa) el contador en cada vuelta del bucle.

[3] Las acciones a realizar en cada vuelta del bucle.

[4] El valor final del contador. En cuanto se rebase el bucle termina. Dicho valor se pone como condición del bucle, pero a la inversa; es decir, la condición mide el valor que tiene que tener el contador para que el bucle se repita y no para que termine.

Ejemplo:

```
let i=10; //Valor inicial del contador, empieza valiendo 10

/* condición del bucle, mientras i sea menor de
200, el bucle se repetirá, cuando i rebase este
valor, el bucle termina */

while (i<=200){
    consolé.log(i)
        /*en cada vuelta del bucle, simplemente se escribe el valor
      del contador por consola*/
    i+=10;
        /* Variación del contador, en este caso cuenta de 10 en
            10 */
}
/* Al final, el bucle escribe por consola:
10
20
30
...
y así hasta 200
*/
```

BUCLES DE CENTINELA

Es el segundo tipo de bucle básico. Una condición lógica llamada centinela, que puede ser desde una simple variable booleana hasta una expresión lógica más compleja, sirve para decidir si el bucle se repite o no. De modo que cuando la condición lógica se incumpla, el bucle termina.
Esa expresión lógica a cumplir es lo que se conoce como centinela y normalmente la suele realizar una variable booleana.

Ejemplo:

```
let salir=false; /* En este caso el centinela es una variable
            booleana que inicialmente vale falso */
let n;
while(salir==false){
      /* Condición de repetición: que salir siga siendo falso.
          Ese es el centinela.
          También se podía haber escrito simplemente:
              while(!sal ir)
   */
   n= parseInt(Math.random()*5)+l; // Lo que se repite en el bucle:
  consolé.log(i);                                  // calcular un número aleatorio de 1 a
                                                   // 5 y escribirlo por consola
   salir=(i%7==0); /* El centinela vale verdadero si el número
                                  es múltiplo de 7*/
                                  saldremos*/
```

- Los bucles de contador se repiten un número concreto de veces, los bucles de centinela no.
- Un bucle de contador podemos considerar que es seguro que finalice, el de centinela puede no finalizar si el centinela jamás varía su valor (aunque, si está bien programado, alguna vez lo alcanzará).
- Un bucle de contador está relacionado con la programación de algoritmos basados en series.

Un bucle podría ser incluso mixto: de centinela y de contador. Por ejemplo:

```
let salir = false;  //centinela
let n;
let i=1; //contador
while (salir == false && i<=5) {
    n = parseInt(Math.random() * 500 + 1);
    console.log(n);
    i++;
    sal ir = (n % 7 == 0);
}
consolé.log("Último número "+n)
```

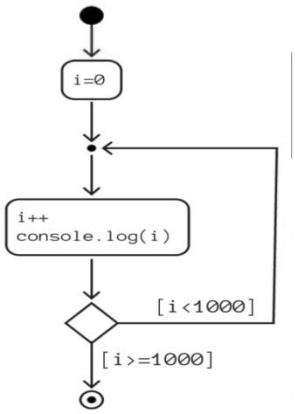
2.7.5 INSTRUCCIÓN DO.. WHILE

La única diferencia respecto a la anterior está en que la expresión lógica se evalúa después de haber ejecutado las sentencias. Es decir, el bucle al menos se ejecuta una vez. Los pasos son:

[1] Ejecutar sentencias.
[2] Evaluar expresión lógica.
[3] Si la expresión es verdadera volver al paso 1, si no continuar fuera del while.

Sintaxis:

```
do {
   instrucciones
} while (expresión lógica)
```



Ejemplo (contar de uno a 1000):

```
let i=0;
do {
    i++;
    consolé.log(i);
} while (i<1000);
```

Se utiliza cuando sabemos al menos que las sentencias del bucle se van a repetir una vez (en un bucle **while** puede que incluso no se ejecuten las sentencias que hay dentro del bucle si la condición fuera falsa, ya desde un inicio).

2.7.6 INSTRUCCIÓN FOR

Es un bucle más complejo especialmente pensado para rellenar arrays o para ejecutar instrucciones controladas por un contador.
Una vez más se ejecutan una serie de instrucciones en el caso de que se cumpla una determinada condición.

Sintaxis:

```
for(inicialización;condición;incremento){
  sentencias
}
```

Las sentencias se ejecutan mientras la condición sea verdadera. Además, antes de entrar en el bucle se ejecuta la instrucción de inicialización y en cada vuelta se ejecuta el incremento. Es decir, el funcionamiento es:

[1] Se ejecuta la instrucción de inicialización

[2] Se comprueba la condición

[3] Si la condición es cierta, entonces se ejecutan las sentencias. Si la condición es falsa, abandonamos el bloque for

[4] Tras ejecutar las sentencias, se ejecuta la instrucción de incremento y se vuelve al paso 2

Ejemplo (contar números del 1 al 1000):

```
for(let i=l;i <=1000;i++){
    consolé.log(i);
}
```

2.7.7 ABANDONAR UN BUCLE

JavaScript proporciona una instrucción llamada ***break*** que permite abandonar de golpe un bucle:

```
let m=parseInt(Math.random()*10)+1;
for(let i=l;i <=1000;i++) {
      consolé.log('i=$ i , m=$ m');
      if(m==10){
            break;
      }
      m=parseInt(Math.random()*10)+1;
}
```

Posible salida:

```
i=l,    m=7
i=2,    m=l
i=3,    m=8
i=4, m=10
```

Este extraño bucle cuenta crea un contador (i) que empieza valiendo uno y termina valiendo 1000 y cuenta de uno en uno. Sin embargo, otra variable llamada m, calcula un número aleatorio del uno al 10. La instrucción if comprueba si el número aleatorio calculado realmente es el 10, de ser así, interrumpe el bucle mediante la instrucción break.

La instrucción break puede ser un atajo muy interesante para salir de un bucle de forma creativa, pero también, si se usa demasiado y con poco cuidado, puede producir malos hábitos. Usar break crea código desestructurado, en el que es difícil reconocer fácilmente cuál es su flujo de trabajo.

2.7.8 BUCLES ANIDADOS

Es posible crear un bucle dentro de otro, en una operación que se llama anidamiento. Ejemplo:

```
let numero; let triangulo="";
numero=prompt("Introduce un número:");
if (!isNaN(numero) && numero>0){
  for(let i=1;i<=numero;i++){
    for(let j=1;j<=i;j++) {
        //document.write("* ");
        triangulo=triangulo+"* ";
    }
    //document.write("<br>");
    triangulo=triangulo+"<br>";
  }
  document.write(triangulo);
}
```

Es este ejemplo encontramos dos bucles for anidados.