TEMA 1. FUNDAMENTOS DE JAVASCRIPT.

1. INTRODUCCIÓN.

JavaScript se diseñó con una sintaxis similar al lenguaje C y aunque adopta nombres y convenciones del lenguaje Java, éste último no tiene relación con JavaScript ya que tienen semánticas y propósitos diferentes.

JavaScript fue desarrollado originariamente por Brendan Eich, con el nombre de Mocha, el cual se renombró posteriormente a LiveScript y quedó finalmente como JavaScript.

Hoy en día JavaScript es una marca registrada de Oracle Corporation, y es usado con licencia por los productos creados por Netscape Communications y entidades actuales, como la fundaciónMozilla.

2. INTEGRACIÓN DE CÓDIGO JAVASCRIPT CON HTML.

Los navegadores web nos permiten integrar nuestro código JavaScript de varias formas.

Podremos insertar código JavaScript en cualquier lugar de nuestro código HTML, usando las etiquetas <script> </script> y empleando un atributo type indicaremos qué tipo de lenguaje de script estamos utilizando, **aunque no es obligatorio utilizarlo**.

Por ejemplo:

```
<script type="text/javascript">
// El código de JavaScript vendrá aquí.
</script>
```

Esta forma de integrar el código JavaScript dentro de HTML no es recomendable.

Otra forma de integrar el código de JavaScript es añadir un fichero externo que contenga el código de JavaScript. Ésta sería la forma más recomendable, ya que así se consigue una separación entre el código y la estructura de la página web y como ventajas adicionales podrás compartir código entre diferentes páginas, centralizar el código para la depuración de errores, tendrás mayor claridad en tus desarrollos, más modularidad, seguridad del código y conseguirás que las páginas carguen más rápido. La rapidez de carga de las páginas se consigue al tener el código de JavaScript en un fichero independiente, ya que si más de una página tiene que acceder a ese fichero lo cogerá automáticamente de la caché del navegador con lo que se acelerará la carga de la página.

Para añadir un fichero JavaScript tendremos que añadir a la etiqueta script el atributo src, con el nombre y la ruta del fichero que contiene el código de JavaScript. Generalmente los ficheros que contienen texto de JavaScript tendrán la extensión .js.

Por ejemplo:

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

Si necesitas cargar más de un fichero .js repetiremos la misma instrucción cambiando el nombre del fichero. Las etiquetas de <script> y </script> son obligatorias a la hora de incluir el fichero .js. No debemos escribir código JavaScript entre la etiqueta de apertura y cierre.

Para **referenciar el fichero origen .js** de JavaScript dependerá de la localización física de ese fichero. Por ejemplo, en la línea anterior el fichero tucodigo.js deberá estar en el mismo directorio que el fichero .html. Podrás enlazar fácilmente a otros ficheros de JavaScript localizados en directorios diferentes de tu servidor o de tu dominio. Es conveniente recordar que vamos a evitar las rutas absolutas a la hora de referenciar los ficheros .js

Ejemplo:

```
<script type="text/javascript"
src="../js/ejemplo.js"></script>
```

(el fichero `ejemplo.js` se encuentra en el directorio anterior (`../`) al actual, dentro de la carpeta `js/`)

Cuando alguien examine el código fuente de tu página web verá el enlace a tu fichero .js, en lugar de ver el código de JavaScript directamente. Esto no quiere decir que tu código no sea inaccesible, ya que simplemente copiando la ruta de tu fichero .js y tecleándola en el navegador podremos descargar el fichero .js y ver todo el código de JavaScript. En otras palabras, nada de lo que tu navegador descargue para mostrar la página web podrá estar oculto de la vista de cualquier programador.

A veces te puedes encontrar que tu script se va a ejecutar en un navegador que no soporta JavaScript. Para ello dispones de una etiqueta <noscript>Texto informativo </noscript> que te permitirá indicar un texto adicional que se mostrará indicando que ese navegador no soporta JavaScript.

3. COMENTARIOS EN EL CÓDIGO.

A la hora de programar en cualquier lenguaje de programación, es muy importante que comentes tu código.

Los comentarios son sentencias que el intérprete de JavaScript ignora. Sin embargo, estas sentencias permiten a los desarrolladores dejar notas sobre cómo funcionan las cosas en sus scripts.

Los comentarios ocupan espacio dentro de tu código de JavaScript, por lo que cuando alguien se descargue vuestro código necesitará más o menos tiempo, dependiendo del tamaño de vuestro fichero. Es muy recomendable que documentes tu código lo máximo posible, ya que esto te proporcionará muchas más ventajas que inconvenientes.

JavaScript permite dos estilos de comentarios. Un estilo consiste en dos barras inclinadas hacia la derecha (sin espacios entre ellas), y es muy útil para comentar una línea sencilla. JavaScript ignorará cualquier carácter a la derecha de esas barras inclinadas en la misma línea, incluso si aparecen en el medio de una línea.

Ejemplos de comentarios de una única línea:

```
// Este es un comentario de una línea
 let nombre="Marta" // Otro comentario sobre esta línea
// Podemos dejar, por ejemplo
//
  // una línea en medio en blanco
```

Para comentarios más largos, por ejemplo, de una sección del documento, podemos emplear en lugar de las dos barras inclinadas el /* para comenzar el comentario y */ para cerrar la sección de comentarios.

Por ejemplo:

```
/* Ésta es una sección de comentarios
  en el código de JavaScript */
```

O también:

```
/* ----------------------------------
function imprimir(){
}
Imprime el listado de alumnos en orden alfabético
------------------------------------*/
const imprimir() => {
     // Líneas de código JavaScript
  }
```

4. VARIABLES Y CONSTANTES.

Una variable es un espacio en memoria, que almacena una información, que podemos utilizar durante la ejecución de nuestro código JavaScript.

Para utilizar una variable, hay que declararla, en JavaScript podemos utilizar una variable sin haberla declarado, pero esto es algo que debemos evitar.

Para declarar una variable vamos a utilizar la palabra reservada var o let seguida de un identificador, que es muy importante que sea significativo.

```
let edad;
```

Otra forma de declarar una variable es darle un valor en la declaración(inicialización)

```
let edad = 38;
```

JavaScript no es un lenguaje tipado, por lo que no tendremos que indicar el tipo de información que vamos a guardar en una variable, es más, vamos a poder guardar informaciones de diferentes tipos en la misma variable.

```
let valor = 38;
valor = "Hola";
valor = 3.58;
```

Esto que acabamos de hacer es válido, aunque no aconsejable.

A la hora de dar nombres a los identificadores de las variables, tendremos que utilizar nombres que realmente describan el contenido de la variable. No podremos usar palabras reservadas, ni símbolos de puntuación en el medio de la variable, ni espacios en blanco. Los nombres de las variables han de construirse con caracteres alfanuméricos y el carácter subrayado (_).

Es conveniente utilizar let en lugar de var, ya que let no nos permite declarar dos variables con el mismo identificador, mientras que var si.

Cuando vamos a utilizar una variable que no va a cambiar de valor el ECMAScript 6 (ES6), recomienda la utilización de constantes. Es importante tener en cuenta que no vamos a poder cambiar su valor y que tenemos que iniciar la constante en la declaración.

```
const nombre = "pepito" // En este caso la cte nombre no va a
cambiar durante el programa.
```

5. ENTRADA SALIDA BÁSICA.

En este punto vamos a ver algunos cuadros de diálogo que nos van a permitir empezar a interactuar con el usuario:

5.1. ALERT

Es un método que pertenece al objeto window, que veremos más adelante.

El método window.alert() (también podemos escribirlo alert()) muestra un diálogo de alerta con contenido opcional especificado y un botón OK (Aceptar).

```
alert(message);
```

- ✓ **message** es un valor opcional del texto que se desea mostrar en el diálogo de alerta.

5.2. CONFIRM

El método confirm() muestra una ventana de diálogo con un mensaje opcional y dos botones, Aceptar y Cancelar.

```
let result = confirm(message);
```

- ✓ **message** es la cadena que se muestra opcionalmente en el diálogo.
- ✓ **result** es un valor booleano indicando si se ha pulsado Aceptar o Cancelar (Aceptar devuelve true y Cancelar devuelve false).

5.3. PROMPT

El método prompt() muestra un diálogo con mensaje opcional, que solicita al usuario que introduzca un texto.

```
let result = prompt(message, default);
```

- ✓ **result** es una cadena de texto que contiene el valor introducido por el usuario, o null.
- ✓ **message** es una cadena de texto que se mostrará al usuario. Este parámetro es opcional y puede ser omitido si no se necesita mostrar nada en la ventana.
- ✓ **default** es una cadena de texto que contiene el valor predeterminado para el texto de entrada.

El valor que devuelve es un valor de tipo cadena, si lo que necesitamos es trabajar con valores numéricos debemos realizar el cambio de tipo cadena al tipo numérico con el que necesitemos trabajar, para ello tenemos dos funciones parseInt y parseFloat, que realizan la conversión a un valor entero y a un valor real.

5.3.1. PARSEINT()

parseInt es una función de alto nivel que nos permite hacer la conversión de valor de tipo cadena de caracteres a un valor numérico de tipo entero y que no está asociada a ningún objeto.

```
let valor_entero = parseInt(cadena);
```

La función parseInt comprueba el primer argumento, una cadena, e intenta devolver un entero.

5.3.2. PARSEFLOAT()

parseFloat es una función de alto nivel, que nos permite hacer la conversión de un valor de tipo cadena de caracteres a un valor numérico de tipo real (float) y que no está asociada a ningún objeto.

```
let valor_real = parseFloat(cadena);
```

Si el primer carácter no se puede convertir a número, parseFloat devuelve NaN. Para fines aritméticos, el valor NaN no es un número para ninguna base. Puede llamar a la función isNaN para determinar si el resultado de parseFloat es NaN. Si se pasa NaN en operaciones aritméticas, la operación resultante también será NaN.

5.4. DOCUMENT.WRITE

Nos va a permitir insertar un contenido que puede ser texto o un elemento HTML dentro de nuestro código HTML.

Podemos mostrar una cadena de caracteres, el contenido de una variable, el resultado de una expresión, el resultado de la ejecución de una función….

La sintaxis es la siguiente:

```
document.write("Hola qué tal???");
```

En el caso de que le estemos pasando una cadena que contenga código HTML, lo va a interpretar y nos va a mostrar la acción que realice el código HTML, en lugar, de la cadena del código.

```
document.write("<h1> hola qué tal???<h1>");
```

En este caso se mostrará por pantalla un encabezado de tipo h1 con el texto ***hola qué tal???***

Hoy en día document.write(), está en desuso no es necesaria ya que se puede reemplazar por otras opciones como ***value*** para valores de formulario ***o textContent (o innerHTML***) para el contenido de etiquetas.

5.5. CONSOLA

Los navegadores incluyen una consola, que es muy utilizada por los programadores web, ya que nos permite realizar tareas de desarrollo.

Algunas de las tareas que nos va a permitir realizar la consola son:

- ✓ Muestra mensajes de información, error o alerta que se reciben al hacer las peticiones para cargar desde la red los elementos incluidos en las páginas.
- ✓ Inspeccionar los elementos de nuestras páginas y depurar código.
- ✓ También permite interactuar con la página, ejecutando expresiones o comandos de JavaScript.
- ✓ Comprobar la velocidad de carga.
- ✓ Simular diferentes resoluciones.
- ✓ …

El propósito general de la consola es poder comprobar el funcionamiento de las páginas o aplicaciones y descubrir posibles errores en el código o en los datos que manejamos.

Para poder trabajar con la consola tenemos que abrirla desde nuestro navegador, para ello debemos acceder al menú y a partir de ahí podremos acceder. La forma de hacerlo cambia un poco en cada navegador, lo que siempre funciona igual en todos los navegadores para abrir la consola es utilizar la tecla F12.

Tenemos diferentes métodos para poder trabajar con la consola, pero el método más utilizado es console.log, ya que nos va a permitir interactuar con la consola desde nuestro código JavaScript.

```
console.log("hola"); // mostrará por consola el texto hola

const nombre="manolito";
console.log (nombre); // mostrará por consola el contenido de la
variable nombre
```

6. TIPOS DE DATOS.

Las variables en JavaScript podrán contener cualquier tipo de dato, los tipos de datos soportados en JavaScript:

TIPOS DE DATOS SOPORTADOS POR JAVASCRIPT		
Tipo	**Ejemplo**	**Descripción**
Cadena	"Hola mundo"	Una serie de caracteres dentro de comillas dobles.
Número	9.45	Un número sin comillas dobles.
Boolean	true.	Un valor verdadero o falso.
Null	null.	Desprovisto de contenido, simplemente es un valor null.

Utilizar solo estos tipos de datos, simplifica mucho las tareas de programación, especialmente aquellas que abarcan tipos incompatibles entre números.

- ✓ Para crear **variables numéricas** también se puede utilizar notación científica. Para ello usamos un exponente (que se define con una letra "e"), seguido a continuación del exponente al cual está elevado. También podremos crear números en bases diferentes (base 8, base 16). Para crear un número en base 8 comenzamos con un 0 seguido de números del 0 al 7. Para los números hexadecimal se pone un 0 seguido de una "x" y cualquier carácter hexadecimal (del 0 al 9 y de la A a la F).

- ✓ Para crear **variables de tipo cadena**, simplemente tenemos que poner el texto entre comillas, pueden ser comillas simples o comillas doble, funciona con las dos.
- ✓ Los tipos de **datos booleanos** solo pueden tomar dos valores: true o false. Y se usan para tomar decisiones. Se muestra un ejemplo de uso de una variable booleana dentro de una sentencia if, modificando la comparación usando == y ===.

```
// Los datos numéricos pueden ser enteros o reales
let miEntero = 33;
let miDecimales = 2.5;
let comaFlotante = 2344.983338;
let numeral = 0.573;

// pueden tener notación científica
let numCientifico = 2.9e3;
let otroNumCientifico = 2e-3;
alert(otroNumCientifico);


// podemos escribir números en otras bases
let numBase10 = 2200;
let numBase8 = 0234;
let numBase16 = 0x2A9F;
alert(numBase16);


// tipo de datos cadena de caracteres
let miCadena = "Hola!!! esto es una cadena";
let otraCadena = "2323232323"; //es una cadena


// caracteres de escape en cadenas
let cadenaConSaltoDeLinea = "linea1\nLínea2\nLínea3";
let cadenaConComillas = "cadena con \"comillas dobles\"";
let cadenaNum = "11";
let sumaCadenaConcatenacion = otraCadena + cadenaNum;
//alert(sumaCadenaConcatenacion);


// tipos de datos booleano
let miBooleano = true;
let falso = false;
if (niBooleano){alert ("era true");
}else{alert("era false");
}
let booleano = (23==="23");
alerts(booleano)

//operador typeof para conocer un tipo
alert("El tipo de miEntero es: " + typeof(miEntero));
alert("El tipo de miCadena es: " + typeof(miCadena));
alert("El tipo de miEntero es: " + typeof(miBooleano));
```

6.1. CONVERSIONES DE TIPOS DE DATOS.

Aunque los tipos de datos en JavaScript son muy sencillos, a veces te podrás encontrar con casos en los que las operaciones no se pueden realizan correctamente, y eso es debido a que los tipos de datos no son compatibles con las operaciones que queremos hacer.

Por ejemplo, cuando intentamos sumar dos números:

```
4 + 5 // resultado = 9
```

Si uno de esos números está en formato de cadena de texto, JavaScript lo que hará es intentar convertir el otro número a una cadena y los concatenará, por ejemplo:

```
4 + "5"       // resultado = "45"
```

Otro ejemplo podría ser:

```
4 + 5 + "6"   // resultado = "96"
```

Esto puede resultar ilógico, pero sí que tiene su lógica. La expresión se evalúa de izquierda a derecha. La primera operación funciona correctamente devolviendo el valor de 9 pero al intentar sumarle una cadena de texto "6" JavaScript lo que hace es convertir ese número a una cadena de texto y se lo concatenará al comienzo del "6".

Para convertir cadenas a números dispones de las funciones: **parseInt()** y **parseFloat():**

Por ejemplo:

```
parseInt("34")          // resultado = 34
parseInt("89.76")          // resultado = 89
```

parseFloat devolverá un entero o un número real según el caso:

```
parseFloat("34")          // resultado = 34
parseFloat("89.76")     // resultado = 89.76


4 + 5 + parseInt("6")        // resultado = 15
```

Si lo que deseas es realizar la conversión de números a cadenas, es mucho más sencillo, ya que simplemente tendrás que concatenar una cadena vacía al principio, y de esta forma el número será convertido a su cadena equivalente:

```
("" + 3400)          // resultado = "3400"
  ("" + 3400).length     // resultado = 4
```

En el segundo ejemplo podemos ver la gran potencia de la evaluación de expresiones. Los paréntesis fuerzan la conversión del número a una cadena. Una cadena de texto en JavaScript tiene una propiedad asociada con ella que es la longitud (length), la cual te devolverá en este caso el número 4, indicando que hay 4 caracteres en esa cadena "3400". La longitud de una cadena es un número, no una cadena.

7. OPERADORES.

JavaScript es un lenguaje rico en operadores, símbolos y palabras que realizan operaciones sobre uno o varios valores, para obtener un nuevo valor.

Cualquier valor sobre el cual se realiza una acción (indicada por el operador), se denomina un operando. Una expresión puede contener un operando y un operador (denominado operador unario), como por ejemplo en b++, o bien dos operandos, separados por un operador (denominado operador binario), como por ejemplo en a + b.

Comparación: Comparan los valores de 2 operandos, devolviendo un resultado de true o false (se usan extensivamente en sentencias condicionales y repetitivas).

```
==   !=   === !==   >   >=   <   <=
```

Aritméticos.: Unen dos operandos para producir un único valor que es el resultado de una operación aritmética u otra operación sobre ambos operandos.

```
+   -   *   /   %   ++   - -   +valor   -valor
```

Asignación: Asigna el valor a la derecha de la expresión a la variable que está a la izquierda**.**

```
=   +=   -=   *=   /=   %=
```

Boolean: Realizan operaciones booleanas aritméticas sobre uno o dos operandos booleanos.

```
&&   ||   !
```

Bit a Bit: Realizan operaciones aritméticas o de desplazamiento de columna en las representaciones binarias de dos operandos.

```
&     |     ^   <<     >>     >>>
```

Objeto: Ayudan a los scripts a evaluar la herencia y capacidades de un objeto particular**.** antes de que tengamos que invocar al objeto y sus propiedades o métodos.

```
.     []     ()     delete     in     instanceOf     new     this
```

Misceláneos: Operadores que tienen un comportamiento especial.

```
,     ?:     typeof     void
```

7.1. OPERADORES DE COMPARACIÓN.

Operadores de comparación en JavaScript

OPERADORES DE COMPARACIÓN EN JAVASCRIPT			
Sintaxis	**Nombre**	**Tipos de operandos**	**Resultados**
==	Igualdad.	Todos.	Boolean.
!=	Distinto.	Todos.	Boolean.
===	Igualdad estricta.	Todos.	Boolean.
!==	Desigualdad estricta.	Todos.	Boolean.
>	Mayor que.	Todos.	Boolean.
>=	Mayor o igual que.	Todos.	Boolean.
<	Menor que.	Todos.	Boolean.
<=	Menor o igual que.	Todos.	Boolean.

En valores numéricos, los resultados serían los mismos que obtendríamos con cálculos algebraicos. Por ejemplo:

```
30  ==  30      //  true
30  ==  30.0      //  true
5  !=  8          //  true
9  >  13          //  false
7.29  <=  7.28      //  false
```

También podríamos comparar cadenas a este nivel:

```
"Marta"  ==  "Marta"        //  true
"Marta"  ==  "marta"        //  false
```

Si por ejemplo comparamos un número con su cadena correspondiente:

```
"123" == 123          // true
```

JavaScript cuando realiza esta comparación, convierte la cadena en su número correspondiente y luego realiza la comparación. También dispones de otra opción, que consiste en convertir las funciones paseInt() o parse Float() el operando correspondiente:

```
parseInt("123") == 123 // true
```

Los operadores === y !== comparan tanto el dato como el tipo de dato. El operador === sólo devolverá true, cuando los dos operandos son del mismo tipo de datos (por ejemplo ambos son números) y tienen el mismo valor.

7.2. OPERADORES ARITMÉTICOS.

Operadores aritméticos en JavaScript

OPERADORES ARITMÉTICOS EN JAVASCRIPT			
Sintaxis	**Nombre**	**Tipos de Operando**	**Resultados**
+	Más.	integer, float, string.	integer, float, string.
-	Menos.	integer, float.	integer, float.
*	Multiplicación.	integer, float.	integer, float.
/	División.	integer, float.	integer, float.
%	Módulo.	integer, float.	integer, float.
++	Incremento.	integer, float.	integer, float.
--	Decremento.	integer, float.	integer, float.
+valor	Positivo.	integer, float, string.	integer, float.
-valor	Negativo.	integer, float, string.	integer, float.
**	Potencia		

Veamos algunos ejemplos:

```
let a = 10; // Inicializamos a al valor 10
let z = 0; // Inicializamos z al valor 0
z = a; // a es igual a 10, por lo tanto z es
igual a 10.
z = ++a; // El valor de a se incrementa justo antes de
ser asignado a z, por lo que a es 11 y
z valdrá 11.
z = a++; // Se asigna el valor de a (11) a z y luego
se incrementa el valor de


//a (pasa a ser 12).
z = a++; // a vale 12 antes de la asignación, por lo que
z es igual a 12; una vez hecha la asignación a valdrá 13.
```

Otros ejemplos:

```
let x = 2;
let y = 8;
let z = -x; // z es igual a -2, pero x sigue siendo
igual a 2.
z = - (x + y); // z es igual a -10, x es igual a 2 e y es
igual a 8.
z = -x + y; // z es igual a 6, pero x sigue siendo
igual a 2 e y igual a
```

7.3. OPERADORES DE ASIGNACIÓN.

Operadores de asignación en JavaScript.

OPERADORES DE ASIGNACIÓN EN JAVASCRIPT			
Sintaxis	**Nombre**	**Ejemplo**	**Significado**
=	Asignación.	x = y	x = y
+=	Sumar un valor.	x += y	x = x + y
-=	Substraer un valor.	x -= y	x = x - y

*=	Multiplicar un valor.	x *= y	x = x * y
/=	Dividir un valor.	x /= y	x = x / y
%=	Módulo de un valor.	x %= y	x = x % y

7.4. OPERADORES BOOLEANOS.

Los operadores booleanos te van a permitir evaluar expresiones, devolviendo como resultado verdadero o falso.

OPERADORES DE BOOLEAN EN JAVASCRIPT			
Sintaxis	Nombre	Operandos	Resultados
&&	**AND.**	**Boolean.**	**Boolean.**
||	**OR.**	**Boolean.**	**Boolean.**
!	**Not.**	**Boolean.**	**Boolean.**

Ejemplos:

```
!true            //  resultado =  false
!(10  >  5)            //  resultado  =  false
!(10  <  5)            //  resultado  =  true
!("gato"  ==  "pato")       //  resultado  =  true
5  >  1  &&  50  >  10       //  resultado  =  true
5  >  1  &&  50  <  10       //  resultado  =  false
5  <  1  &&  50  >  10       //  resultado  =  false
5  <  1  &&  50  <  10       //  resultado  =  false
```

TABLA DE VALORES DE VERDAD DEL OPERADOR AND			
Operando Izquierdo	Operador AND	Operando Derecho	**Resultado**
True	&&	True	True
True	&&	False	False
False	&&	True	False
False	&&	False	False

TABLA DE VALORES DE VERDAD DEL OPERADOR OR			
Operando Izquierdo	Operador OR	Operando Derecho	**Resultado**
True	||	True	True
True	||	False	True
False	||	True	True
False	||	False	False

Ejemplos:

```
5  >  1  ||  50  >  10       //  resultado  =  true
5  >  1  ||  50  <  10       //  resultado  =  true
5  <  1  ||  50  >  10       //  resultado  =  true
5  <  1  ||  50  <  10       //  resultado  =  false
```

7.5. OPERADORES MISCELÁNEOS.

El operador coma ,

Este operador, indica una serie de expresiones que van a ser evaluadas en secuencia, de izquierda a derecha. La mayor parte de las veces, este operador se usa para combinar múltiples declaraciones e inicializaciones de variables en una única línea. Ejemplo:

```
let nombre, direccion, apellidos, edad;
```

Otra situación en la que podemos usar este operador coma es dentro de la expresión loop. En el siguiente ejemplo inicializamos dos variables de tipo contador, y las incrementamos en diferentes porcentajes. Cuando comienza el bucle, ambas variables se inicializan a 0 y a cada paso del bucle una de ellas se incrementa en 1, mientras que la otra se incrementa en 10.

```
for ( let i=0, j=0 ; i < 125; i++, j+10)
{
     // instrucciones
}
```

Nota: no confundir la coma, con el delimitador de parámetros ";" en la instrucción for.

? : (operador condicional)
Este operador condicional es la forma reducida de la expresión if …. else.

La sintaxis formal para este operador condicional es:

```
condicion ? expresión si se cumple la condición: expresión si no
se cumple;
```

Si usamos esta expresión con un operador de asignación:

```
 let = condicion ? expresión si se cumple la condición:
expresión si no se cumple;
```

Ejemplo:

```
let a,b;
a = 3;
b = 5;
let h = a > b ? a : b;  // a h se le asignará el valor 5;
```

typeof (devuelve el tipo de valor de una variable o expresión).
Este operador unario se usa para identificar cuando una variable o
Este operador unario se usa para identificar cuando una variable o expresión es de alguno de los siguientes tipos: `number, string, boolean, object, function o undefined.`

Ejemplo:

console.log(typeof 42); // salida esperada: “number”

console.log(typeof ‘audi’);//salida esperada: “String”

console.log(typeof true); //salida esperada “boolean”

console.log(typeof variable sin declarar); //salida esperada “undefined”

8. ESTRUCTURAS DE CONTROL CONDICIONALES.

8.1. CONSTRUCCIÓN IF

La decisión más simple que podemos tomar en un programa es la de seguir una rama determinada si una determinada condición es true.

Sintaxis:

```
if  (condición) { // entre paréntesis irá la condición que se
evaluará a true o false.
     // instrucciones a ejecutar si se cumple la condición
}
```

Ejemplo:

```
if (miEdad >30){
alert("Ya eres una persona adulta");
}
```

8.2. CONSTRUCCIÓN IF … ELSE

En este tipo de construcción, podemos gestionar que haremos cuando se cumpla y cuando no se cumpla una determinada condición.

Sintaxis:

```
if  (condición) { // entre paréntesis irá la condición que se
evaluará a true o false.
     // instrucciones a ejecutar si se cumple la condición
} else {
     // instrucciones a ejecutar si no se cumple la condición
}
```

Ejemplo:

```
if (miEdad >30) {
     alert("Ya eres una persona adulta.");
} else {
     alert("Eres una persona joven.");
}
```

8.3. CONSTRUCCIÓN SWITCH

Nos permite realizar la evaluación múltiple de un valor.

Sintaxis:

```
switch(expression) {
    case n:
        instrucciones;
        break;
    case n:
       instrucciones;
        break;
    default:
        instrucciones;
}
```

Ejemplo:

```
switch (new Date().getDay()) {
  case 0:
    dia = "Domingo";
    break;
  case 1:
    dia = "Lunes";
    break;
  case 2:
    dia = "Martes";
    break;
  case 3:
    dia = "Miércoles";
    break;
  case 4:
    dia = "Jueves";
    break;
  case 5:
```

```
        dia = "Viernes";
        break;
    case 6:
        dia = "Sábado";
    default:
            dia = "Número de día incorrecto";
}
```

9. ESTRUCTURAS DE CONTROL REPETITIVAS

Los bucles son estructuras repetitivas, que se ejecutarán un número de veces fijado expresamente, se dice que un proceso repetitivo de tipo es un proceso repetitivo por contador.

9.1. BUCLE FOR.

Este tipo de bucle te deja repetir un bloque de instrucciones un número limitado de veces.

Sintaxis:

```
for (expresión inicial; condición; incremento) {
    // Instrucciones a ejecutar dentro del bucle.
}
```

Ejemplo:

```
for ( let i=1; i<=20; i++) {
    // instrucciones que se ejecutarán 20 veces.
}
```

9.1.1. ESTRUCTURA FOR...IN

Una estructura de control derivada de for es la estructura for...in, es un bucle que se asocia con elementos iterables, como arrays, strings…

La sintaxis es:

```
for(indice in iterable) {
  ...
}
```

Si se quieren recorrer todos los elementos que forman un array, la estructura for...in es una forma muy rápida y eficiente de hacerlo:

```
const dias = ["Lunes", "Martes", "Miércoles", "Jueves", "Viernes", "Sábado", "Domingo"];
 for(let i in dias) {
  alert(dias[i]);
}
```

La variable que se indica como índice es la que se puede utilizar dentro del bucle for...in para acceder a los elementos del array. De esta forma, en la primera repetición del bucle la variable i vale 0 y en la última vale 6.

Esta estructura de control es la más adecuada para recorrer arrays (y objetos), ya que evita tener que indicar la inicialización y las condiciones del bucle for simple y funciona correctamente cualquiera

que sea la longitud del array. De hecho, sigue funcionando igual, aunque varíe el número de elementos del array.

9.1.2. ESTRUCTURA FOR...OF

Es un bucle parecido al anterior, la diferencia está en que en lugar de devolvernos el índice del elemento de la estructura iterable que estamos recorriendo, es te caso nos devuelve el elemento.

La sintaxis es la siguiente:

```
for (variable of iterable) {
        …
}
```

Si se quieren recorrer todos los elementos que forman un array, la estructura for...of es una forma muy rápida y eficiente de hacerlo:

```
const dias = ["Lunes", "Martes", "Miércoles", "Jueves", "Viernes", "Sábado", "Domingo"];
        for(let dia of dias) {
         console.log(dia);
        }
```

En este caso en cada iteración del bucle obtenemos el contenido de cada una de las posiciones del array, es decir, Lunes, en la primera pasada, Martes en la segunda, y así sucesivamente hasta recorrer todo el array.

9.2. BUCLE WHILE().

Este tipo de bucles se utilizan cuando queremos repetir la ejecución de unas sentencias un número indefinido de veces, siempre que se cumpla una condición. Este no es un proceso repetitivo por contador, es un proceso repetitivo por condición, es decir, va a evaluar una condición, si la condición se cumple ejecuta el conjunto de instrucciones que tiene y vuelve a evaluar la condición…. Y así sucesivamente hasta que la condición no se cumpla, en ese caso finaliza.

Un bucle while, primero evalua y luego ejecuta, por lo que se va a ejecutar 0 ó n veces.

Sintaxis:

```
        while (condición) {
           // Instrucciones a ejecutar dentro del bucle.
        }
```

Ejemplo:

```
        let i=0; while (i <=10) {
           // Instrucciones a ejecutar mientras se cumpla la condición.
           i++;
        }
```

9.3. BUCLE DO … WHILE().

El tipo de bucle do...while es la última de las estructuras para implementar procesos repetitivos de las que dispone JavaScript.

Igual que el proceso repetitivo while, visto anteriormente, es un proceso repetitivo por condición, pero en este caso primero se ejecuta y después se evalúa, si la condición se cumple vuelve a ejecutar las instrucciones, y así hasta que no se cumpla la condición.

Es prácticamente igual que el bucle while(), con la diferencia, de que sabemos seguro que el bucle por lo menos se ejecutará siempre una vez, por lo tanto lo utilizaremos cuando el proceso repetitivo se va vaya a ejecutar 1 ó N veces.

Sintaxis:

```
do {
   // Instrucciones a ejecutar dentro del bucle.
}while (condición);
Ejemplo:
   let  a = 1; do{
alert("El valor de a es: "+a);// Mostrará esta alerta 2 veces.
 a++;
  }while (a<3);
```