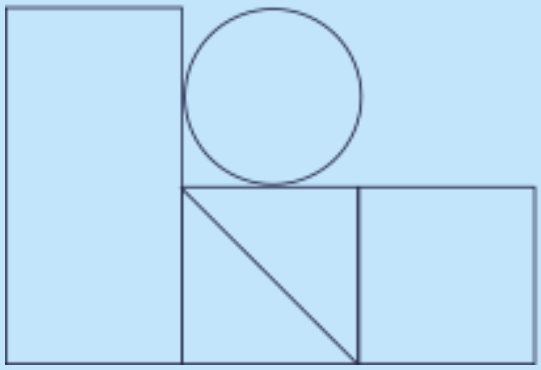
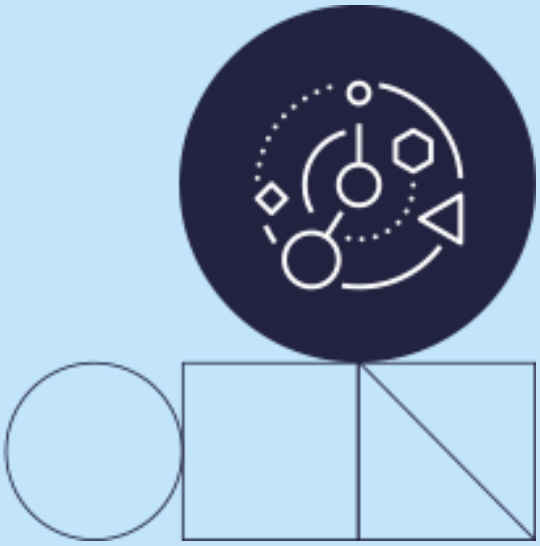
IBM SkillsBuild

IBM

Desarrollo Web

JavaScript para el desarrollo web





Índice

Introducción

Comenzamos en este tema con la tercera pata que compone el estándar **HTML5**, **JavaScript**.

Una página web **HTML5** constará de tres partes bien definidas:

- **El código HTML**: Compuesto por etiquetas **HTML** que conformará el esqueleto de nuestra página WEB.
- **El código CSS**: Con el que daremos un aspecto gráfico a nuestra página WEB.
- **El código JavaScript**: Con el que aportaremos animación e interactividad a nuestra página WEB.

Características de JavaScript

- Se ejecuta en local.
- Es interpretado, no compilado.
- Es de respuesta inmediata.
- Agrega interactividad a los sitios web.
- Proporciona efectos visuales dinámicos.

¿Dónde va JavaScript?

Para "linkar" una página **JavaScript** a un **html**, se puede poner en el **head**, en cualquier etiqueta o "linkar" la página entera (de forma similar a como lo hacíamos con **CSS**);

En el head

Entre etiquetas <script></script>.

```
<head>
  <meta charset="UTF-8">
  <title>Ejemplo de HEAD con JavaScript</title>
  <script>
    alert("Hola mundo");
  </script>
</head>
```

Nota: Si nuestra página va a tener en el head tanto **CSS** como **JavaScript**, lo habitual es poner primero el código **CSS**, ya que es muy frecuente que el **script** haga referencia a elementos **CSS** y deben estar ya cargados en memoria cuando la ejecución llegue a la parte de **JavaScript**.

En una etiqueta

```
<p onClick="alert('Hola mundo');">Página de prueba JavaScript</p>
```

Con el anterior comando hemos sacado un mensaje de alerta que dice "Hola mundo"

Linkar una página JavaScript a nuestro HTML

En este caso, creamos la página **JavaScript** y en el **head** de nuestra página **HTML** ponemos:

```
<head>
  <meta charset="UTF-8" />
  <title>Ejemplo de página con JavaScript externo</title>
  <script src="nombre_de_la_página_JavaScript.js"></script>
</head>
```

De esa forma el contenido del archivo .js quedará anexado a nuestra página web.

Entrada y salida de datos

Para la entrada de datos JavaScript dispone de dos herramientas:

- El comando prompt()
- La entrada de datos por formulario HTML

Prompt()

Con prompt introducimos la variable mediante una ventana.

```
    var nombre=prompt("Introduce nombre "
, "nombre por defecto");
```

En este ejemplo, lo que el usuario introduzca en la ventana que se le abrirá, será guardado en la variable nombre.

A la función **prompt()** puedo pasarle dos parámetros. El primer parámetro es el texto a mostrar y el segundo el valor por defecto.

```
    var edad = prompt("Que edad tienes?:
", 18);
```

Importante: Todo lo que se introduzca por **prompt()** siempre será considerado como **string**, aunque sean números. Si quiero poder operar con dichos números tengo que hacer la conversión con **parseint**, **parsefloat**, etc.

Para la salida de datos disponemos de tres opciones:

- El comando alert()
- El comando console.log()
- El comando document.write()

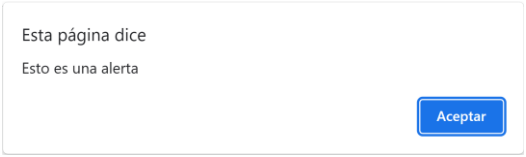
Alert()

El commando alert() nos sacará una ventana emergente con un aviso. Hay dos tipos de alert():

Alerta normal

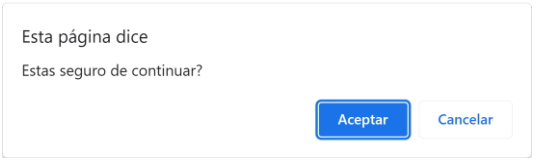
```
    alert("Esto es una alerta");
```

Lo que nos sacará en el navegador lo siguiente:



Alertas de confirmación

```
    var confirmacion = confirm("Estas
seguro de continuar?");
```

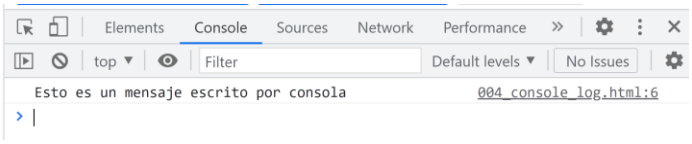


En la variable confirmación almaceno la contestación del usuario que será **true** o **false** dependiendo de si acepta o cancela.

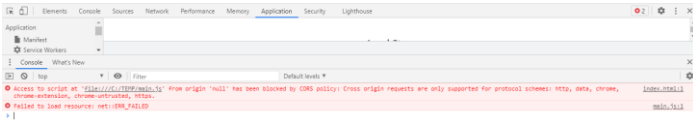
console.log()

Con este comando se nos mostrará un mensaje en la consola del navegador. No será visible a menos que abramos la consola. En la mayoría de los navegadores esto se hace con F12.

```
    console.log("Esto es un mensaje
escrito por consola");
```



La consola nos permite trabajar con JavaScript de manera instantánea, corregir errores, ver que llevan variable dentro ver qué pasa en nuestro código ver por dónde va el flujo del programa etcétera.



Document.write()

Con document.write() podemos escribir directamente en el código **HTML** de nuestra página.

Se devuelve un String si el usuario hace clic en "OK ", se devuelve el valor de entrada. Si el usuario hace clic en "cancelar ", se devuelve NULL. Si el usuario hace clic en aceptar sin introducir ningún texto, se devuelve una cadena vacía.

```
  <head>
    <meta charset="utf-8">
    <title>Documento sin título</title>
    <script>
      var nombre = "Angel";
      var apellido = "García";
      document.write("<h1>Tu nombre
es</h1>" + nombre + " " + apellido);
    </script>
  </head>
```

Ejemplos de entrada y salida de datos

```
    //Salida por ventana emergente
    alert("Esto es una ventana
emergente.");

    //Entrada de datos
    var edad = prompt("Introduzca edad:
");

    // Salida de datos leyendo desde una
variable
    alert("La edad introducida es: " +
edad);

    // Salida de datos por consola
    console.log("Ha terminado el
programa");

    // Salida de datos al BODY de mi
página
    document.write("<h1>Esto es un H1
puesto desde JavaScript</h1>");
```

Estructuras básicas

Sentencias

Llamamos sentencia a cada “orden” que programemos en **JavaScript**. Terminan con “**;**”.

```
    alert("Esto es una sentencia ");
    document.write("Esta es otra ");
```

Funciones predefinidas

Son comandos que ya vienen por defecto en **JavaScript** listos para ser utilizados. Siempre llevan paréntesis.

```
alert();
document.write();
```

Además de estas funciones predefinidas, nosotros podremos crear las nuestras.

Comentarios

En **JavaScript** tenemos dos tipos de comentarios, de una sola línea o de varias:

// Comentario de una sola línea.
/*...*/ Comentario de varias líneas.

```
//      Comentario de una sola línea.

/* Comentario de varias líneas.
Siempre va entre
los signos de asterisco y barra
invertida */
```

Tipos de datos

JavaScript tiene solo tres tipos básicos:

- **Valores numéricos**: Ya sean números enteros o de coma flotante (con decimales).
- **Strings**: Caracteres alfanuméricos, es decir, palabras o frases.
- **Booleans**: Con dos posibles valores True o False.

Modo estricto

En **JavaScript** tenemos un modo que, al activarlo, nos permite ser más restrictivo en nuestro modo de programar. Para ello, al principio de nuestra página pondremos:

'use strict'

Por ejemplo, sin este modo activo podríamos definir una variable con la siguiente sintaxis:

```
num=5;
```

Con el modo estricto activo tendríamos que poner:

```
var num=5;
```

Además, existen determinadas funcionalidades que solo funcionan con el modo estricto activado.

Declaración de variables

Por definición, una variable es un espacio en la memoria del ordenador donde se guardará un valor, que puede cambiar a la largo de la ejecución del programa.

Se pueden declarar las variables de dos formas, con **let** y con **var**:

- **Let**: permite declarar variables limitando su alcance al bloque o declaración donde se está usando. Es decir, las variables declaradas con let solo se pueden usar desde dentro de funciones, condicionales o bucles.
- **Var**: define una variable global o local (en una función) sin importar el ámbito del bloque. Las variables declaradas con var estarán disponibles para todos los elementos del código.

Las variables tienen dos ámbitos posibles:

- **Global**: Podemos acceder a ellas desde cualquier parte de nuestro código.
- **Local**: Podemos acceder a ellas solo dentro del ámbito en que fueron definidas. Normalmente funciones. Serán accesibles desde la propia función o desde funciones anidadas en niveles superiores a la anterior. Es decir, las variables locales se pueden ver desde dentro hacia fuera, pero nunca desde fuera hacia dentro de la función.

```
    var num = 5;
    document.write(num);    //num vale 5

    if (true) {
        var num = 10;
        document.write(num);    //num
vale 10
    }

    document.write(num);    //num vale 10
```

Con **let**:

```
    let num = 5;
    document.write(num);    //num vale 5

    if (true) {
        let num = 10;
        document.write(num);    //num
vale 10
    }

    document.write(num);    //num vale 5
```

let lo que realmente hace es crear una nueva variable local a nivel de bloque, en este caso dentro del **if**. Fuera no tiene validez.

```
    var puntuación;
    puntuación=500;

    var puntuación=500, record=1000,
jugador= "Angel";
```

Si declaramos una variable entre comillas, JavaScript entiende que es un string, aunque sea un número:

```
    var num1=5;     //Aquí num1 es un
número (int).
    var num2= "5";  //Aquí num2 es un
string.
```

Para modificar una variable ya declarada no ponemos var, simplemente el nombre y el valor:

```
    var num=5;  //Declaracion y
asignación
    num=10;     //Reasignación
```

El comando typeof() nos indica el tipo de una variable.

Podemos poner typeof num; o typeof(num);

Casting de variables

Si necesitamos convertir un string a number tenemos dos opciones:

1-Usar la función **Number**():

```
var num1=45;
Resultado= num1+Number(num2);
```

2-Usar **parseInt**:

```
alert(parseInt(num2)+23);
```

De igual forma tenemos disponible la función **parseFloat()** para convertir strings a números decimales.

Constantes

Por definición, una variable es un espacio en la memoria del ordenador donde se guardará un valor, que puede cambiar a la largo de la ejecución del programa. En el caso de una constante, se define como un espacio en la memoria del ordenador donde se guardará un valor, que **NO** puede cambiar a la largo de la ejecución del programa. Es decir, si declaramos una constante y dentro guardamos un valor, dicho valor ya no podrá ser modificado.

Sintaxis:

```
const nombre="Antonio";
```

Ejemplos de operaciones con variables

```
//Podemos trabajar con números, strings y
booleans
// Declaración de variables:
// No puede empezar por un número
// Esto de error:    var 2_cliente;
// Evitar tildes y Ñ

//Variable correctamente declarada
let _numero;

// Inicialización de la variable
_numero = 63;

//Variable declara e inicializada
var numero4 = 647;

// Sepueden declara e inicializar varias
variables a la vez.
var a=3, b=4, c=5;

//Sería lo mismo que poner:
/*
var a=3;
let b=4;
var c=5;
*/

// JavaScript es Case sensitive.
Distingue entre mayúsculas/minúsculas
var numero = 10;
var Numero = 20;

console.log("La variable 'numero' vale:"
+ numero);
console.log("La variable 'Numero' vale:"
+ Numero);

//Con variable numéricas podemos operar
matemáticamente

var sumando1 = 35;
var sumando2 = 45;
```

```
var resultado = sumando1+sumando2;

console.log("El resultado de sumar " +
sumando1 + " + "
            + sumando2 + " es " +
resultado);


 // Operador modulo o resto de la
división
 var resto = 46 % 5;
 console.log("El resto de 46/5 es " +
resto);

 //Operador incremento
 var numeroInicial = 10;
 let numeroIncrementado =
++numeroInicial;

 //Sería como poner que
numeroIncrementado = numeroInicial +1;
 console.log("El operador incremento
sobre numeroInicial daría " +
numeroIncrementado);

 //IMPORTANTE: No es lo mismo ++variable
que variable++
 var numero5 = 5;

 document.write("El número antes del
incremento vale " + numero5++);
 document.write("<br>");
 document.write("El número después del
incremento vale " + numero5);

 document.write("<br>");
 document.write("<br>");

 // Ahora al revés
 let numero6 = 5;

 document.write("El número antes del
incremento vale " + ++numero6);
 document.write("<br>");
 document.write("El número después del
incremento vale " + numero6);

 //Constantes. Como las variables pero no
se puede cambiar su valor
```

```
 // Se declaran en mayúsculas
 const MICONSTANTE = 4765;
 //MICONSTANTE = 345;   Esto daría error

 // Hay que declararlas e inicializarlas
obligatoriamente.
 //const MICONSTANTE2;  Esto daría error
```

Strings

Los strings se definen como variables cuyo contenido son caracteres alfanuméricos, es decir, texto. Siempre van entre comillas.

```
//Declarión de strings
//------------------------
//------------------------
let nombre = "Mi nombre es Angel";
let nombre2 = 'Mi nombre es Angel';

/* Aunque declaremos números, si lo
hacemos entre comillas,
nos lo tratará como strings */
let edad2 = "43";
console.log(edad2+20);

//Las comillas anidadas siempre alternas,
let nombre3 = 'Mi nombre es "Angel" ';
let nombre4 = "Mi nombre es 'Angel' ";

/* Cuando declaremos numeros con los que
no vamos a operar
matemáticamente lo haremos como strings
*/
let telefono = "666666666";

//Los strings no se suman, se concatenan:
let nombre5 = "Angel";
let espacio = " ";
let apellido = "García";
let nombreCompleto =
nombre+espacio+apellido;

console.log(nombreCompleto);

//Otra forma de hacerlo
console.log(nombre5 + " " +apellido);

console.log(nombre5 + " " +edad2);
```

Funciones de los strings

JavaScript dispone de funciones predefinidas que nos facilitan la labor de trabajar con strings. Las más usadas son:

Parseint(): Convierte un STRING a número. Otra opción es usar la función **Number()**.

toString: Convierte un número a **string**:

```
var num=3;
var num2=num.toString();
```

toUppercase: Convierte un string a mayúsculas.

toLowercase: Convierte un String a minúsculas.

Length: Calcula la longitud de un texto. Si la usamos con **arrays** me dice el número de elementos del **array**.

Concat: Para concatenar texto. Como el +

```
var texto=texto1.concat(" "+texto2);
```

indexOF(): Para buscar un texto dentro de otro. Me saca si existe un texto dentro de otro y en qué posición está. Busca la primera coincidencia.

También tenemos un **LastIndexOf**() que nos saca la última coincidencia.

```
var texto="Esto es el texto";
var busqueda= texto.indexOf("texto");
document.write(busqueda);
```

En este caso me imprimiría 11.

El método **search**() funciona igual:

```
var busqueda= texto.search("texto");
```

Si no encuentra el texto me devolverá -1.

Match(): Funciona como **indexOF** o **search**, pero en este caso nos devuelve un array con los resultados. En principio me devuelve la primera coincidencia. Si quiero que me saque todas tengo que usar la expresión regular:

```
var busqueda= texto.match(/texto/gi);
```

Substr(): Sácame desde el carácter 14, 5 caracteres en adelante:

```
var busqueda= texto.substr(14,5);
```

CharAt(): Para seleccionar una letra determinada de un string.

```
var busqueda= texto.CharAt(44);
```

Me mostraría la letra que se encuentre en la posición 44.

StartWith(): Para buscar un texto al inicio del string. Devuelve **true** o **false**, dependiendo de si lo ha encontrado o no.

Es decir, si el texto comienza por los caracteres que le digamos.

```
var texto="Hola mundo";
var busqueda= texto.StarWith("Hola");
```

En este caso daría true.

```
var busqueda=
texto.StarWith("mundo");
```

En este caso daría false.

endsWith(); Como el anterior pero mirando si el texto termina con los caracteres que le digamos.

includes(); Busca una palabra en un texto. Es case sensitive.

replace(); Nos permite reemplazar un texto por otro.

```
var busqueda= texto.replace("Hola",
"Adiós");
```

Me busca “Hola” y me lo reemplaza por “Adiós”.

slice(); Me separa un string a partir del carácter que le digamos.

```
var texto= "Hola mundo";
var busqueda= texto.slice(5);
```

Me corta el string a partir del carácter 5, es decir, me deja solo “mundo”.

También puedo decirle posición de comienzo y fin:

```
    var busqueda= texto.slice(5,7);
```

split(); Me mete el string en un array.

```
    var busqueda= texto.split();
```

Podemos indicarle un separador y nos mete las palabras que hay entre espacios, por ejemplo:

```
    var busqueda= texto.split(" ");
```

trim(): Me quita los espacios por delante y por detrás del string.

Plantillas en strings

Podemos sustituir valores dentro un string sin necesidad de estar concatenándolos, usando una plantilla. Para ello uso las comillas invertidas y la interpolación de variables con $().

Modo normal:

```
    var nombre=prompt("Introduce tu nombre: ");
    var apellidos=prompt("Introduce tus apellidos: ");
    var texto="Tu nombre es "+nombre+" y tus apellidos"+apellidos;
```

Con plantilla:

```
    var texto= `
    <h3>Tu nombre es $(nombre)</h3>
    <h3> y tus apellidos $(apellidos)</h3>
    `;
```

Operadores

Las variables por sí solas son de poca utilidad. Hasta ahora, solo se ha visto cómo crear variables de diferentes tipos y cómo mostrar su valor mediante la función `alert()`. Para hacer programas realmente útiles, son necesarias otro tipo de herramientas.

Los operadores permiten manipular el valor de las variables, realizar operaciones matemáticas con sus valores y comparar diferentes variables. De esta forma, los operadores permiten a los programas realizar cálculos complejos y tomar decisiones lógicas en función de comparaciones y otros tipos de condiciones.

Operadores de Asignación

El operador de asignación es el más utilizado y el más sencillo. Este operador se utiliza para guardar un valor específico en una variable. El símbolo utilizado es `=` (no confundir con el operador `==` que se verá más adelante):

```
    var numero1 = 3;
```

A la izquierda del operador, siempre debe indicarse el nombre de una variable. A la derecha del operador, se pueden indicar variables, valores, condiciones lógicas, etc.:

```
    var numero1 = 3;
    var numero2 = 4;

    /* Error, la asignación siempre se
realiza a una variable,
        por lo que en la izquierda no se
puede indicar un número */
    5 = numero1;

    // Ahora, la variable numero1 vale 5
    numero1 = 5;

    // Ahora, la variable numero1 vale 4
    numero1 = numero2;
```

Operadores Incremento y decremento

Estos dos operadores solamente son válidos para las variables numéricas y se utilizan para incrementar o decrementar en una unidad el valor de una variable.

```
    var numero = 5;
    ++numero;
    alert(numero);  // numero = 6
```

El operador de incremento se indica mediante el prefijo `++` en el nombre de la variable.

El resultado es que el valor de esa variable se incrementa en una unidad.

Por tanto, el anterior ejemplo es equivalente a:

```
    var numero = 5;
    numero = numero + 1;
    alert(numero);  // numero = 6
```

De forma equivalente, el operador decremento (indicado como un prefijo `--` en el nombre de la variable) se utiliza para decrementar el valor de la variable:

```
    var numero = 5;
    --numero;
    alert(numero);  // numero = 4
```

El anterior ejemplo es equivalente a:

```
    var numero = 5;
    numero = numero - 1;
    alert(numero);  // numero = 4
```

Los operadores de incremento y decremento no solamente se pueden indicar como prefijo del nombre de la variable, sino que también es posible utilizarlos como sufijo. En este caso, su comportamiento es similar pero muy diferente. En el siguiente ejemplo:

```
    var numero = 5;
    numero++;
    alert(numero);  // numero = 6
```

El resultado de ejecutar el script anterior es el mismo que cuando se utiliza el operador **++numero**, por lo que puede parecer que es equivalente indicar el operador **++** delante o detrás del identificador de la variable. Sin embargo, el siguiente ejemplo muestra sus diferencias:

```
var numero1 = 5;
var numero2 = 2;
numero3 = numero1++ + numero2;
// numero3 = 7, numero1 = 6

var numero1 = 5;
var numero2 = 2;
numero3 = ++numero1 + numero2;
// numero3 = 8, numero1 = 6
```

Si el operador **++** se indica como prefijo del identificador de la variable, su valor se incrementa **antes** de realizar cualquier otra operación. Si el operador **++** se indica como sufijo del identificador de la variable, su valor se incrementa **después** de ejecutar la sentencia en la que aparece.

Por tanto, en la instrucción `numero3 = numero1++ + numero2;`, el valor de `numero1` se incrementa después de realizar la operación (primero se suma y `numero3` vale `7`, después se incrementa el valor de `numero1` y vale `6`). Sin embargo, en la instrucción `numero3 = ++numero1 + numero2;`, en primer lugar se incrementa el valor de `numero1` y después se realiza la suma (primero se incrementa `numero1` y vale `6`, después se realiza la suma y `numero3` vale `8`).

Es decir, aunque ambas expresiones se parecen mucho, no es lo mismo el operador suma e incremento (numero++) que incremento y suma (++numero).

Operadores Lógicos

Los operadores lógicos son imprescindibles para realizar aplicaciones complejas, ya que se utilizan para tomar decisiones sobre las instrucciones que debería ejecutar el programa en función de ciertas condiciones.

El resultado de cualquier operación que utilice operadores lógicos siempre es un valor lógico o booleano.

Negación

Uno de los operadores lógicos más utilizados es el de la negación. Se utiliza para obtener el valor contrario al valor de la variable:

```
    var visible = true;
    alert(!visible);  // Muestra "false"
y no "true"
```

La negación lógica se obtiene prefijando el símbolo `!` al identificador de la variable. El funcionamiento de este operador se resume en la siguiente tabla:

variable	!variable
true	false
false	true

Si la variable original es de tipo booleano, es muy sencillo obtener su negación. Sin embargo, ¿qué sucede cuando la variable es un número o una cadena de texto?

Para obtener la negación en este tipo de variables, se realiza en primer lugar su conversión a un valor booleano:

- Si la variable contiene un número, se transforma en `false` si vale `0` y en `true` para cualquier otro número (positivo o negativo, decimal o entero).
- Si la variable contiene una cadena de texto, se transforma en `false` si la cadena es vacía (`""`) y en `true` en cualquier otro caso.

```
    var cantidad = 0;
    vacio = !cantidad;  // vacio = true

    cantidad = 2;
    vacio = !cantidad;  // vacio = false

    var mensaje = "";
    mensajeVacio = !mensaje;  //
mensajeVacio = true

    mensaje = "Bienvenido";
    mensajeVacio = !mensaje;  //
mensajeVacio = false
```

AND

La operación lógica **AND** obtiene su resultado combinando dos valores booleanos. El operador se indica mediante el símbolo **&&** y su resultado solamente es **true** si los dos operandos son true:

variable1	variable2	variable1 && variable2
true	true	true
true	false	false
false	true	false
false	false	false

```
    var valor1 = true;
    var valor2 = false;
    resultado = valor1 && valor2; //
resultado = false

    valor1 = true;
    valor2 = true;
    resultado = valor1 && valor2; //
resultado = true
```

OR

La operación lógica **OR** también combina dos valores booleanos. El operador se indica mediante el símbolo **||** y su resultado es true si alguno de los dos operandos es true:

variable1	variable2	variable1 || variable2
true	true	true
true	false	true
false	true	true
false	false	false

```
    var valor1 = true;
    var valor2 = false;
    resultado = valor1 || valor2; //
resultado = true

    valor1 = false;
    valor2 = false;
    resultado = valor1 || valor2; //
resultado = false
```

Operadores Matemáticos

JavaScript permite realizar manipulaciones matemáticas sobre el valor de las variables numéricas. Los operadores definidos son: suma (**+**), resta (**-**), multiplicación (*****) y división (**/**). Ejemplo:

```
    var numero1 = 10;
    var numero2 = 5;

    resultado = numero1 / numero2;  //
resultado = 2
    resultado = 3 + numero1;        //
resultado = 13
    resultado = numero2 - 4;        //
resultado = 1
    resultado = numero1 * numero2; //
resultado = 50
```

Además de los cuatro operadores básicos, JavaScript define otro operador matemático que no es sencillo de entender cuando se estudia por primera vez, pero que es muy útil en algunas ocasiones.

Se trata del operador "módulo" (o resto de la división), que calcula el resto de la división entera de dos números. Si se divide por ejemplo 10 y 5, la división es exacta y da un resultado de 2. El resto de esa división es 0, por lo que módulo de 10 y 5 es igual a 0.

Sin embargo, si se divide 9 y 5, la división no es exacta, el resultado es 1 y el resto 4, por lo que módulo de 9 y 5 es igual a 4.

El operador módulo en JavaScript se indica mediante el símbolo **%**, que no debe confundirse con el cálculo del porcentaje:

```
    var numero1 = 10;
    var numero2 = 5;
    resultado = numero1 % numero2; //
resultado = 0

    numero1 = 9;
    numero2 = 5;
    resultado = numero1 % numero2; //
resultado = 4
```

Los operadores matemáticos también se pueden combinar con el operador de asignación para abreviar su notación:

```
    var numero1 = 5;
    numero1 += 3;  // numero1 = numero1 +
3 = 8
    numero1 -= 1;  // numero1 = numero1 -
1 = 4
    numero1 *= 2;   // numero1 = numero1
* 2 = 10
    numero1 /= 5;   // numero1 = numero1
/ 5 = 1
    numero1 %= 4;   // numero1 = numero1
% 4 = 1
```

Operadores Relacionales

Los operadores relacionales definidos por JavaScript son idénticos a los que definen las matemáticas: mayor que (**>**), menor que (**<**), mayor o igual (**>=**), menor o igual (**<=**), igual que (**==**) y distinto de (**!=**).

Los operadores que relacionan variables son imprescindibles para realizar cualquier aplicación compleja, como se verá en el siguiente capítulo de programación avanzada. El resultado de todos estos operadores siempre es un valor booleano:

```
    var numero1 = 3;
    var numero2 = 5;
    resultado = numero1 > numero2; //
resultado = false
    resultado = numero1 < numero2; //
resultado = true

    numero1 = 5;
    numero2 = 5;
    resultado = numero1 >= numero2; //
resultado = true
    resultado = numero1 <= numero2; //
resultado = true
    resultado = numero1 == numero2; //
resultado = true
    resultado = numero1 != numero2; //
resultado = false
```

Se debe tener especial cuidado con el operador de igualdad (**==**), ya que es el origen de la mayoría de errores de programación, incluso para los usuarios que ya tienen cierta experiencia desarrollando scripts. El operador **==** se utiliza para comparar el valor de dos variables, por lo que es muy diferente del operador **=**, que se utiliza para asignar un valor a una variable:

```
    // El operador "=" asigna valores
    var numero1 = 5;
    resultado = numero1 = 3;  // numero1
= 3 y resultado = 3

    // El operador "==" compara variables
    var numero1 = 5;
    resultado = numero1 == 3; // numero1
= 5 y resultado = false
```

Los operadores relacionales también se pueden utilizar con variables de tipo cadena de texto:

```
    var texto1 = "hola";
    var texto2 = "hola";
    var texto3 = "adios";

    resultado = texto1 == texto3; //
resultado = false
    resultado = texto1 != texto2; //
resultado = false
    resultado = texto3 >= texto2; //
resultado = false
```

Cuando se utilizan cadenas de texto, los operadores "mayor que" (**>**) y "menor que" (**<**) siguen un razonamiento no intuitivo: se compara letra a letra comenzando desde la izquierda hasta que se encuentre una diferencia entre las dos cadenas de texto. Para determinar si una letra es mayor o menor que otra, las mayúsculas se consideran menores que las minúsculas y las primeras letras del alfabeto son menores que las últimas (`a` es menor que `b`, `b` es menor que `c`, `A` es menor que `a`, etc.)

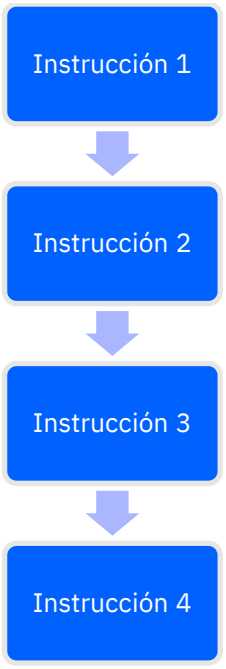
Condiciones y bucles en JavaScript

En los lenguajes de programación, las instrucciones que te permiten controlar las decisiones y bucles de ejecución se denominan "Estructuras de Control". Una estructura de control dirige el flujo de ejecución a través de una secuencia de instrucciones, basadas en decisiones simples y en otros factores.

Una parte muy importante de una estructura de control es la “condición”. Cada condición es una expresión que se evalúa a true o false.

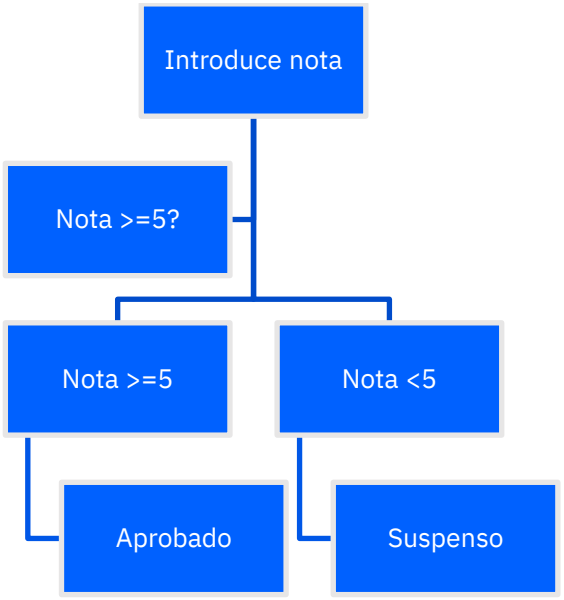
JavaScript ofrece un total de cuatro instrucciones para procesar código de acuerdo a condiciones determinadas por el programador: if, switch, for y while.

Hasta ahora nosotros hemos realizado script de ejecución lineal. Es decir, ejecutando instrucciones una detrás de otra. Hasta que no terminaba una instrucción, no se ejecutaba la siguiente:

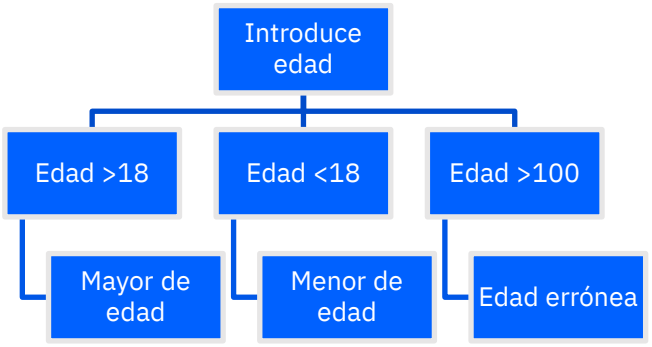


Con los condicionales vamos a modificar el flujo de ejecución según nos convenga en función de si se cumple una condición.

En el siguiente diagrama se representa el flujo de ejecución de un condicional simple. Se evalúa la variable nota y si esta es igual o mayor que 5 se ejecutarán una serie de instrucciones, de lo contrario se ejecutarán otras, pero nunca ambas, o unas u otras.



Según las necesidades de nuestro código, el condicional se puede complicar lo que sea necesario:



Condicionales

La toma de decisiones en programación es similar a la toma de decisiones en la vida real. En programación también enfrentamos algunas situaciones en las que queremos que se ejecute un cierto bloque de código cuando se cumple alguna condición.

Los lenguajes de programación utilizan instrucciones condicionales para controlar el flujo de ejecución del programa en función de ciertas condiciones. Estos se utilizan para hacer que el flujo de ejecución avance y se ramifique en función de los cambios en el estado de un programa.

Comparadores

Para poder tomar decisiones se han de hacer comparaciones.

Dichas comparaciones se llevan a cabo mediante los operadores de comparación y los operadores lógicos:

OPERADOR DE COMPARACIÓN	DESCRIPCIÓN	EJEMPLO
==	Igual que... Comprueba valor.	x==y
===	Estrictamente igual. Comprueba valor y tipo.	x===y
!=	Diferente. Es igual que poner <>	x!=y x<>y
<	Menor que...	x<y
>	Mayor que...	x>y
<=	Menor o igual que...	x<=y
>=	Mayor o igual que...	x>=y

OPERADOR LÓGICO	DESCRIPCIÓN	EJEMPLO
&&	Y lógico	a>b && b<c
||	O lógico	a=b || a=c
!	NO lógico	x!=y

JavaScript trabaja con dos tipos de condicionales, IF...ELSE y SWITCH...CASE. A continuación, veremos el uso de ambos.

IF...ELSE

Esta declaración es la condición más simple para tomar decisiones. Se utiliza para decidir si una determinada declaración o bloque de instrucciones se ejecutará o no, es decir, si una determinada condición es verdadera, entonces un bloque de instrucción se ejecuta de otro modo se ejecutarán otras instrucciones de código.

La sintaxis puede tener varias formas:

Opción 1. Una sola condición a comprobar:

If(Condición a cumplir){
Instrucciones a ejecutar en caso de que la condición se cumpla
}**else**{
Instrucciones a ejecutar en caso de que la condición no se cumpla
}

Por ejemplo, un coche cuesta 30000€. Preguntar al usuario cuánto dinero tiene. Comparar ambas cantidades y sacar un mensaje de alerta diciendo si puede o no comprar el coche.

```
    var precio=30000;
    var dinero=prompt("Introduce cuanto dinero tienes: ");

    if(dinero>precio){
        alert("Te puedes comprar el coche");
    }else{
        alert("Te vas en autobus");
    }
```

En este ejemplo hemos evaluado como condición que el dinero que se tiene ha de ser igual o mayor al precio del coche para poder comprarlo.

Opción 2. Más de una condición a comprobar

If(Condición a cumplir){
Instrucciones a ejecutar en caso de que la condición se cumpla
}**else IF**(Segunda condición a cumplir){
Instrucciones a ejecutar en caso de que la segunda condición se cumpla
}**else**{
Instrucciones a ejecutar en caso de que ninguna condición se cumpla
}

Se pueden poner tantos else...if como queramos.

El else final no es obligatorio pero es útil para ejecutar instrucciones en caso de que ninguna de las condiciones anteriores se cumpla.

Vamos a añadir al ejemplo anterior una segunda condición. Además de tener el dinero, para comprar el coche se ha de ser mayor de edad:

```
    var precio = 30000;
    var dinero = prompt("Introduce cuánto dinero tienes: ");
    var edad = prompt("Introduce tu edad: ");
    if ((precio < dinero) && (edad >= 18)) {
        alert("Te puedes comprar el coche");
    } else if ((precio < dinero) && (edad < 18)) {
        alert("Tienes el dinero pero no la edad");
    }
    else if ((precio > dinero) && (edad >= 18)) {
        alert("Tienes la edad pero no el dinero");
    }
    else if ((precio > dinero) && (edad < 18)) {
        alert("Ni dinero ni edad");
    }
```

Si el usuario me introdujese por ventana un valor que no es un número tendríamos un problema ya que recordemos que todo lo que el usuario introduzca por prompt se considera un string y, para hacer la comprobación de si edad>=18 necesitamos que edad sea un número.

Para averiguar si un valor es numérico usamos la función isNaN().

```
var num1=prompt("Introduce numero");
var num2=prompt("Introduce numero2");
if(!isNaN(num1)&&!isNaN(num2)){
alert(parseInt(num1)+parseInt(num2));
}else{
alert("No has introducido números");
}
```

SWITCH...CASE

Otra opción que nos brinda JavaScript para usar condicionales es switch...case.

Cualquier evaluación de se pueda hacer con switch...case también se puede hacer con IF... else.

La sintaxis del condicional switch...case es:

switch(Variable_a_comprobar) {

case "opcion1" :

instrucciones;

break;

case "opcion2" :

instrucciones;

break;

.....

case "opcionN" :

instrucciones;

break;

default :

instrucciones;}

Analizamos un poco más en detalle la sintaxis de este código que incluye las palabras claves: **switch**, **case**, **break** y **default**:

- Al lado de la palabra clave '**switch**' pondremos entre paréntesis el nombre de la variable cuyo valor queremos comprobar. Según sea dicho valor se debe de ejecutar una serie de instrucciones.
- Cada valor contra el que vamos a comparar se incluye tras la palabra clave '**case**' separada por dos puntos (:) y el valor encerrado en comillas Por ejemplo, si nuestro programa solicitara un número del 1 al 10, y según eligiera el usuario el programa le diera un resultado concreto, escribiríamos un 'case' para cada nº: 1, 2...10
- Incluimos un '**break**' en cada cláusula 'case' para que el programa no se pare ahí, sino que salga de ese punto y continúe después del final de la instrucción switch con las instrucciones siguientes que contenga el programa.
- La última clausula '**default**' realiza el mismo cometido que el último 'else' que ponemos tras una serie de 'else if'. Es decir, si no se ha cumplido ninguna de las condiciones y queremos que se ejecute algo concreto, lo ponemos en esa cláusula 'default'.

En resumen, la instrucción 'switch' actúa como un conmutador en conjunto con los diferentes 'case' (la traducción de la palabra 'switch' es precisamente esa: interruptor, conmutador). Imagina una vía de tren que en un punto se desdobla en múltiples vías que van en diferentes direcciones. Igual que se accionaba un mecanismo para que el tren siguiera por una de esas vías en función de su destino, los switch...case consiguen ese efecto, haciendo que el programa se encamine por el case que cumple la condición, y por tanto, que el programa ejecute las instrucciones que se han definido allí.

Partiendo de una variable edad, crear un programa que imprima diferentes mensajes según sea el valor de edad.

```
    var edad = 18;
    var imprime = "";

    switch (edad) {
        case 18:
            imprime = "Acabas de cumplir
la mayoría de edad";
            break;
        case 25:
            imprime = "Eres un adulto";
            break;
        case 50:
            imprime = "Eres maduro";
            break;
        default:
            imprime = "Otra edad no
contemplada"

            break;
            document.write(imprime);
    }
```

Es importante destacar que switch...case evalúa condiciones concretas de igualdad. Es decir, es nuestro ejemplo hemos comprobado si edad = 18 o si edad = 25 o edad = 50.

Pero no podríamos hacer comprobaciones del tipo edad >= 18, por ejemplo. Para este tipo de comprobaciones hay que usar un if...else.

Bucles

Los bucles, son una estructura de programación que nos permite ejecutar instrucciones de forma repetitiva dentro de un bloque de programación.

Además, los bucles se ejecutan a través de una condición.

Existen dos tipos de bucles:

- **Bucles determinados**: Aquellos en los que se sabe de antemano cuántas veces se va a repetir un código. Un bucle de este tipo es “For”.
- **Bucles indeterminados**: Aquellos en los que no se sabe a priori cuántas veces se va a repetir el código. Dependerá de que se cumpla o no una condición. Dos bucles de este tipo son “While” y “Do while”.

El bucle determinado FOR

Es un tipo de bucle cuya ejecución dura un número determinado de veces o hasta que su condición se evalué como falsa (false).

Su sintaxis es:

for(inicio ; condición ; incremento/decremento){

código a repetir}

La estructura se describe de la siguiente manera:

- inicio**:** Esta expresión indica la condición para el inicio del ciclo. Usualmente es la declaración de una variable numérica. Se suele crear una variable de nombre i, j, k, etc.
- condición**:** Esta expresión es la condición que se debe mantener para que siga ejecutando el ciclo.
- incremento/decremento**:** Esta expresión cambia el valor de la variable numérica indicada en inicio.
- código a repetir **:** Cuerpo del Bucle. Es el conjunto de instrucciones que se ejecutan durante cada iteración del bucle (en cada vuelta).

```
    for(var i=0;i<10;i++){
        document.write("Hola"+"<br>");
        }

    document.write("Ejecución
terminada.");
```

En este bucle for hemos creado una variable local llamada **i** que hemos inicializado a cero. La condición es que el bucle se repita mientras que **i<10**. A cada vuelta de bucle **i** se incrementará en una unidad (con lo que en la décima vuelta la condición dejará de cumplirse). Y en cada vuelta de bucle se imprimirá la palabra hola.

Después de la décima vuelta la condición dejará de cumplirse ya que **i** pasará a valer 10 y nuestro programa se saldrá del bucle y continuará con la siguiente línea de ejecución, en este caso, imprimir **`Ejecución terminada`**.

Hola
Hola
Hola
Hola
Hola
Hola
Hola
Hola
Hola
Hola
Ejecución terminada.

Otro ejemplo. Crear un bucle for que se ejecute 10 veces y que imprima en pantalla lo que vale la variable local en cada una de las vueltas.

```
  for (i=0 ; i<10 ; i++){
    document.write("En esta vuelta de
bucle I vale " + i);
    document.write("<br>");
  }

  document.write("Ejecución terminada.");
```

En esta vuelta de bucle I vale 0
En esta vuelta de bucle I vale 1
En esta vuelta de bucle I vale 2
En esta vuelta de bucle I vale 3
En esta vuelta de bucle I vale 4
En esta vuelta de bucle I vale 5
En esta vuelta de bucle I vale 6
En esta vuelta de bucle I vale 7
En esta vuelta de bucle I vale 8
En esta vuelta de bucle I vale 9
Ejecución terminada.

El principal uso del bucle for es recorrer arrays, poniendo como condición que el bucle se repetirá tantas veces como elementos tengamos en nuestro array:

```
  var meses = ["Enero", "Febrero",
"Marzo", "Abril", "Mayo", "Junio",
    "Julio", "Agosto", "Septiembre",
"Octubre", "Noviembre", "Diciembre"];
  for (var i = 0; i < meses.length; i++)
{
    document.write(meses[i] + "<br/>");
  }
```

Enero
Febrero
Marzo
Abril
Mayo
Junio
Julio
Agosto
Septiembre
Octubre
Novienbre
Diciembre

Veamos un ejemplo de while.

Repetir el ejemplo de los meses del año que vimos en el for, pero implementándolo con un bucle while:

```
    var contador = 0;
    var meses = ["Enero", "Febrero",
"Marzo", "Abril", "Mayo", "Junio",
        "Julio", "Agosto", "Septiembre",
"Octubre", "Novienbre", "Diciembre"];
    while (contador < meses.length) {
        document.write(meses[contador] +
"<br>");
        contador++;
    }
```

WHILE. DO...WHILE

Es similar al ciclo for explicado anteriormente. Se ejecuta hasta que la condición sea falsa (false). La diferencia se basa en que la evaluación se realiza al final de cada iteración.

Si usamos while podemos encontrarnos con que la condición no se cumple ni en la primera comprobación, con lo que el código no se ejecutaría nunca.

En el caso de do...while nos aseguramos de que el código se ejecuta al menos una vez.

Sintaxis:

While(Condición a cumplir){

Código a ejecutar mientras se cumpla la condición}

do{ Código a ejecutar mientras se cumpla la condición }

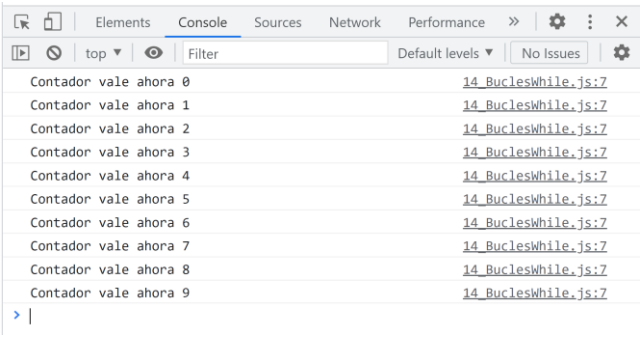
While(Condición a cumplir)

Enero
Febrero
Marzo
Abril
Mayo
Junio
Julio
Agosto
Septiembre
Octubre
Novienbre
Diciembre

Otro ejemplo sería crear un programa que pida al usuario el número de ejecuciones e imprima en consola cuánto vale una variable contador que se incremente en cada ciclo que va haciendo el bucle:

```
    let contador = 0;
    let ciclos = Number(prompt("Introduce número de ejecuciones"));

    while(contador < ciclos){
        console.log("Contador vale ahora " + contador);
        contador++;
    }
```



Un ejemplo de do...while es crear un número aleatorio entre 0-100 y pedir al usuario que intente adivinarlo.

Cada intento incrementará un contador.

```
    //Creamos un número aleatorio entre 0 y 1.
    // Despues lo multiplicamos por 100 para que esté entre 0-100
    //Lo redondeamos para que sea entero
    var numero = parseInt(Math.random()*100);

    var numero_introducido;
    var contador = 0;


    while (numero != numero_introducido)
{
      numero_introducido = Number(prompt("Introduce número: "));
      contador++;

      if (numero_introducido > numero) {
        alert("Demasiado alto");
      }

      if (numero_introducido < numero) {
        alert("Demasiado bajo");
      }
    }


    alert("CORRECTO¡¡¡, el número era el " + numero + ". Has acertado en " + contador + " intentos.");
```

Break

Nos permite salir de un bucle de ejecución al cumplirse una condición.

En el siguiente ejemplo tenemos un bucle que imprimiría los años del 2000 al 2010. Pero hemos introducido un break cuando el año = 2005:

```
var year = 2000;
while (year < 2010) {
  document.write(year + "<br>");
  if (year == 2005) {
    break;
  }
  year++;
}
```

2000
2001
2002
2003
2004
2005

En este caso cuando el año sea igual a 2005, se sale del bucle y no ejecuta el resto de instrucciones del mismo.

Arrays

Hasta ahora hemos visto cómo dentro de una variable metíamos un único valor, ya sea numérico, de texto o boleano.

Los arrays nos permiten trabajar con conjuntos de valores y almacenarlos en una única dirección de memoria.

En **JavaScript** los valores del **array** NO tienen que ser del mismo tipo.

- var Articulos=[“zapatilla”,”camiseta”,”pantalon”,”calcetines”];
- var Articulos=new array [“zapatilla”,”camiseta”,”pantalon”,”calcetines”];

Artículos: zapatilla, camiseta, pantalón y calcetines.

En **JavaScript** un **array** se declara de las siguientes formas:

```
    var Articulos=["zapatilla","camiseta","pantalón","calcetines"];
```

O:

```
    var Articulos =new array("zapatilla","camiseta","pantalón","calcetines");
```

En ambos ejemplos hemos creado un array (es decir, una colección de elementos) al que hemos llamado Artículos y dentro hemos almacenado los valores: zapatilla, camiseta, pantalón y calcetines.

Para poder trabajar con los datos que contiene un array hay que referirnos a ellos según su posición en el array.

Este es un ejemplo de acceso a un array:

```
    alert(Articulos[1]);        Resultado: camiseta
```

La posición de los elementos empieza a contar desde cero, es decir, en nuestro array Artículos, el elemento en la posición cero es zapatilla, en la posición 1

tenemos camiseta, en la posición 2 tenemos pantalón y en la posición 3 tenemos calcetines.

Dado que en nuestro ejemplo hemos pedido que se nos saque por pantalla un mensaje de alerta con el elemento en la posición 1, nos sacará camiseta.

Si queremos cambiar o ingresar un elemento en un array, de la misma forma lo haremos refiriéndonos a su posición dentro del array:

En el siguiente ejemplo vamos a crear un array de tres elementos y vamos a pedir al usuario que introduzca un valor mediante un comando prompt. Dicho valor se almacenará en la posición 4.

```
    var
array1=["objeto1","objeto2","objeto3"];
    array1[3]="objeto4";
    array1[4]=prompt("Introduce
objeto4");
    alert(array1[4]);
```

Funciones de los arrays

JavaScript nos brinda una serie de funciones predefinidas listas para que nos faciliten el trabajo con arrays:

- **Length**: Devuelve la longitud del array.
- **push**: Agrega elementos al final del array.
- **unshift**: Agrega elementos al principio del array.
- **pop**: Elimina elementos al final del array.
- **shift**: Elimina elementos al principio del array.

Aquí podemos ver un ejemplo de estas funciones:

```
<html>

<head>
  <meta charset="utf-8">
  <title>Documento sin título</title>
  <script>
    var articulos = ["balon", "botas",
"camiseta", "pantalon"];
  </script>
</head>

<body>
  <script>
    document.write("<p>El primer articulo
es < strong > "+articulos[0]+"</strong
></p > ");
    document.write("<P>La longitud del
array es " + articulos.length + "</p>");
    document.write("<p>El último articulo
es <strong>" + articulos[articulos.length
- 1] + "</strong></p>");
    articulos.unshift("pelota");
    articulos.push("canasta");
    document.write("<p>Añadimos el
artículo <strong>" + articulos[0] +
"</strong></p>");
    document.write("<p>El primer articulo
ahora es <strong>" + articulos[0] +
"</strong></p>");
    document.write("<p>El último articulo
ahora es <strong>" +
articulos[articulos.length - 1] +
"</strong></p>");
    document.write("<P>La longitud del
array ahora es " + articulos.length +
"</p>");
    document.write(articulos);
    articulos.push(prompt("introduce
nuevo articulo"));
    document.write("<br>");
    document.write(articulos);

  </script>
</body>
</html>
```

Podemos poner un elemento como **undefined**, con lo que su valor será no definido (o desconocido);

```
articulos[0]= undefined;
```

Para averiguar la posición de un elemento:

```
var posicion=articulos.indexof("pantalones");
```

Convertir un array a texto:

```
var miString=artículos.join();
```

Me lo convierte a un **string** separado por comas.

Para hacer lo contrario, es decir convertir un **string** en **array** lo haremos con el método **split**(). Dentro de los parámetros pondremos el carácter que hará de separador.

```
var cadena="nombre1,nombre2,nombre3";
```

var miArray=cadena.split(“,”);

Para ordenar un array usaremos el método el método **short**(). Para ordenar por orden inverso el método **reverse**();

```
cadena.short();
cadena.reverse();
```

Esto nos ordena un array alfabéticamente.

Recorrer arrays con for

A veces necesitaremos recorrer todos los elementos de un array buscando uno o unos en concreto.

Para recorrer un array hay que usar el bucle **FOR**:

```
document.write("<ul>");
var nombres = ["Angel", "Sara", "Manolo", "Ana"];
for (var i = 0; i < nombres.length; i++) {
    document.write("<li>" + nombres[i] + "</li>");
};
document.write("</ul>");
```

Ejemplo, rellenar un array y recorrerlo después:

```
var trabajadores = new Array();
var contador = 0;
var persona;

while (persona != "salir") {
  persona = prompt("Introduce nombre");
  trabajadores[contador] = persona;
  contador++;
}
trabajadores.pop();
for (var i = 0; i < trabajadores.length; i++) {
  document.write(trabajadores[i] + "<br>");
}
```

Recorrer arrays con for each

Existe una modificación del bucle FOR creada específicamente para facilitar el trabajo con arrays, Es el bucle foreach.

Utilizamos la función **foreach** que va a recibir el elemento por parámetro.

```
    document.write("<ul>");
    var nombres=["Angel", "Sara",
"Manolo", "Ana"];
    nombres.forEach((elemento)=>{
    document.write("<li>"+elemento+"</li>
");
    });
    document.write("</ul>");
```

Opcionalmente la función **foreach** puede recibir más parámetros.

Un segundo parámetro puede ser el índice.

```
nombres.forEach(elemento, index) => {
    document.write("<li>" + index + " - "
+ elemento + "</li>");}
```

Recorrer arrays con for in

Otra forma de recorrer un array es con el bucle **for in**.

```
    for( let nombre in nombres){
        document.write("<li>"+nombres[nom
bre]+"</li>");
        };
```

En este caso nombre es el índice.

Búsquedas en un array

Para realizar búsquedas en un array podemos usar el método **find**() que tiene dentro una función de carga.

```
    var
busqueda=nombres.find(function(nombre){
        return nombre==nombre1;
        });
```

busqueda almacena el texto que estamos buscando.

Esto se puede simplificar poniéndolo de la siguiente forma:

```
    var busqueda=nombres.find(nombre =>
nombre==nombre1;);
```

De forma similar podemos buscar el índice con el método **findIndex**().

```
    var busqueda=nombres.findIndex(nombre
=> nombre==nombre1);
```

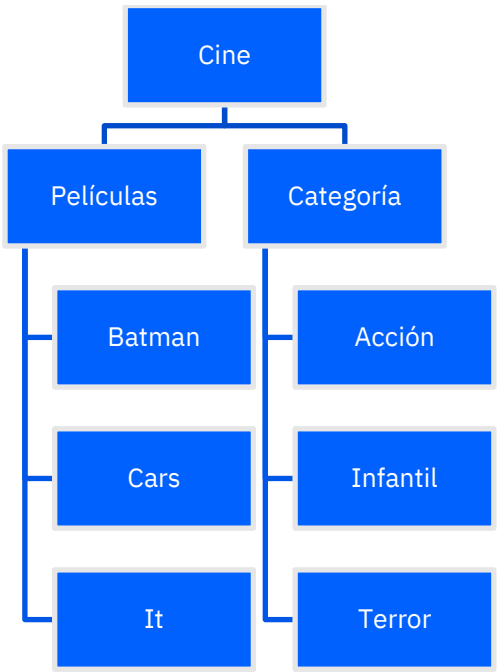
Otro método interesante es **some()** que nos permite comprobar si alguno de los elementos del array cumple una determinada condición. Por ejemplo, comprobar si hay elementos mayores de 40:

```
    var numeros=[3,45,33,65,776];
    var
busqueda=numeros.some(numero=>numero>40);
```

Almacena **true** o **false**.

Arrays multidimensionales

No son más que un array compuesto de otros arrays. Es decir, un array que, en lugar de tener números o strings en su interior, tiene otros arrays. Coloquialmente podríamos decir que es un array de arrays.




```
        var peliculas=['Batman', 'Cars',
'It'];
        var categoria=['Acción',
'infantil', 'Terror'];
        var cine=[ películas, categoria];
        console.log(cine);
```

Para acceder a un elemento, por ejemplo, categoría infantil, deberemos indicar la posición que dicho elemento ocupa en el primer array (en el array contenedor) seguida de la posición que ocupa en el segundo array (el array contenido):

```
console.log(cine[0][1]);
```

En este ejemplo nos imprimiría Cars.

Los arrays multidimensionales no son una herramienta muy usada, pero no está de más que el alumno los conozca.

POO

Ya vimos en anteriores temas los principios de la programación orientada a objetos. Pues bien, JavaScript es un lenguaje de programación orientado a objetos, por lo que comparte los conceptos clave de dicho paradigma de programación. No obstante la POO en JavaScript no se suele usar, ya que la POO está pensada para otro tipo de programas diferentes de las páginas web. Realmente en programación web no tiene mucho sentido el uso de clases, objetos, atributos, métodos, etc. Pero no está de más que el alumno se familiarice con dichos conceptos.

En JavaScript los objetos tienen propiedades y métodos.

Las propiedades se modifican con la nomenclatura del punto y el valor entre comillas:

```
nombreDelObjeto.propiedad= "valor ";
Renault.ancho= "2000 ";
Botton.style.width= "500px ";
Document.write();
Windows.alert();
```

Para llamar a los métodos:
nombreDelObjeto.metodo();

```
Renault.acelera();
```

Ejemplo:

```
  <body>
    <input type="button" id="boton">
    <input type="button" id="boton2">
    <script>
        var
miBoton=document.getElementById("boton");
            miBoton.style.width="300px";
            miBoton.style.height="300px";

        var
miBoton2=document.getElementById("boton2"
);
            miBoton2.style.width="300px";
            miBoton2.style.height="300px"
;
            miBoton2.focus();
    </script>
  </body>
```

Funciones

Las funciones se definen como un conjunto de instrucciones preparadas para ser llamadas en cualquier momento. Solo se ejecutarán si las llamamos. Su finalidad es, pues, la reutilización de código.

Sintaxis:

Para usar una función, hay que definirla primero:

function nombre_funcion(){

//Código a ejecutar

}

Y luego, en otro punto de la página **HTML** llamamos a la función:

nombre_funcion();

Salvo las funciones predefinidas de JavaScript, es indispensable llamar a una función para que se ejecute.

Este es un ejemplo para declarar una función de nombre suma. Dicha función leerá el valor de dos variables y sacará el valor de la suma de ambas.

```
    //Declaración de la función
    let num1 =5;
    let num2 = 10;

    function suma(){
    console.log("La suma de "+ num1 +" y
"+ num2 +" es " + (num1+num2));
    }

    //Llamada a la función
    suma();
```

En esta ocasión, nuestra función ha tomado dos variables que ya estaban instanciadas e incializadas, pero podemos crear funciones que trabajen con valores que deberemos pasarles cuando las llamemos. Son las llamadas funciones con parámetros.

```
    //Declaración de la función


    function suma2(num1, num2){
        console.log("La suma de "+num1+"
y "+num2+" es "+(num1+num2));
    }


    //Llamada a la función
    suma(3, 7);
```

Ya no hemos cogido el valor de num1 y num2 de dos variables preexistentes. Se las hemos pasado por parámetro a nuestra función.

En la declaración de la función hemos puesto que nuestra función, al ser llamada, recibirá dos parámetros, num1 y num2. Por lo tanto, en la llamada a la función estamos obligados a pasarle dichos parámetros o nos dará error.

Al llamar a la función hemos pasado el valor de 3 para num1 y 7 para num2.

Al margen de las funciones que nosotros creemos según nuestras necesidades, existen funciones predefinidas en JavaScript listas para que las usemos, como por ejemplo la función Date() que nos muestra la hora del sistema.

```
<html>
<head>
  <meta charset="utf-8">
  <title>Documento sin título</title>
  <script>
    function fecha() {
      document.write(Date());
    }
  </script>
</head>

<body>
  <h1>Hola es día:
    <script>fecha();</script>
  </h1>
</body>
</html>
```

Escribir una función que pida el nombre al usuario y confeccione un mensaje de bienvenida con dicho nombre:

```
<html>
<head>
  <meta charset="utf-8">
  <title>Documento sin título</title>
  <script>
    var nombre;
    function pedir_nombre() {
      nombre = prompt("Introduce
nombre");
      document.write(nombre);
    }
  </script>
</head>

<body>
  <p>Hola
    <script>pedir_nombre();</script> ,
cómo estás?
  </p>
</body>
</html>
```

Crear una función que sume dos números introducidos por el usuario:

```
<html>
<head>
  <meta charset="utf-8">
  <title>Documento sin título</title>
  <script>
    var num1;
    var num2;
    function suma(num1, num2) {
      num1 = prompt("Introduce número
1: ");
      num2 = prompt("Introduce número 1
:");
      document.write("La suma de " +
num1 + " y " + num2 + " es= " +
(Number(num1) + Number(num2)));
    }
  </script>
</head>

<body>
  <script>
    suma(num1, num2);
  </script>
</body>
</html>
```

Parámetros rest y spread

Cuando no sepamos con exactitud el número de parámetros que le pueden llegar a nuestra función, pondremos tres puntos para hacer referencia al resto de posibles parámetros. Dichos parámetros serán almacenados en un **array** con el nombre de mi parámetro.

```
    function numeros(num1,
num2,...otros){
        document.write(num1 +" , "+
num2+"<br>");
        document.write(otros);
    }

    numeros(2,3,4,54,332);
```

En este ejemplo otros vale 4,54,332. Es un array.

Si no tuviese los tres puntos valdría 4.

Parámetros spread

De forma similar, en el siguiente ejemplo hemos creado un array con dos elementos y se lo hemos pasado a la función como parámetro. Si no usamos los tres puntos, lo que le estamos diciendo es que num1=[1,10] y num2=3 y otros =[4,54,332]. Pero si le ponemos los tres puntos lo que me coge es num1=1, num2=10 y otros=[3,4,54,332]

```
    function numeros(num1,
num2,...otros){
        document.write(num1 +" , "+
num2+"<br>");
        document.write(otros);
    }
    var arrayNumeros=[1,10];
    numeros(…arrayNumeros,3,4,54,332);
```

Funciones anónimas

Las funciones anónimas son aquellas que no tienen nombre y las podemos guardar dentro de una variable.

```
    var miFuncion=function(nombre);
    return "El nombre es " + nombre;
```

Para llamarla:

```
    miFuncion("Angel");
```

Callback

Los Callbacks en JavaScript son como su propio nombre en inglés, indica, ***llamadas de vuelta***, quiere decir que cuando invoco una función pasándole como parámetro otra función (el callback) esta función parámetro se ejecuta cuando la función principal ha terminado de ejecutarse. O cuando a nosotros nos interese...

Este concepto que parece muy complicado es en realidad muy sencillo de entender si lo hacemos con casos prácticos, por ejemplo:

```
    function funcionPrincipal(callback){
        alert('hago algo y llamo al
callback avisando que terminé');
        callback();
    }

    funcionPrincipal(function(){
        alert('termino de hacer algo');
    });
```

Aquí vemos como la función **funcionPrincipal** se ejecuta recibiendo un argumento que es otra función y que se ejecuta después de que termine su labor llamando a callback.

Pero no te preocupes si no lo has entendido, vamos a hacer algunos ejemplos más para ir cogiéndolo, en realidad es muy sencillo.

Siguiendo con el caso anterior, podemos hacer incluso que la **función callback** reciba **argumentos** que se envían desde la función principal...

```
    function funcionPrincipal(callback){
        alert('hago algo y llamo al
callback avisando que terminé');
        callback('Miguel');
    }

    funcionPrincipal(function(nombre){
    alert('me llamo ' + nombre);
    });
```

Con esto ya vemos la fuerza que puede tener esta forma de programación, pero ahora imaginemos que queremos encadenar diferentes funciones con callbacks, lo cual es muy sencillo siguiendo con la misma lógica:

```
    function funcionPrincipal(callback1,
callback2, callback3){
        //código de la función principal
        callback1();
        //más código de la función
principal
        callback2();
        //más código de la función
principal
        callback3();
        //más código si fuera necesario
    }

    funcionPrincipal(
        function(){
            alert('primer callback');
        },
        function(){
            alert('segundo callback');
        },
        function(){
            alert('tercer callback');
        }
    );
```

Aquí vemos como al invocar a la función principal se le pasan como argumentos las tres funciones, que se ejecutan, las cuales podrían recibir parámetros... como hemos visto más arriba, pero de este modo el código queda un poco sucio, ya que no podemos ver claramente las funciones.

De modo que podemos declarar las funciones que se enviarán como argumentos aparte, lo cual nos permite también utilizarlas en otras partes del código, tal y como vemos en el siguiente ejemplo:

```
    function funcionPrincipal(callback1,
callback2, callback3){
        //código de la función principal
        callback1();
        //más código de la función
principal
        callback2();
        //más código de la función
principal
        callback3();
        //más código si fuera necesario
    }

    function callback1(){
            alert('primer callback');
        }

    function callback2(){
            alert('segundo callback');
        }

    function callback3(){
            alert('tercer callback');
        }

    funcionPrincipal(callback1,
callback2, callback3);
```

Podemos observar cómo se declaran todas funciones y luego se pasan como argumentos de la **función principal** para poder utilizarlas dentro de la misma. Pero aunque esto daría para otra entrada, vamos a ver por encima el uso de otra función de JavaScript llamada **setInterval()**, esta función nos sirve para retardar acciones, y necesita dos parámetros para funciones; una función que invoca a la función que vamos a usar y un valor numérico que dice en milisegundos el retraso y actualización de la función invocada.

Esto que parece un lío, vamos a verlo en un código donde mostraremos un alert, un reloj y un texto, y donde el reloj se va modificando cada segundo y el texto tardará en aparecer 3 segundos:

```
  <html>
  <head>
    <meta charset="utf-8">
    <title>Documento sin título</title>
   </head>

  <body>
    <h3 id="demo"></h3>
    <h3 id="demo2"></h3>
    <script>
    function funcionPrincipal(callback1,
callback2, callback3){
        //código de la función principal
        callback1();
        //más código de la función
principal
        var miVar =
setInterval(function(){ callback2() },
1000);
        //más código de la función
principal
        var miVar2 =
setInterval(function(){ callback3() },
3000);
        //más código si fuera necesario
    }

    function callback1(){
            alert('primer callback');
        }

    function callback2(){
            var d = new Date();
            var t =
d.toLocaleTimeString();
            document.getElementById("demo
").innerHTML = t;
        }

    function callback3(){
            document.getElementById("demo
2").innerHTML = 'Esto es el callback3';
        }
```

```
    funcionPrincipal(callback1,
callback2, callback3);
    </script>
  </body>
  </html>
```

El ejemplo en esencia es el mismo, cambiando las funcionalidades de las funciones de callback, pero hemos añadido la función **setInterval()** para que se retrasen e incluso se actualice, aprovechando de paso la forma de hacer un reloj en JavaScript...

Existe una sintaxis para las funciones de callback llamada funciones de flecha, en las que se omite la palabra function y se sustituye por una flecha puesta tras el nombre de la función:

```
    funcionPrincipal(function(nombre){
        alert('me llamo ' + nombre);
    });
```

Quedaría así:

```
    funcionPrincipal(nombre =>{
        alert('me llamo ' + nombre);
    });
```

Funciones flecha

Las funciones de flecha son una manera de definir funciones de callback de una manera mucho más clara y limpia. Simplemente sustituyendo la palabra function por una flecha. Si lleva un parámetro no hace falta poner los paréntesis, si lleva dos ya sí.

```
    function sumame(num1, num2,
sumaYmuestra, sumaPorDos) {
        var suma = num1 + num2;

        sumaYmuestra(suma);
        sumaPorDos(suma);

        return suma;
    }

    sumame(5, 7, dato => {
        console.log('La suma es ', dato);
    },
        dato => {
            console.log('La suma de dos
es ', (dato * 2));

        });
```

Eventos

Los eventos son la manera que tenemos en JavaScript de controlar las acciones de los visitantes y definir un comportamiento de la página cuando se produzcan. Cuando un usuario visita una página web e interactúa con ella se producen los eventos y con JavaScript podemos definir qué queremos que ocurra cuando se produzcan los eventos.

Para entender los eventos necesitamos conocer algunos conceptos básicos:

- **Evento**: Es algo que ocurre. Generalmente los eventos ocurren cuando el usuario interactúa con el documento, pero podrían producirse por situaciones ajenas al usuario, como una imagen que no se puede cargar porque esté indisponible.
- **Tipo de evento**: Es el tipo del suceso que ha ocurrido, por ejemplo, un clic sobre un botón o el envío de un formulario. Cada tipo de elemento de la página ofrece diversos tipos de eventos de JavaScript.
- **Manejador de evento**: es el comportamiento que nosotros podemos asignar como respuesta a un evento. Se especifica mediante una función JavaScript, que se asocia a un tipo de evento en concreto. Una vez asociado el manejador a un tipo de evento sobre un elemento de la página, da vez que ocurre ese tipo de evento sobre ese elemento en concreto, se ejecutará el manejador de evento asociado.

Con JavaScript podemos definir qué es lo que pasa cuando se produce un evento, como podría ser que un usuario pulse sobre un botón, edite un campo de texto o abandone la página.

El manejo de eventos es el caballo de batalla para hacer páginas interactivas, porque con ellos podemos responder a las acciones de los usuarios.

Para definir las acciones que queremos realizar al producirse un evento utilizamos los manejadores de eventos. Existen muchos tipos de eventos sobre los que podemos asociar manejadores de eventos, para muchos tipos de acciones del usuario.

En JavaScript podemos definir eventos de dos maneras distintas. Una manera es en el propio código HTML, usando atributos de los elementos (etiquetas) a los que queremos asociar los manejadores de eventos. Otra manera un poco más avanzada es usando los propios objetos del DOM. Vamos a ver ambas maneras a continuación.

Manejadores de evento especificados en el código HTML

El maneador de eventos se puede colocar en la etiqueta HTML del elemento de la página que queremos que responda a las acciones del usuario. Para ello usamos atributos especiales en las etiquetas del HTML, que tienen el prefijo "on" seguido del tipo de evento. Por ejemplo, el manejador asociado en el atributo "onclick" se ejecuta cuando se produce un clic en una etiqueta.

Vamos a poner un ejemplo sobre el manejador de eventos onclick. Ya sabemos que sirve para describir acciones que queremos que se ejecuten cuando se hace un clic. Por tanto, si queremos que al hacer click sobre un botón se ejecuta alguna función, escribimos el manejador onclick en la etiqueta <INPUT type=button> de ese botón.

Algo parecido a esto:

```
    <INPUT type="button" value="pulsame"
onclick="miFunción()"></INPUT>
```

Se coloca un atributo nuevo en la etiqueta que tiene el mismo nombre que el evento, en este caso onclick. El atributo se iguala a las sentencias JavaScript que queremos que se ejecuten al producirse el evento.

```
    <INPUT type="button" value="pulsar"
onclick="alert('Botón pulsado')">
    </INPUT>
```

Dado el código anterior, al hacer clic sobre el botón aparecerá un mensaje de alerta con el texto "Botón pulsado".

Cada elemento de la página tiene su propia lista de eventos soportados, vamos a ver otro ejemplo de manejo de eventos, esta vez sobre un menú desplegable, en el que definimos un comportamiento cuando cambiamos el valor seleccionado.

```
    <SELECT
onchange="window.alert('Cambiaste la
selección')">
      <OPTION value="opcion1">Opcion 1
      <OPTION value="opcion2">Opcion 2
    </SELECT>
```

Dentro de los manejadores de eventos podemos colocar tantas instrucciones como deseemos, pero siempre separadas por punto y coma.

Lo habitual es colocar una sola instrucción, y si se desean colocar más de una se suele crear una función con todas las instrucciones y dentro del manejador se coloca una sola instrucción que es la llamada a la función.

Vamos a ver cómo se colocarían en un manejador varias instrucciones.

```
  <input type=button value=Pulsar
   onclick="x=30; window.alert(x);
window.document.bgColor = 'red'">
```

Son instrucciones muy simples como asignar a x el valor 30, hacer una ventana de alerta con el valor de x y cambiar el color del fondo a rojo.

Sin embargo, tantas instrucciones puestas en un manejador quedan un poco confusas, habría sido mejor crear una función:

```
  <script>
    function ejecutaEventoOnclick(){
         var x = 30;
         window.alert(x);
         window.document.bgColor = 'red';
    }
    </script>

    <FORM>
    <input type="button" value="Pulsar"
onclick="ejecutaEventoOnclick()">
    </FORM>
```

Siempre es una idea hacer un código mantenible y poner varias instrucciones en un atributo onclick no es una buena idea.

Manejadores de eventos asociados con addEventListener

La segunda forma de asociar manejadores de eventos a elementos de la página es mediante el método addEventListener(). Es una forma un poco más avanzada, pero mejora todavía la mantenibilidad del código, ya que permite separar mejor el código de la funcionalidad del código del contenido.

El código HTML debería usarse simplemente para definir el contenido de neustra página. Si tenemos instrucciones JavaScript dentro de las etiquetas, colocando atributos como "onclick" o "onchange" lo que estamos haciendo es colocar código de la funcionalidad dentro del código HTML, lo que es poco recomendable. Por tanto, la técnica que vamos a conocer ahora es todavía más adecuada, porque nos va a permitir escribir el código de la funcionalidad (los eventos JavaScript) sin ensuciar el código HTML.

Para asociar un evento a un elemento de la página necesitamos hacer dos pasos:

- Acceder al objeto sobre el que queremos definir el evento. Para esto tenemos que acceder al DOM para localizar el objeto adecuado, que representa la etiqueta sobre la que queremos asociar el manejador del evento.
- Sobre el objeto del DOM, aplicamos addEventListener(), indicando el tipo de evento y la función manejadora.

Por ejemplo, tenemos este elemento de la página:

```
  <input type="button" id="'miBoton'"
value="Pulsa click">
```

Lo más cómodo para acceder a un elemento de la página y recuperar el objeto del DOM asociado a esa etiqueta es usar el identificador (atributo "id"). En este caso el identificador es "miBoton". Para acceder a ese elemento usamos el método getElementById() del objeto document, enviando el identificador.

```
    var miBoton =
document.getElementById('miBoton');
```

Ahora tenemos el objeto del DOM asociado a ese botón en la variable "miBoton". Sobre ese objeto ya podemos invocar el método addEventListener(). A este método debemos pasarle dos parámetros. El primero es el tipo de evento que queremos detectar y el segundo la función manejadora del evento que queremos que se ejecute cuando se produce el evento.

```
    miBoton.addEventListener('click',
function() {
        alert('Has hecho clic!!')
    })
```

Al hacer clic sobre el botón se mostrará el mensaje de alerta.

Vamos a ver un segundo ejemplo, sobre una imagen en la que vamos a asociar un manejador para el tipo de evento "mouseover", que se produce cuando el usuario pone el puntero del ratón encima de un elemento.

Tenemos una imagen, en la que hemos puesto un atributo id para llegar a ella.

```
  <img
src="https://picsum.photos/200/300"
id="imagen">
```

Ahora le asociamos el manejador de evento para el tipo de evento "mouseover" con este código JavaScript.

```
    var miImagen =
document.getElementById('imagen');
    miImagen.addEventListener('mouseover'
, function() {
    alert('Has pasado el ratón encima de
la imagen')
    })
```

Apéndice de eventos de JavaScript

La siguiente tabla resume los eventos más importantes definidos por **JavaScript**:

Evento	Descripción	Elementos para los que está definido
onblur	Deseleccionar el elemento	<button>, <input>, <label>, <select>, <textarea>, <body>
onchange	Deseleccionar un elemento que se ha modificado	<input>, <select>, <textarea>
onclick	Pinchar y soltar el ratón	Todos los elementos
ondblclick	Pinchar dos veces seguidas con el ratón	Todos los elementos
onfocus	Seleccionar un elemento	<button>, <input>, <label>, <select>, <textarea>, <body>
onkeydown	Pulsar una tecla (sin soltar)	Elementos de formulario y <body>
onkeypress	Pulsar una tecla	Elementos de formulario y <body>

onkeyup	Soltar una tecla pulsada	Elementos de formulario y <body>
onload	La página se ha cargado completamente	<body>
onmousedown	Pulsar (sin soltar) un botón del ratón	Todos los elementos
onmousemove	Mover el ratón	Todos los elementos
onmouseout	El ratón *"sale"* del elemento (pasa por encima de otro elemento)	Todos los elementos
onmouseover	El ratón *"entra"* en el elemento (pasa por encima del elemento)	Todos los elementos
onmouseup	Soltar el botón que estaba pulsado en el ratón	Todos los elementos
onreset	Inicializar el formulario (borrar todos sus datos)	<form>
onresize	Se ha modificado el tamaño de la ventana del navegador	<body>
onselect	Seleccionar un texto	<input>, <textarea>
onsubmit	Enviar el formulario	<form>
onunload	Se abandona la página (por ejemplo al cerrar el navegador)	<body>

DOM

La finalidad de JavaScript es interactuar con el código HTML, para ello tenemos que tener muy claro lo que es el **DOM**. (Document Object Model). Es decir, el árbol de etiquetas que componen nuestro HTML.

Suponiendo que tenemos un elemento HTML con un id=caja podemos referirnos a él con:

```
    var
cajas=document.getElementById("caja").inn
erHTML;
```

De esta forma cargamos en la variable cajas el valor del elemento con id=caja.

También podemos modificar las propiedades o el valor del elemento con id=caja.

```
    var
cajas=document.getElementById("caja");
    cajas.innerHTML;
    cajas.style.background="red";
```

Otra forma de seleccionar un elemento de nuestra página es con querySelector.

```
    var
cajas=document.querySelector("#caja");
```

De esta forma selecciono también el elemento con id=caja.

Si quiero seleccionar un elemento cuya class=caja:

```
    var
cajas=document.querySelector(".caja");
```

Vemos pues cómo podemos, desde JavaScript, seleccionar y modificar elementos de HTML mediante el uso del DOM.

De esta forma si selecciono el elemento solo es para seleccionar el nombre de la etiqueta,

```
    var
cajas=document.querySelector("caja");
```

con # para hacer referencia a su ID

```
    var
cajas=document.querySelector("#caja");
```

y con punto para hacer referencia a su clase.

```
    var
cajas=document.querySelector(".caja");
```

Seleccionar clases y etiquetas

Acabamos de ver cómo seleccionar elementos concretos por su etiqueta, nombre o clase.

Para seleccionar todos los elementos de un tipo:

```
    var
todos_los_div=document.getElementsByTagNa
me('div');
```

Con esto he seleccionado todos los elementos de tipo **div** y me los mete en un array. Ahora ya podríamos sacar el contenido de un elemento de dicho array:

```
    var
contenido=todos_los_div[2].textContent();
    console.log(contenido);
```

Me mostraría lo que hay en el elemento [2] del array.

También podríamos haber utilizado el innerHTML:

```
    var
contenido=todos_los_div[2].innerHTML();
```

La única diferencia es que con **innerHTML(),** si quiero, puedo añadirle un valor nuevo al contenido.

También podemos seleccionar elementos por su etiqueta .class con:

document.getElemenByClassname();

BOM

El BOM (Browser Object Model) contiene multitud de propiedades que nos permiten trabajar con el navegador.

De igual forma que en el DOM podemos seleccionar objetos del documento, con el BOM podemos seleccionar objetos del navegador y modificarlos. Por ejemplo, podemos seleccionar la ventana:

```
console.log(window.innerHeight);
console.log(window.innerWidth);
```

En este caso jugamos con el tamaño de la ventana.

No existen estándares oficiales para el modelo de objetos de navegador (BOM) por lo que no suele ser muy usado, pero es interesante que el alumno sepa al menos de su existencia y funcionamiento.

Más cosas que podemos hacer:

console.log(screen.Width); Similar a lo anterior

Sacar la URL cargada y más datos:

console.log(window.location);

Sacar la Href:

console.log(window.location.href);

Abrir una pestaña nueva en el navegador:

window.open(URL_de_destino);