

Lenguaje Javascript básico

Índice

Introducción a Javascript	3
Conceptos elementales Javascript	3
Dónde colocar JS	5
Posibilidades de introducción de datos	7
Posibilidades de visualización de JavaScript	8
Variables	11
Hoisting	14
Tipos de datos	15
Tipado dinámico	15
Datos Primitivos	15
Objetos	17
Arrays	22
Funciones predefinidas	24
Operadores	33
Operadores aritméticos	33
Operadores relacionales	34
Operadores de asignación	35
Operadores de cadenas	36
Operadores lógicos	37
Estructuras de control de flujo	39
Condicionales	39
La declaración "if"	39
La declaración «else»	40
Operador condicional (ternario)	41
Condiciones complejas con "if"	42
Declaración "If else" anidada	42
Declaración «else if»	43
Declaración "switch"	46
Iteraciones	48
Bucle "while"	49

Bucle "do while"	50
Bucle "for"	52
Ruptura de bucle con «break»	54
Ruptura de bucle con «continue»	54
El bucle forEach	54
El bucle for...of	55
El bucle for...in	55
map() reduce() y filter()	57
Funciones	59
Alcance de las variables	63
Otras características	64
Funciones anidadas	64
Funciones anónimas y funciones de flecha	66
Paso de parámetros por valor o por referencia	67
Introducción al DOM	69
El objeto document	71
Introducción a los eventos	72
Formularios	76
Propiedades de formularios y elementos	76
Validación de formularios	79
Expresiones regulares en JS	81
Almacenamiento en el lado cliente	95
Cookies	95
API Web Storage	98

Fuentes

[Javier Pérez de Arrilucea](#)

<https://developer.mozilla.org>

<https://uniwebsidad.com/libros/javascript>

<https://www.w3schools.com/js>

[ECMAScript 5](#)

[ECMAScript 6 o 2015](#)

<https://javascript.info/>

www.javascripttutorial.net

Introducción a Javascript

JavaScript le permite agregar interactividad a una página web. Por lo general, utiliza JavaScript con HTML y CSS para mejorar la funcionalidad de una página web, como validar formularios, crear mapas interactivos y mostrar gráficos animados.

Cuando se carga una página web, es decir, después de que se hayan descargado HTML y CSS, el motor JavaScript del navegador web ejecuta el código JavaScript. Luego, el código JavaScript modifica el HTML y el CSS para actualizar la interfaz de usuario de forma dinámica.

El motor JavaScript es un programa que ejecuta código JavaScript. Al principio, los motores de JavaScript se implementaron como intérpretes.

Sin embargo, los motores de JavaScript modernos generalmente se implementan como compiladores justo a tiempo que compilan código de JavaScript en código de bytes para mejorar el rendimiento.

JavaScript del lado del cliente vs. del lado del servidor

Cuando se utiliza JavaScript en una página web, se ejecuta en los navegadores web. En este caso, JavaScript funciona como un lenguaje del lado del cliente.

JavaScript puede ejecutarse tanto en navegadores web como en servidores. Un entorno popular del lado del servidor de JavaScript es Node.js. A diferencia del JavaScript del lado del cliente, el JavaScript del lado del servidor se ejecuta en el servidor que le permite acceder a bases de datos, sistemas de archivos, etc.

Conceptos elementales Javascript

Para insertar JavaScript en una página HTML, se utiliza el elemento **<script>**.

Con la etiqueta de HTML <script> indicamos al navegador que vamos a empezar a escribir en lenguaje Javascript. A partir de ahí escribiremos todo nuestro código y cuando hayamos finalizado nuestro código en JS colocaremos la etiqueta de cierre </script>.

Hay dos formas de usar el elemento <script> en una página HTML:

- Incrustar el código JavaScript directamente en la página HTML.
- Hacer referencia a un archivo de código JavaScript externo.

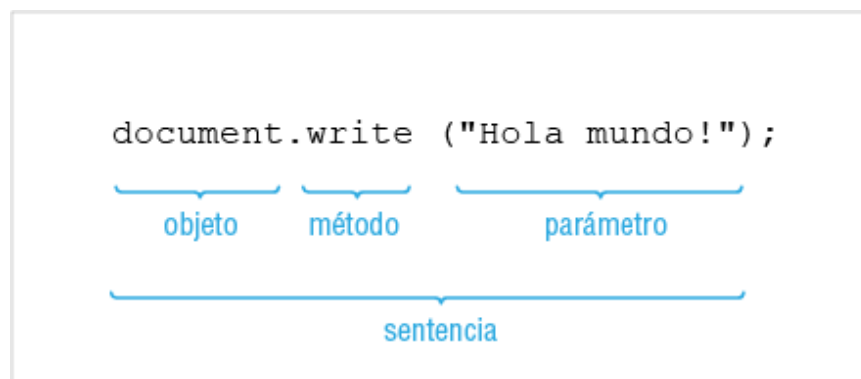
Veamos el siguiente ejemplo:

```
<!DOCTYPE html>
<html>
  <head>
  </head>
  <body>
    <script>
      document.write("Hola mundo!")
    </script>
  </body>
</html>
```

¿qué es una sentencia? Se llama así a cada una de las instrucciones que forman un script y que serán ejecutadas por el navegador.

La sentencia lo que hace es mostrar el texto "hola mundo! ".

Analicemos ahora el código en detalle.



El **objeto** sería el elemento de donde partimos para realizar la acción, en este caso es "document", el documento HTML que aparece al cargarse la página.

El **método** es una capacidad que tiene el objeto para hacer algo, en este caso sería escribir (write).

Y el **parámetro** es un dato que el método puede necesitar para ejecutar la acción. En este caso el método necesita que le pasemos el texto que deberá escribir.

En resumen, le hemos dicho al documento que escriba "Hola mundo!"

Ya solo nos falta comentar los signos de puntuación que aparecen en la línea de código que estamos analizando:

El **punto "."** que precede a "write" siempre debe colocarse entre el objeto y su método.

Los **paréntesis ()** van después del método y sirven para contener los parámetros.

Y por último el **punto y coma ";"**, que se coloca al final de cada sentencia para separar unas de otras. En este caso que solo hay una no sería necesario, pero es bueno acostumbrarse a ponerlos.

Dónde colocar JS

inline (script)

<script>... </script>

Puede bloquear la fase de parseo del HTML dependiendo de su posición en el HTML

Se puede colocar cualquier número de scripts en un documento HTML.

Los scripts se pueden colocar en el cuerpo <body>, o en la cabecera <head> de una página HTML, o en ambos.

Ejemplo en la cabecera

```
<!DOCTYPE html>
<html>
  <head>
    <title>ejemplo JS cabecera</title>
    <script>
      alert('Hello, World!');
    </script>
  </head>
  <body>
  </body>
</html>
```

Ejemplo en el cuerpo

```
<!DOCTYPE html>
<html>
<head>
  <title>ejemplo JS cuerpo</title>
</head>
<body>
  <script>
    alert('Hello, World!');
  </script>
</body>
</html>
```

La colocación de **scripts en la parte inferior del elemento <body> mejora la velocidad** de visualización mejor que en la cabecera, porque la interpretación del script ralentiza la visualización, bloquea el parseo del DOM, va a parecer que la página no está haciendo nada.

Los Trackers (código en javascript para uso de Google analytics,...) pueden requerir que se hagan en la cabecera.

Archivo externo

Existe la posibilidad de colocar nuestro **código Javascript en un archivo externo** con extensión «js», al cual llamaremos desde nuestro archivo HTML con la etiqueta **<script>** y el atributo «**src**».

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

Los scripts externos son prácticos cuando se usa el mismo código en muchas páginas web diferentes.

Los archivos JavaScript tienen la extensión de archivo .js .

```
<!DOCTYPE html>
<html>
<head>
  <title>Ejemplo JS archivo externo</title>
</head>
<body>
  <script src="js/app.js"></script>
</body>
</html>
```

Y el archivo js tendría el siguiente contenido:

```
alert('Hello, World!');
```

Colocar **scripts en archivos externos tiene algunas ventajas:**

- Separa HTML y código
- Permite reutilizar el código en varias páginas simplemente vinculando el archivo sin tener que reescribirlo.
- Hace que HTML y JavaScript sean más fáciles de leer y mantener
- Los archivos JavaScript almacenados en caché pueden acelerar la carga de la página, reduciéndose así los tiempos de descarga y acceso a las páginas.

Para agregar varios archivos de secuencia de comandos a una página, utilice varias etiquetas de secuencia de comandos:

Ejemplo

```
<script src="myScript1.js"></script>
```

```
<script src="myScript2.js"></script>
```

El motor JavaScript interpreta los archivos en el orden en que aparecen.

Referencias externas

Se puede hacer referencia a un script externo de 3 formas diferentes:

- Con una URL completa (una dirección web completa)
Ej: <script src="https://www.w3schools.com/js/myScript.js"></script>
- Con una ruta de archivo (como / js /)
Ej: <script src="/js/myScript.js"></script>
- Sin camino
Ej: <script src="myScript.js"></script>

Posibilidades de introducción de datos

JavaScript puede "leer" datos de diferentes formas:

- Escribiendo en un cuadro de alerta de window usando `prompt()`.
- Leyendo de un elemento HTML, usando los distintos input de un `<form>`.

Usando `prompt()`

El método `prompt ()` muestra un cuadro de diálogo que solicita al visitante una entrada.

A menudo se usa un cuadro de aviso si desea que el usuario ingrese un valor antes de ingresar una página.

Nota: Cuando aparece un cuadro de aviso, el usuario tendrá que hacer clic en "Aceptar" o "Cancelar" para continuar después de ingresar un valor de entrada. No abuse de este método, ya que evita que el usuario acceda a otras partes de la página hasta que se cierre el cuadro.

El método `prompt ()` devuelve el valor de entrada si el usuario hace clic en "Aceptar". Si el usuario hace clic en "cancelar", el método devuelve nulo.

```
prompt(text, defaultText_opcional)
```

Usando `<form>`

Se utiliza un formulario HTML para recopilar la entrada del usuario. La entrada del usuario se envía con mayor frecuencia a un servidor para su procesamiento.

El elemento HTML `<form>` se utiliza para crear un formulario HTML para la entrada del usuario, como: campos de texto, casillas de verificación, botones de opción, botones de envío, etc.

Un `<input>` elemento se puede mostrar de muchas formas, según el `type` atributo:

- `<input type="text">` Muestra un campo de entrada de texto de una sola línea.
- `<input type="radio">` Muestra un botón de opción (para seleccionar una de las muchas opciones)
- `<input type="checkbox">` Muestra una casilla de verificación (para seleccionar cero o más de muchas opciones)
- `<input type="submit">` Muestra un botón de envío del form.
- `<input type="button">` Muestra un botón en el que se puede hacer clic.

Posibilidades de visualización de JavaScript

JavaScript puede "mostrar" datos de diferentes formas:

- Escribiendo en un elemento HTML, usando innerHTML.
- Escribir en la salida HTML usando document.write().
- Escribiendo en un cuadro de alerta, usando window.alert().
- Escribiendo en la consola del navegador, usando console.log().

Usando innerHTML

Para acceder a un elemento HTML, en JavaScript puede usarse el método document.getElementById(id) que devuelve el elemento que posee el atributo «id» con el valor especificado.

El atributo id define el elemento HTML. La propiedad innerHTML define el contenido HTML:

Ejemplo

```
<!DOCTYPE html>
<html>
<body>
  <h1>Mi primera Web</h1>
  <p>Mi primer texto</p>
  <p id="demo"></p>
  <script>
    document.getElementById("demo").innerHTML = 5 + 6;
  </script>
</body>
</html>
```

Cambiar la propiedad innerHTML de un elemento HTML es una forma común de mostrar datos en HTML.

Usando document.write () o document.writeln ()

Para fines de prueba, es conveniente utilizar document.write() o document.writeln ():

Ejemplo

```
<!DOCTYPE html>
<html>
<body>
  <h1>Mi primera Web</h1>
  <p>Mi primer texto</p>
  <script>
    document.write(5 + 6);
  </script>
</body>
```



```
</html>
```

El uso de `document.write ()` después de cargar un documento HTML, eliminará todo el HTML existente :

Ejemplo

```
<!DOCTYPE html>
<html>
<body>
  <h1>Mi primera Web</h1>
  <p>Mi primer texto</p>
  <button type="button" onclick="document.write(5 + 6)">Haz click</button>
</body>
</html>
```

El método `document.write ()` solo debe usarse para pruebas.

Usando `window.alert ()`

Podemos utilizar un cuadro de alerta para mostrar datos:

Ejemplo

```
<!DOCTYPE html>
<html>
<body>
  <h1>Mi primera Web</h1>
  <p>Mi primer texto</p>
  <script>
    window.alert(5 + 6);
  </script>
</body>
</html>
```

Podemos omitir la palabra clave `window`.

En JavaScript, el objeto de ventana es el objeto de alcance global, lo que significa que las variables, propiedades y métodos por defecto pertenecen al objeto de ventana. Esto también significa que especificar la palabra clave `window` es opcional:

Ejemplo

```
<!DOCTYPE html>
<html>
<body>
  <h1>Mi primera Web</h1>
  <p>Mi primer texto</p>
  <script>
    alert(5 + 6);
  </script>
</body>
</html>
```

Usando console.log ()

Para fines de depuración, se puede llamar al método console.log() en el navegador para mostrar datos.

Aprenderemos más sobre la depuración en un apartado posterior.

Ejemplo

```
<!DOCTYPE html>
<html>
<body>
  <script>
    console.log(5 + 6);
  </script>
</body>
</html>
```

Impresión de JavaScript

JavaScript no tiene ningún objeto de impresión ni métodos de impresión.

No podemos acceder a los dispositivos de salida desde JavaScript.

La única excepción es que se puede llamar al método window.print() en el navegador para imprimir el contenido de la ventana actual.

Ejemplo

```
<!DOCTYPE html>
<html>
<body>
  <button onclick="window.print()">Imprimir</button>
</body>
</html>
```

Sintaxis de JavaScript

Variables

En este apartado vamos a hablar de los **datos** y de cómo debemos trabajar con ellos en Javascript.

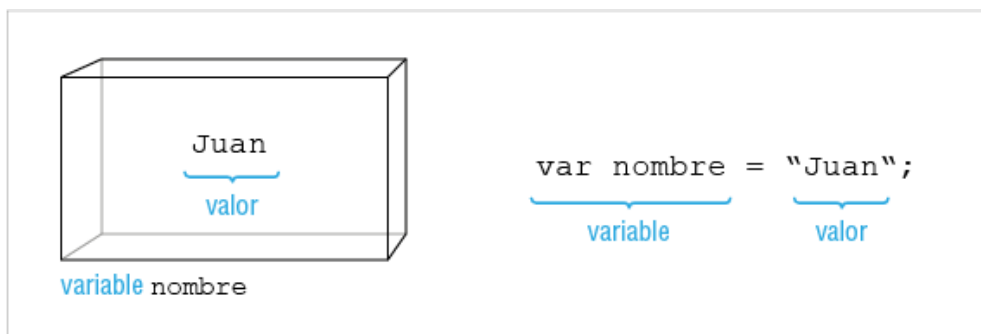
Cuando damos instrucciones a través del código que creamos vamos a estar continuamente trabajando con datos, por ejemplo los que nos puede proporcionar el usuario a través de un formulario. Para que podamos utilizar esos datos y operar con ellos debemos *guardarlos* en **variables**.

Hay 3 formas de declarar una variable de JavaScript:

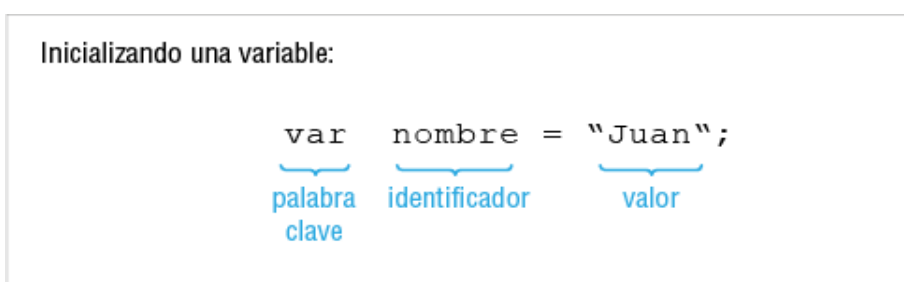
- Utilizando var
- Utilizando let
- Utilizando const

Variables var o let

- **Las variables** son una especie de contenedores que nos permiten almacenar valores para poder posteriormente acceder a ellos.



En el gráfico de arriba, la caja representa una **variable** llamada "nombre" que contiene el valor "Juan". A su derecha vemos el código necesario para crear dicha variable, lo que se denomina **inicializar** (1º declaramos la variable "nombre" y 2º le asignamos el valor "Juan"), veámoslo con más detalle.



1º Escribimos la palabra clave **var** que indica que estamos declarando (creando) una variable.

2º Colocamos el **identificador**, que es el nombre que le damos a esa variable.

3º El operador de asignación «=», que se necesita para asignar un valor a la variable.

4º Por último se escribe el **valor**, que en este caso como es un texto (lo que se denomina **cadena de texto** o **string**) debe de ir entre comillas.

- Vamos a repasar y aclarar algunos términos:

Variable: Sería como un contenedor donde se almacenan datos/valores.

Palabra clave o palabra reservada: Son las palabras que se usan para crear las sentencias (por ejemplo var), están en inglés y no pueden utilizarse como nombre para variables u otros elementos que creamos.

Identificador: El **nombre único** que se da a cada variable. Debe de seguir ciertas normas.

- No puede empezar con un número.
- Puede contener letras, números, guiones bajos (_) y el signo \$.
- Al ser Javascript case-sensitive (distingue entre mayúsculas y minúsculas), "Nombre" y "nombre" serían variables diferentes.
- En Javascript al igual que en otros lenguajes de programación, se recomienda el sistema **camelCase** para nombrar variables con más de una palabra. Se trata de comenzar las palabras con mayúsculas para así poder leerlas más fácilmente a pesar de estar unidas (la letra con la que comienza la variable suele ser minúscula).

```
var primerNombre = "Juan";
```

- Y como habíamos comentado, no se pueden utilizar las **palabras clave**.

Declarar: Definir por primera vez una variable, se hace con "var" seguido del identificador (var nombre;). No es necesario asignarle un valor en el momento de la creación, se puede asignar más adelante.

JavaScript es un lenguaje de tipado dinámico. Esto significa que no necesita especificar el tipo de variable en la declaración.

Inicializar: Si en el momento de declarar una variable también se le asigna un valor. Ejemplo:

```
<!DOCTYPE html>
<html>
<head>
  <title>Documento definición variables</title>
</head>
<body>
  <h2>Variables en JavaScript</h2>
  <p>Suma de dos variables x=5, y=6.</p>
  <p id="demo"></p>
  <script>
    var x = 5;
```

```
var y = 6;
var z = x + y;
document.getElementById("demo").innerHTML = "La suma es: " + z;
</script>
</body>
</html>
```

¿Por qué se llaman variables?

La característica más importante de las variables es que su contenido no es constante, de ahí radica su verdadera utilidad.

Gracias a ello podemos ir modificando el valor que contiene cuando lo necesitemos, veamos un ejemplo:

Imagina que queremos crear un script en el que recojamos por medio de un formulario el nombre del usuario, y después le demos la bienvenida.

Ejemplo

```
<!DOCTYPE html>
<html>
  <head>
  </head>
  <body>
    <script>
      var nombre;
      nombre = prompt("Introduzca su nombre:");
      document.write ("Bienvenido " + nombre);
    </script>
  </body>
</html>
```

El signo **"+"** que aparece en el código es el operador que nos permite concatenar (sumar) cadenas de texto, y como en este caso una cadena y una variable.

Al meter el nombre del usuario en la variable, podremos utilizar este script para todos los usuarios, tan solo debemos cambiar el valor de la variable "nombre" por el que introduzca el usuario en el formulario.

Comentarios: Para hacer comentarios en Javascript tenemos dos opciones.

```
// para comentar una línea
/* para comentar
varias líneas */
```

Variables Let

Las variables definidas con let no se pueden volver a declarar.

Las variables definidas con let deben declararse antes de su uso.

El alcance (scope) de Let es a nivel de bloque.

```
function varScope() {
  var x = 1;
  if (true) {
    var x = 7; // misma variable!
    console.log(x); // 7
  }
  console.log(x); // 7
}
function letScope() {
  let x = 1;
  if (true) {
    let x = 7; // variable diferente
    console.log(x); // 7
  }
  console.log(x); // 1
}
```

Mientras el scope requerido sea el mismo se recomienda utilizar `let` tanto como sea posible en tu código, en lugar de `var`. No hay ninguna razón para usar `var`, a menos que necesites admitir versiones antiguas de Internet Explorer con tu código (no es compatible con `let` hasta la versión 11; Edge el moderno navegador de Windows admite `let` perfectamente).

Variables const

Las variables definidas con `const` no se pueden volver a declarar.

Las variables definidas con `const` no se pueden reasignar.

Así pues, tenemos la palabra clave `const`, que nos permite almacenar valores que nunca se pueden cambiar:

```
const daysInWeek = 7;
const hoursInDay = 24;
```

const funciona exactamente de la misma manera que `let`, excepto que a `const` no le puedes dar un nuevo valor. En el siguiente ejemplo, la segunda línea arrojará un error:

```
const daysInWeek = 7;
daysInWeek = 8;
```

Se suele usar **const** en la definición de objetos y arrays, ya que en sí no cambian sino que la modificación de los valores se hace normalmente a través de sus atributos.

Hoisting

Durante la fase de compilación se asigna memoria a las declaraciones de variables y funciones. En el caso de las variables `var` se les asignan el valor `undefined`, por lo que si accedemos a ellas antes de declararla tendrá dicho valor `undefined`, pero no ocurre lo mismo con `let` o `const` que darían un error, se encuentran en una zona denominada **dead zone**.

Tipos de datos

Tipado dinámico

JavaScript es un "lenguaje tipado dinámicamente", por lo que no es necesario especificar qué tipo de datos contendrá una variable (números, cadenas, arreglos, etc.).

Por ejemplo, si declaras una variable y le das un valor entre comillas, el navegador trata a la variable como una cadena (string):

```
let myString = 'Hello';
```

Incluso si el valor contiene números, sigue siendo una cadena, así que ten cuidado:

```
let myNumber = '500'; // Vaya, esto sigue siendo una cadena
typeof myNumber;
```

```
myNumber = 500; // mucho mejor — ahora este es un número
typeof myNumber;
```

Intenta ingresar las cuatro líneas anteriores en tu consola una por una y ve cuáles son los resultados. Notarás que estamos usando un operador especial llamado `typeof` — esto devuelve el tipo de datos de la variable que escribes después. La primera vez que se llama, debe devolver `string`, ya que en ese punto la variable `myNumber` contiene una cadena, `'500'`. Es muy útil para conocer el tipo que tiene la variable sobre la que operamos.

Ejemplos:

```
typeof 5;           // number
typeof false;       // boolean
typeof "Carlos";    // string
typeof undefined;   // undefined
```

Datos Primitivos

Vamos a conocer los **tipos de datos primitivos** más usuales que podemos almacenar en las variables y algunas de sus características.

A la hora de asignar valores a una variable debemos tener en cuenta el tipo de dato, porque éste condicionará la manera en que lo asignemos.

Numéricos: Números enteros o decimales, positivos o negativos.

Los decimales deben ser escritos con punto en vez de coma.

```
let x = 12;
let y = 12.00; // número con decimales
```

Booleanos: Los únicos dos valores que admite son **"true"** o **"false"** (verdadero o falso), aunque ahora nos suene un poco extraño, ya nos iremos familiarizando con ellos y viendo su utilidad. De momento solo debemos saber que nos servirán cuando en los programas se tengan que tomar decisiones, es decir cuando haya que realizar una acción dependiendo de que algo sea cierto (true) o falso (false), por ejemplo, mostrar un determinado contenido si a la comprobación de si hoy es Domingo, el resultado es cierto (true).

```
let x = true;
let y = false;
```

Cadenas de texto (strings): Caracteres alfanuméricos, palabras y frases.

Para asignar un valor de este tipo a una variable hay que encerrarlo **entre comillas**, bien sean sencillas o dobles. En el caso de que el texto incluya comillas, las comillas que encierran a éste deberán ser diferentes, por ejemplo si son dobles, el string deberá ser encerrado con simples.

```
let texto1 = "esto es una cadena de texto";
let texto2 = 'esto es una "cadena de texto" con comillas dentro';
let texto3 = "esto es una 'cadena de texto' con comillas simples dentro";
```

Si incluimos números en una cadena de texto, serán tomados como simple texto y no se podrá operar matemáticamente con ellos.

- Aunque existen más tipos de datos estos son los más básicos y con los que trabajaremos en nuestros primeros pasos en Javascript. Vamos ahora a practicar unas pequeñas operaciones con los valores de tipo numérico y string que acabamos de aprender.

```
<script>
  let resultado = 10 + 15 - 3;
  document.write ("El resultado es: " + resultado);
</script>
```

En el código de arriba hemos guardado en la variable "resultado" el **valor numérico** de una operación aritmética en la que hacemos una suma y una resta de tres números utilizando los operadores "+" y "-".

En la siguiente línea hemos dado la instrucción de que se escriba la **cadena de texto** "El resultado es: " más el contenido de la variable "resultado". Con lo que el navegador nos tiene que mostrar: **"El resultado es: 22"**.

- Hagamos ahora una prueba algo diferente, vamos a mezclar en una variable dos tipos de valores, numéricos y cadena de texto.

```
<script>
  let resultado = "Juan" + 15 + 3;
  document.write (resultado);
</script>
```


Podemos comprobar cómo el navegador al encontrarse con valores numéricos después del string, los trata como si también fuesen cadenas de texto y se limita a escribirlos uno detrás de otro: **"Juan153"**

- Probemos ahora a invertir la posición de los elementos.

```
<script>
    let resultado = 15 + 3 + "Juan";
    document.write (resultado);
</script>
```

En este caso al encontrarse primero con los valores numéricos, realiza la operación de suma para después añadirle la cadena de texto, siendo el resultado en pantalla: "18Juan".

En resumen, **el orden de los elementos incide en los resultados.**

El carácter de escape de barra invertida (\) convierte los caracteres especiales en caracteres de cadena:

Code	Result	Description
\'	'	Single quote
\"	"	Double quote
\\	\	Backslash

Ejemplo

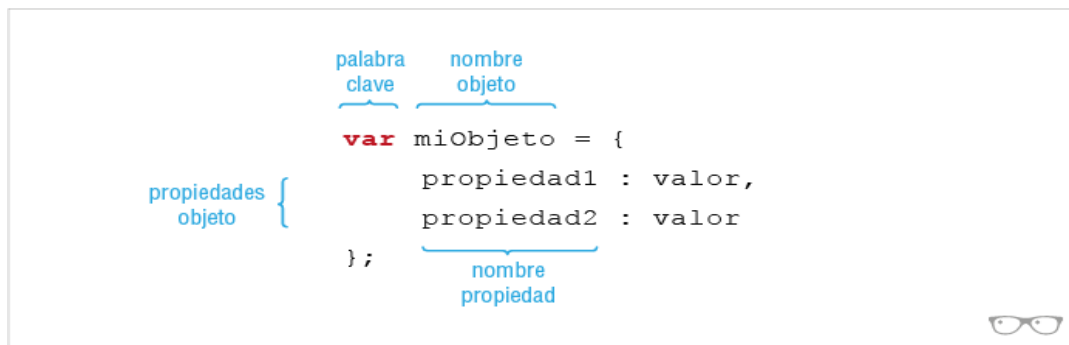
```
<html>
<body>
    <h2>JavaScript Strings</h2>
    <p>La secuencia \" inserta una comilla doble en una cadena.</p>
    <p id="demo"></p>
    <script>
        let text = "Somos los llamados \"Vikingos\" del norte.";
        document.getElementById("demo").innerHTML = text;
    </script>
</body>
</html>
```

Objetos

En JavaScript, un objeto es una colección desordenada de pares clave-valor. Cada par clave-valor se denomina propiedad.

La clave de una propiedad puede ser una cadena. Y el valor de una propiedad puede ser cualquier valor, por ejemplo, una cadena, un número, una matriz e incluso una función.

JavaScript le proporciona muchas formas de crear un objeto. El más comúnmente usado es usar la notación literal de objeto.



Sintaxis objeto

Por ejemplo, lo siguiente crea un nuevo objeto person:

```
let person = {
  firstName: 'John',
  lastName: 'Doe'
};
```

El objeto person tiene dos propiedades firstName y lastName.

Otro modo de definir y crear un objeto único (instancia) es con la palabra clave **«new»**. (Es preferible por simplicidad y rapidez utilizar la primera manera en vez de esta).

```
var miObjeto = new Object({propiedad1:valor, propiedad2:valor,...});
```

```
var person = new Object();
person.firstName: 'John';
person.lastName: 'Doe';
```

Accediendo a las propiedades

Para acceder a una propiedad de un objeto, utiliza una de dos notaciones: la notación de puntos y la notación tipo matriz.

1) La notación de punto (.)

A continuación se ilustra cómo utilizar la notación de puntos para acceder a una propiedad de un objeto:

objectName.propertyName

Por ejemplo, para acceder a la propiedad firstName del objeto person, utiliza la siguiente expresión:

```
person.firstName
```

Este ejemplo crea un objeto person y muestra el nombre y el apellido en la consola:

```
let person = {
  firstName: 'John',
  lastName: 'Doe'
};
```

```
console.log(person.firstName);  
console.log(person.lastName);
```

2) Notación tipo matriz ([])

A continuación se ilustra cómo acceder al valor de la propiedad de un objeto a través de la notación tipo matriz:

`objectName['propertyName']`

Por ejemplo:

```
let person = {  
  firstName: 'John',  
  lastName: 'Doe'  
};  
console.log(person['firstName']);  
console.log(person['lastName']);
```

Cuando el nombre de una propiedad contiene espacios, debe colocarlo entre comillas. Por ejemplo, el siguiente objeto `address` tiene `'building no'` como propiedad:

```
let address = {  
  'building no': 3960,  
  street: 'North 1st street',  
  state: 'CA',  
  country: 'USA'  
};
```

Para acceder a la propiedad `'building no'`, debe usar la notación similar a una matriz:

```
address['building no'];
```

Si usa la notación punto: `address.'building no';`

obtendrá un error: `SyntaxError: Unexpected string`

Modificar el valor de una propiedad

Para cambiar el valor de una propiedad, utilice el operador de asignación (`=`). Por ejemplo:

```
let person = {  
  firstName: 'John',  
  lastName: 'Doe'  
};  
person.firstName = 'Jane';  
console.log(person); //{ firstName: 'Jane', lastName: 'Doe' }
```

En este ejemplo, cambiamos el valor de la propiedad `firstName` del objeto `person` de `'John'` a `'Jane'`.

Agregar una nueva propiedad a un objeto

A diferencia de los objetos en otros lenguajes como Java y C#, puede agregar una propiedad a un objeto después de la creación del objeto.

La siguiente declaración agrega la propiedad age al objeto person y le asigna 25:

```
person.age = 25;
```

Eliminar una propiedad de un objeto

Para eliminar una propiedad de un objeto, utiliza el delete operador:

```
delete objectName.propertyName;
```

El siguiente ejemplo elimina la age propiedad del person objeto:

```
delete person.age;
```

Si intenta volver a acceder a la propiedad de edad, obtendrá un valor undefined.

Comprobar si existe una propiedad

Para comprobar si existe una propiedad en un objeto, utiliza el operador **in**:
propertyName in objectName

El operador in devuelve true si propertyName existe en el objectName.

El siguiente ejemplo crea un objeto employee y usa el operador in para verificar si las propiedades ssn y employeeId existen en el objeto:

```
let employee = {  
  firstName: 'Peter',  
  lastName: 'Doe',  
  employeeId: 1  
};  
console.log('ssn' in employee); //false  
console.log('employeeId' in employee); //true
```

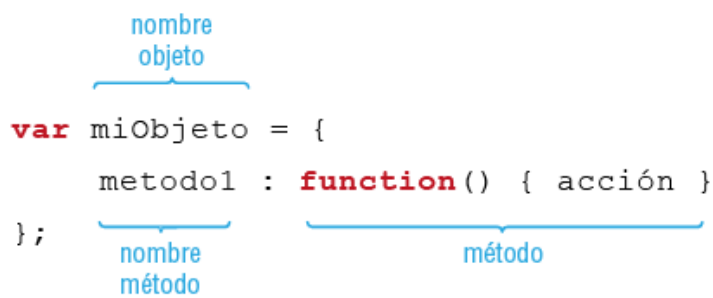
Métodos

Los métodos son las acciones que pueden ejecutar los objetos. Como ya adelantamos anteriormente los métodos son funciones dentro de objetos. Más concretamente son funciones almacenadas como propiedades de un objeto.

Existen dos maneras de incluir esas funciones en nuestros objetos:

1) Directamente cuando definimos el método

Guardando en la propiedad que representa al método una función anónima. La función anónima no posee un nombre propio sino que adopta el del método que la contiene.



```

var miObjeto = {
  metodo1 : function() { acción }
};

```



Sintaxis definición de un método en objeto

Ejemplo:

```

var cliente1 = {
  nombre: "Juan",
  edad: 30,
  saludar: function() { alert("Hola " + cliente1.nombre); }
};

```

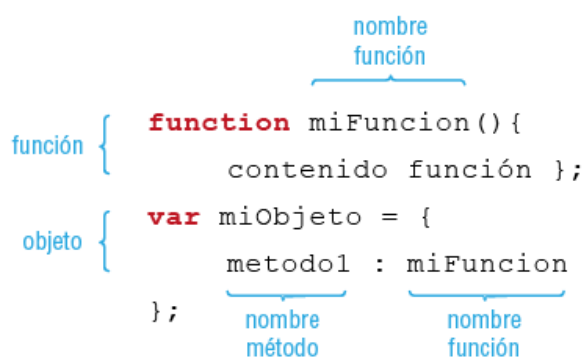
```
cliente1.saludar();
```

En el ejemplo de arriba hemos definido un método llamado «saludar». Para ello simplemente hemos colocado dos puntos «:» entre el nombre del método y la función.

Posteriormente para llamar al método colocamos el nombre del objeto seguido de un punto y del nombre del método, para finalizar escribimos dos paréntesis «()».

2) Asignando una función ya existente

Asignando al método el nombre de la función creada previamente.



```

function miFuncion() {
  contenido función };

var miObjeto = {
  metodo1 : miFuncion
};

```



Sintaxis definición de un método llamando a función

Ejemplo:

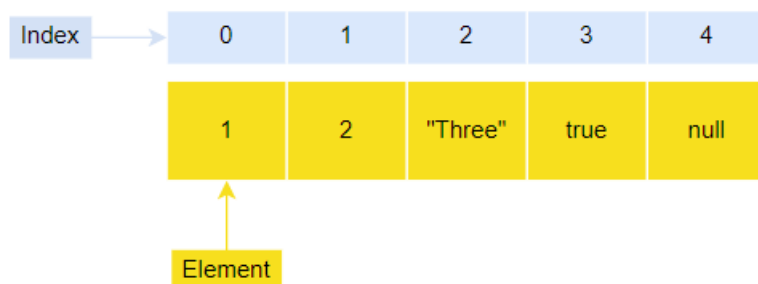
```
function fSaludar (nombre){  
    alert("Hola " + nombre);  
}; // Se declara la función con un parámetro  
  
var cliente1 = {  
    saludar: fSaludar // Asignamos la función al método  
};  
  
cliente1.saludar("Santi"); // Invocamos el método y le pasamos un argumento
```

En el ejemplo de la parte superior también hemos añadido un parámetro a la función, al invocar el método le hemos pasado el argumento como si de una función cualquiera se tratase.

Arrays

Introducción a los arrays de JavaScript

En JavaScript, un array es una lista ordenada de valores. Cada valor se denomina elemento especificado por un índice:



Un array de JavaScript tiene las siguientes características:

1. Primero, un array puede contener valores de tipos mixtos. Por ejemplo, puede tener una matriz que almacene elementos con los tipos número, cadena, booleano y nulo.
2. En segundo lugar, el tamaño de un array es dinámico y de crecimiento automático. En otras palabras, no necesita especificar el tamaño del array por adelantado.

Creación de arrays de JavaScript

JavaScript le proporciona dos formas de crear un array. El primero es usar el constructor Array de la siguiente manera:

```
let scores = new Array();
```

El array scores está vacío, lo que contiene elementos.

Si conoce la cantidad de elementos que contendrá el array, puede crear un array con un tamaño inicial como se muestra en el siguiente ejemplo:

```
let scores = Array(10);
```

Para crear un array e inicializarlo con algunos elementos, pasa los elementos como una lista separada por comas al constructor `Array()`.

Por ejemplo, lo siguiente crea el array `scores` que tiene cinco elementos (o números):

```
let scores = new Array(9,10,8,7,6);
```

Tenga en cuenta que si usa el constructor `Array()` para crear un array y le pasa un número, está creando un array con un tamaño inicial.

Sin embargo, cuando pasa un valor de otro tipo `string` al constructor `Array()`, crea un array con un elemento de ese valor. Por ejemplo:

```
let athletes = new Array(3); //crea un array con tamaño inicial 3
let scores = new Array(1, 2, 3); //crea un array con los números 1,2 3
let signs = new Array('Red'); // crea un array con un elemento 'Red'
```

JavaScript le permite omitir el operador `new` cuando usa el constructor `Array()`. Por ejemplo, la siguiente declaración crea el array `artists`.

```
let artists = Array();
```

En la práctica, rara vez usará el constructor `Array()` para crear un array.

La forma preferida de crear un array es usar la notación literal de array:

```
let arrayName = [element1, element2, element3, ...];
```

La forma literal de array usa los corchetes `[]` para envolver una lista de elementos separados por comas.

El siguiente ejemplo crea el array `colors` que contiene elementos de cadena:

```
let colors = ['red', 'green', 'blue'];
```

Para crear un array vacío, usa corchetes sin especificar ningún elemento como este:

```
let emptyArray = [];
```

Acceso a elementos de un array de JavaScript

Las matrices o arrays de JavaScript están indexadas en base cero. En otras palabras, el primer elemento de una matriz comienza en el índice 0, el segundo elemento comienza en el índice 1 y así sucesivamente.

Para acceder a un elemento en un array, especifica un índice entre corchetes []:

```
arrayName[index]
```

A continuación se muestra cómo acceder a los elementos del array mountains:

```
let mountains = ['Everest', 'Fuji', 'Nanga Parbat'];

console.log(mountains[0]); // 'Everest'
console.log(mountains[1]); // 'Fuji'
console.log(mountains[2]); // 'Nanga Parbat'
```

Para cambiar el valor de un elemento, asigna ese valor al elemento de esta manera:

```
let mountains = ['Everest', 'Fuji', 'Nanga Parbat'];
mountains[2] = 'K2';
console.log(mountains);
```

Producción:

```
[ 'Everest', 'Fuji', 'K2' ]
```

Funciones predefinidas

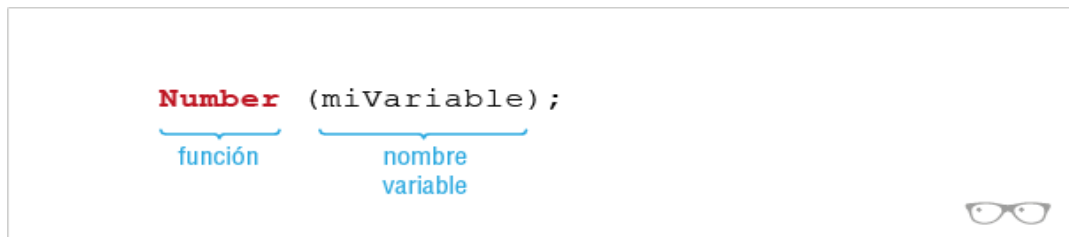
Vamos a ver las funciones predefinidas que nos ofrece Javascript.

Javascript posee un conjunto de **funciones ya definidas** que podemos utilizar con diferentes tipos de datos. Estas funciones **se denominan predefinidas o globales**.

Funciones o Métodos de números

Función Number ()

Esta función nos permite convertir diferentes valores a números. Si la conversión no es posible devolverá el valor NaN (Not a Number).



Sintaxis-función Number

Veamos un ejemplo:

```
var miEdad = "27";  
document.write (Number(miEdad) + 3);
```

En la variable «miEdad» hemos guardado la cadena de texto «27».

A la hora de mostrar esa variable en un alert hemos convertido su valor a un número, utilizando la función «Number» y pasándole como argumento la variable que queremos convertir «miEdad».

Como prueba de que el valor se ha convertido en número se ha podido sumar el número 3.

Función String ()

Esta función nos permite convertir cualquier valor en una cadena de texto (string).

Su sintaxis es igual a la de «Number».

```
document.write (String(2016) + " " + String(true));
```

En este caso hemos convertido un número y el valor booleano «true» a texto y los hemos concatenado.

Función parseInt ()

Permite convertir una cadena de texto en un número entero (integer).

Funciona bajo las siguientes reglas:

Solo convierte caracteres válidos, si encuentra uno no válido o un espacio, se detiene y solo convierte los caracteres desde el comienzo hasta ese punto.

Si el primer carácter a convertir no es válido devolverá NaN.

No tiene en cuenta los espacios al comienzo de la cadena.

La mejor manera de verlo es con un ejemplo.

```
document.write (parseInt("2.5") + "<br/>");  
document.write (parseInt("123 45") + "<br/>");  
document.write (parseInt("a89"));
```

Esta función acepta un segundo argumento que nos permite indicar la base de numeración en la que está el valor que se va a convertir.

Binario (2), octal (8), hexadecimal (16).

Si no se especifica ningún parámetro se tomará como decimal. Una vez hecha la conversión el valor estará en base 10.

```
document.write (parseInt("11001", 2));
```

En el ejemplo de arriba tomará la cadena de texto «11001» como un número binario y lo convertirá en un número decimal.

El separador decimal en Javascript es el punto en vez de la coma.

Función parseFloat ()

Nos permite convertir una cadena en un número real (float) atendiendo a las mismas reglas que «parseInt», la única diferencia es que admite el punto "." como separador decimal.

```
document.write (parseFloat("2.5") + "<br/>"); // El resultado será 2.5
document.write (parseFloat("2,5") + "<br/>"); // El resultado será 2
document.write (parseFloat("a89")); // El resultado será NaN
```

En el ejemplo superior en el caso de 2,5 el resultado será 2 ya que como hemos comentado anteriormente "parseFloat" solo acepta números, y el signo punto como separador decimal.

Función isNaN ()

Determina si el valor que le pasamos **no** es un número.

En el caso de que no sea un número devuelve «true» y si es un número devuelve «false».

```
document.write (isNaN(2.5) + "<br/>"); // Devolverá false
document.write (isNaN("2.5") + "<br/>"); // Devolverá false
document.write (isNaN("palabra") + "<br/>"); // Devolverá true
document.write (isNaN(true)); // Devolverá false
```

En el último caso del ejemplo de arriba, devolverá false ya que antes de la comprobación se aplica la conversión de tipos, por la que true es 1.

Función isFinite ()

Antes de ver ésta función conozcamos dos tipos de **valores especiales** que todavía no hemos visto, «infinity» e «-infinity».

Infinity:

Indica que el resultado de una operación ha alcanzado un valor demasiado alto para Javascript.

El número más alto que acepta Javascript es 1.7976931348623157e+308, un número mayor es considerado infinito.

-Infinity:

Es igual a «infinity» pero aplicado a los **números negativos**.

Probemos con un ejemplo.

```
document.write ((2/0) + "<br/>"); // Devolverá infinity
document.write ((-2/0) + "<br/>"); // Devolverá -infinity
```

La función «**isFinite**» determina si el valor es finito. En el caso de que sea finito devuelve true y si es un valor **de tipo infinity, -infinity o NaN, devuelve true**.

Veámoslo:

```
document.write (isFinite(25) + "<br/>"); // Devolverá true
document.write (isFinite(2/0) + "<br/>"); // Devolverá false
document.write (isFinite(-2/0) + "<br/>"); //Devolverá false
```

Función encodeURIComponent () / decodeURI ()

Esta función es utilizada para codificar una URI. Esto significa que transforma los caracteres especiales excepto (, / ? : @ & = + \$ #) a formato UNICODE, de modo que al leer una cadena que contenga caracteres especiales, no se produzcan errores.

```
document.write (encodeURIComponent("Qué tal?"));
```

Al ejecutarse la sentencia de arriba, nos mostrará la cadena de texto codificada, en este caso sustituirá la «e» acentuada y el espacio en blanco transformándolos mediante código UTF-8 a formato UNICODE. El resultado será: Qu**%C3%A9%20**tal?

UTF-8 es el formato de codificación preferido para e-mail y webs.

En HTML-5 la codificación de caracteres por defecto es en UTF-8, también acepta UTF-16 pero hay que especificarlo.

Con anterioridad se utilizaba la función escape() que es similar a encodeURIComponent(), pero está en desuso.

Con decodeURI() realizaremos el proceso inverso. Anteriormente se utilizaba unescape().

Función encodeURIComponent () / decodeURIComponent ()

De manera parecida a encodeURIComponent() se utiliza para codificar una URI. Codifica los caracteres especiales, incluidos (, / ? : @ & = + \$ #).

Probemos con un ejemplo muy común, una URL.

```
document.write (encodeURIComponent("http://www.misitio.com"));
```

En este caso sustituye el carácter «:» y las dos barras, por código UTF-8.

`http%3A%2F%2F`www.misitio.com

Función eval ()

Evalúa una cadena ejecutándola como si fuese un fragmento de código Javascript.

```
document.write (eval("var cantidad = 20; cantidad + 5"));
```

En este último ejemplo el navegador interpretará el contenido de la cadena de texto y lo ejecutará como código, dando el resultado de 25.

Como hemos podido ver en este caso se puede colocar más de una instrucción, tan solo necesitamos el punto y coma para separarlas.

Funciones o Métodos de cadenas

Para encontrar la *longitud de una cadena*, usamos la propiedad **length**:

Ejemplo

```
let text = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";  
let length = text.length;
```

Extracción de partes de cadenas

Hay 3 métodos para extraer una parte de una cadena:

- `slice(start, end)` El primer parámetro es la posición del carácter en la que comenzar a extraer, y el segundo parámetro es la posición del carácter posterior al último a ser extraído. Si no se incluye el segundo parámetro lo considera hasta el final de la cadena.
- `substring(start, end)`
- `substr(start, length)`

El método **replace()**

Reemplazo del contenido de la cadena.

replace('strOld', 'strNew') Toma dos parámetros — la cadena que deseas reemplazar, y la cadena con la que deseas reemplazarla.

Conversión a mayúsculas y minúsculas

Una cadena se convierte a mayúsculas con **toUpperCase()**:

Una cadena se convierte a minúsculas con **toLowerCase()**:

El método **concat ()** o el operador **+**

`concat()` o el operador `+` une dos o más cadenas

Template String (Comillas invertidas)

Las comillas invertidas nos permiten extender funcionalidad, incrustar variables y expresiones en una cadena envolviéndolas en `${...}`.

Por ejemplo:

```
let name = "Juan";  
// embed a variable  
console.log( 'Hola, ${name}!' ); // Hola, Juan!  
// embed an expression  
alert( 'El resultado es ${1 + 2}' ); // El resultado es 3
```

La expresión interna `${...}` se evalúa y el resultado se convierte en parte de la cadena. Podemos poner cualquier cosa ahí: una variable como `name` o una expresión aritmética como `1 + 2` o algo más complejo. Los templates strings también nos permiten llamar a una función.

Funciones o Métodos de arrays

La propiedad de longitud

La propiedad **length** de una matriz devuelve la longitud de una matriz (el número de elementos de la matriz).

Ejemplo

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];  
let length = fruits.length;
```

La propiedad `length` es siempre uno más que el índice de matriz más alto.

Eliminar y agregar elementos

Cuando trabaja con matrices, es fácil eliminar elementos y agregar elementos nuevos.

Método **pop()**

Elimina el último elemento de una matriz, desapila:

Ejemplo

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.pop();
```

El método `pop()` devuelve el valor que "apareció":

Ejemplo

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];  
let fruit = fruits.pop(); //fruit= Mango
```

El método **shift()** elimina el primer elemento de la matriz, desencola.

Ejemplo

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];  
let fruit = fruits.shift(); //fruit= Banana
```

Método push() agrega un nuevo elemento a una matriz (al final), apila:

Ejemplo

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.push("Kiwi");
// fruits = ["Banana", "Orange", "Apple", "Mango", "Kiwi"];
```

El método **unshift()** agrega un nuevo elemento al inicio de la matriz, encola.

Ejemplo

```
fruits.unshift("Melon");
// fruits = ["Melon", "Banana", "Orange", "Apple", "Mango", "Kiwi"];
```

Método concat()

Este método devuelve un array como resultado de la unión de dos o más arrays.

```
var miArray = ["red", "green", "blue", "yellow"];

var otroArray = ["orange", "pink", "grey", "silver"];

alert(sumaArrays = miArray.concat(otroArray));
```

Método slice()

Este método toma una parte de un array y lo convierte en uno nuevo.

```
var miArray = ["cero", "uno", "dos", "tres", "cuatro"];

var nuevoArray = miArray.slice(1, 3); // Tomará los valores en las
posiciones entre 1 y 3 (incluyendo la 1 y exceptuando la 3)

alert(nuevoArray); // nuevoArray mostrará "uno, dos"
```

En el ejemplo de arriba a slice le hemos pasado **dos argumentos**, el primero indica cual es la posición del primer valor que debe tomar y el segundo cual es la posición del valor que ya **no** debe tomar.

En el caso de que solo le pasemos **un argumento**, slice tomará todos los valores desde la posición de ese argumento hasta el final del array.

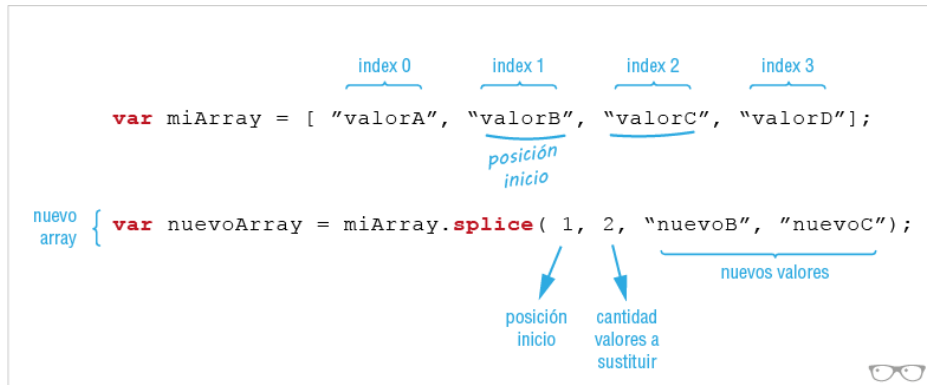
```
var miArray = ["cero", "uno", "dos", "tres", "cuatro"];

var nuevoArray = miArray.slice(2); // Tomará los valores desde la posición
2 hasta el final

alert(nuevoArray); // nuevoArray mostrará "dos, tres, cuatro"
```

Método splice()

Este método nos permite sustituir o borrar elementos de un array.



Sintaxis método slice

Veamos un ejemplo:

```
var miArray = ["cero", "uno", "dos", "tres", "cuatro"];
```

```
miArray.splice(2, 2, "DOS", "TRES");
```

```
alert(miArray); // Resultado "cero, uno, DOS, TRES, cuatro"
```

Con el primer argumento que pasamos a splice() le **indicamos que se situe** en la posición 2.

Con el segundo argumento le indicamos que **elimine** dos valores, en este caso a partir de la posición dos que es donde nos hemos situado.

Finalmente, con los dos últimos argumentos, le damos los valores que vamos a **añadir** en sustitución de los eliminados.

En el caso de que tan solo queramos eliminar elementos, solo utilizaremos los dos primeros argumentos.

```
var miArray = ["cero", "uno", "dos", "tres", "cuatro"];
```

```
miArray.splice(3, 2); // Eliminamos dos elementos desde la posición 3
```

```
alert(miArray);
```

También podemos eliminar elementos con el operador «delete», el único inconveniente es que eliminará el valor de la posición que le indiquemos pero dejara la posición como «undefined».

```
delete miArray[1];
```

Método reverse()

El método «reverse» invierte el orden de los valores de un array.

```
var miArray = ["tomate", "lechuga", "acelga", "berenjena"];

alert(miArray.reverse());
```

Método sort()

Con el método "sort" podemos ordenar alfabéticamente los valores de un array.

```
var miArray = ["tomate", "lechuga", "acelga", "berenjena"];

alert(miArray.sort());
```

Método valueOf()

Es un método que poseen todos los objetos de Javascript y devuelve el propio array.

```
miArray.valueOf(); // Es lo mismo que escribir: miArray
```

Conversión de matrices en cadenas

Método toString() convierte una matriz en una cadena de valores de matriz (separados por comas). O con **split(',')**.

Ejemplo

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo").innerHTML = fruits.toString();
```

Resultado:

Banana,Orange,Apple,Mango

El método **join()** también une todos los elementos de la matriz en una cadena. Se comporta igual toString(), pero además puedes especificar el separador:

Ejemplo

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo").innerHTML = fruits.join(" * ");
```

Resultado:

Banana * Orange * Apple * Mango

Operadores

Operadores aritméticos

En este capítulo trataremos **los operadores**, los signos con los que podemos realizar múltiples operaciones con los datos.

Hasta ahora, aparte del operador de asignación "=" hemos visto dos operadores aritméticos, el "+" y el "-". A continuación tenemos una lista de todos los operadores aritméticos con los que podemos trabajar con números.

+	Adición
-	Substracción
*	Multiplicación
/	División
%	Módulo (resto)
++	Incremento
--	Decremento

El de **adición**, como ya hemos visto, también nos sirve para concatenar (sumar) cadenas de texto.

```
var x = 10;  
x += 5;  
document.getElementById("demo").innerHTML = x;
```

El de **módulo** divide el valor de una expresión por el valor de otra y devuelve **el resto**.

Ejpl.:

```
var x = 5;  
var y = 2;  
var modulo = x % y; // El resultado será 1
```

El de **incremento** aumenta el valor de la variable en una unidad.

Ejpl.:

```
var numero = 2;  
++numero;  
document.write (numero); // El resultado será 3
```

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.

Ejpl.:

```
var numero = 2;  
document.write (numero++); // El resultado sigue siendo 2
```

```
document.write("<br/>" + numero); // El resultado es 3
```

El de **decremento** disminuye el valor de la variable en una unidad.

Prioridad del operador

La precedencia del operador describe el orden en el que se realizan las operaciones en una expresión aritmética.

Ejemplo

```
let x = 100 + 50 * 3;
```

¿El resultado del ejemplo anterior es igual a $150 * 3$, o es igual a $100 + 150$?

¿Se hace primero la suma o la multiplicación?

La multiplicación ($*$) y la división ($/$) tienen mayor precedencia que la suma ($+$) y la resta ($-$).

Y la precedencia se puede cambiar usando paréntesis:

Ejemplo

```
let x = (100 + 50) * 3;
```

Cuando se utilizan paréntesis, las operaciones dentro de los paréntesis se calculan primero.

Cuando muchas operaciones tienen la misma precedencia (como la suma y la resta), se calculan de izquierda a derecha:

Ejemplo

```
let x = 100 + 50 - 3;
```

Operadores relacionales

Estos operadores, también llamados condicionales o de comparación, nos sirven para hacer comparaciones entre valores o variables y determinar si son iguales o diferentes. El resultado de estas operaciones siempre nos dará un valor booleano (true o false).

==	Igual a (independientemente del tipo)
===	Igual valor e igual tipo
>	Mayor que
>=	Mayor o igual que
<	Menor que
<=	Menor o igual que
!=	Distinto (independientemente del tipo)

!== Distinto valor o distinto tipo

Probemos con algunos ejemplos:

```
var x = 5;
document.write(x == 5); // El resultado deberá ser true
document.write("<br/>");
document.write(x === "5"); // El resultado deberá ser false
document.write("<br/>");
document.write(x >= 4); // El resultado deberá ser true
document.write("<br/>");
document.write(x != 5); // El resultado deberá ser false
```

Hay que tener cuidado de no confundir "**==**", que sirve para una comparación de igualdad, con el operador de asignación "**=**", que utilizamos para dar valores a las variables.

Operadores de asignación

Como ya hemos comentado sirven para asignar valores a variables.

- =** Asigna un valor a una variable
- +=** Suma los valores y asigna el resultado a la variable.
- =** Resta los valores y asigna el resultado a la variable.
- *=** Multiplica los valores y asigna el resultado a la variable.
- /=** Divide los valores y asigna el resultado a la variable.
- %=** Divide los valores y asigna a la variable el resto.
- **=** Potencia un valor a otro y lo asigna a la variable.

Veamos unos ejemplos.

```
var x = 8;
document.write(x += 2); //El resultado deberá ser 10
document.write("<br/>");
document.write(x *= 2); //El resultado deberá ser 20
document.write("<br/>");
document.write(x %= 2); //El resultado deberá ser 0
```

A parte de los operadores aritméticos, relacionales y de asignación, tenemos los **operadores lógicos**, pero éstos los veremos un poco más adelante.

En Javascript podemos utilizar espacios, tabulaciones y saltos de línea libremente. Estos elementos nos deben ayudar a crear un código claro y legible.

Operadores de cadenas

El operador+ también se puede usar para agregar (concatenar) cadenas.

Ejemplos de asignación con cadenas.

```
<!DOCTYPE html>
<html>
<body>
  <h2>JavaScript Operators</h2>
  <p>The + operator concatenates (adds) strings.</p>
  <p id="demo"></p>
  <script>
    let text1 = "John";
    let text2 = "Doe";
    let text3 = text1 + " " + text2;
    document.getElementById("demo").innerHTML = text3;
  </script>
</body>
</html>
```

El +=operador de asignación también se puede usar para agregar (concatenar) cadenas:

```
<!DOCTYPE html>
<html>
<body>
  <h2>JavaScript Operators</h2>
  <p>The assignment operator += can concatenate strings.</p>
  <p id="demo"></p>
  <script>
    let text1 = "What a very ";
    text1 += "nice day";
    document.getElementById("demo").innerHTML = text1;
  </script>
</body>
</html>
```

Comparando cadenas

Para comparar dos cadenas, utilice operadores de comparación como >, >=, <, <=y ==operadores.

Los operadores de comparación comparan cadenas en función de los valores numéricos de los caracteres. Y puede devolver el orden de las cadenas que es diferente al que se usa en los diccionarios. Por ejemplo:

```
let result = 'a' < 'b';
console.log(result); // true
```

Sin embargo:

```
let result = 'a' < 'B';
console.log(result); // false
```

Operadores lógicos

Los operadores lógicos servirán para tomar decisiones.

Un **operador lógico** realiza una comparación entre dos valores y da como resultado un valor booleano, es decir, solo puede devolver dos respuestas, o **"true"** o **"false"**. Pero conozcamos primero qué valores devuelven **"false"** y son: el número 0, un string vacío "", el valor null, el valor undefined y NaN.

Operador NOT "!"

Los operandos serán evaluados como valores booleanos y los invertirá, por ejemplo si un valor es "true" después del operador **"!"** será "false".

```
var a = true;
var resultado = !a; // El resultado será false
document.write(resultado);
```

!El operador lógico funciona en base a las siguientes reglas:

- Si a es undefined, el resultado es true.
- Si a es null, el resultado es true.
- Si a es un número distinto de 0, el resultado es false.
- Si a es NaN, el resultado es true.
- Si a es un objeto, el resultado es falso.
- Si a es una cadena vacía, el resultado es verdadero. En el caso de que a sea una cadena no vacía, el resultado es false
- Si los dos operandos son "true" devolverá el valor "true" si no devolverá "false".

Doble negación (!!)

A veces, puede ver la doble negación (!!) en el código. El !! usa el operador lógico NOT (!) dos veces para convertir un valor a su valor booleano real.

Operador AND "&&"

JavaScript usa el doble ampersand (&&) para representar el operador lógico AND. Este operador nos servirá para comprobar varias condiciones en el caso que necesitemos **que se cumplan todas**.

La siguiente expresión utiliza el && operador: `let result = a && b;`

El && operador lleva lo siguiente:

- Evalúa los valores de izquierda a derecha.
- Para cada valor, lo convierte en un valor booleano. Si el resultado es false, se detiene y devuelve el valor original.
- Si todos los valores son valores reales, devuelve el último valor.

a	b	a & & b
verdadero	verdadero	verdadero
verdadero	falso	falso
falso	verdadero	falso
falso	falso	falso

Operador OR “||”

JavaScript usa la tubería doble || para representar el operador lógico OR. Puede aplicar el || operador a dos valores de cualquier tipo:

```
let result = a || b;
```

Si al menos uno de los dos operandos es “true” devolverá el valor “true”.

También nos sirve para comprobar varias condiciones pero en el caso que nos sea suficiente **que se cumpla al menos una**.

a	b	un b
verdadero	verdadero	verdadero
verdadero	falso	verdadero
falso	verdadero	verdadero
falso	falso	falso

El operador || también está cortocircuitado. Significa que si el primer valor se evalúa como true, el operador ya no evalúa el segundo.

La cadena de operadores ||

```
let result = value1 || value2 || value3;
```

El || operador hace lo siguiente:

- Evalúa los valores de izquierda a derecha.
- Para cada valor, lo convierte en un valor booleano. Si el resultado de la conversión es true, se detiene y devuelve el valor.
- Si todos los valores se evaluaron como false, devuelve el último valor.

Precedencia de operadores lógicos

La precedencia del operador lógico es en el siguiente orden de mayor a menor:

- NO lógico (!)
- AND lógico (&&)
- OR lógico (||)

Estructuras de control de flujo

En el presente apartado vamos a conocer las **estructuras de control de flujo**, formadas por sentencias condicionales y bucles que permiten al programa tomar decisiones ante diferentes situaciones.

- Por defecto Javascript ejecuta las órdenes que le damos, de forma secuencial, es decir, se ejecutan las sentencias una detrás de otra en el orden que han sido escritas. Pero en ocasiones necesitamos que algunas instrucciones se ejecuten sólo si se cumplen determinadas condiciones, por ejemplo, si un usuario no ha rellenado un campo de un formulario, que aparezca un aviso invitándole a hacerlo. Para estos casos tenemos las denominadas **estructuras de control de flujo**.

Existen dos tipos de estructuras:

Las condicionales, que permiten o no, que se ejecuten instrucciones dependiendo las condiciones.

Los bucles, que ejecutan instrucciones repetidamente mientras se cumplan las condiciones.

Condicionales

La declaración "if"

Sirve para ejecutar un bloque de código en caso de que una condición se cumpla.

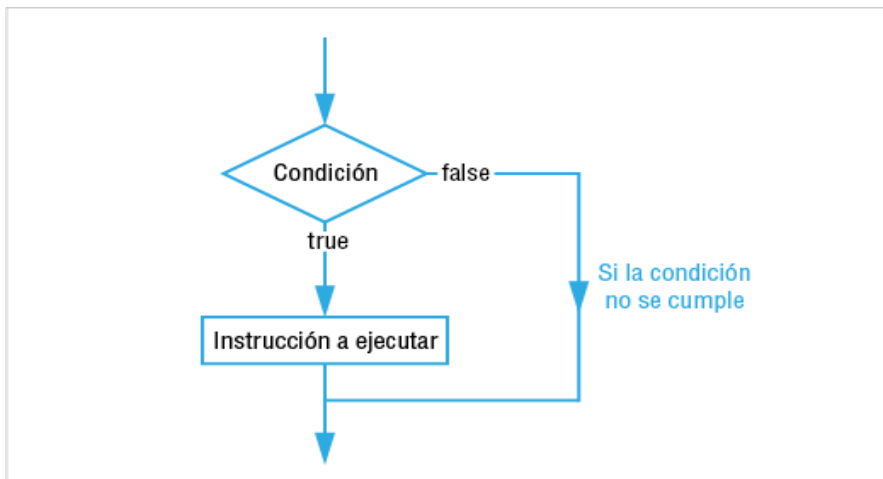


Diagrama Flujo condicional simple

Como vemos en el diagrama de arriba, cuando el programa llega a la condición existen dos opciones: que la respuesta sea **"true"** y se ejecute la instrucción, o que la respuesta sea **"false"** porque la condición no se cumple, y se continúe con el resto del código sin que se ejecute la instrucción.

Lo que en pseudocódigo viene a ser

```
if (condición){
    código a ejecutar si la condición
    se cumple;
}
```

Practiquemos con un ejemplo

```
var x = 5;
if (x == 5){
    document.write("x es igual a 5");
}
```

Como podemos comprobar en el ejemplo de arriba, al cumplirse la condición, **"si (x es igual a 5)"**, se ejecuta la instrucción que está entre llaves **{. .}**.

La declaración «else»

Sirve para complementar a "if", especificando un código a ejecutar en el caso de que no se cumpla la condición.

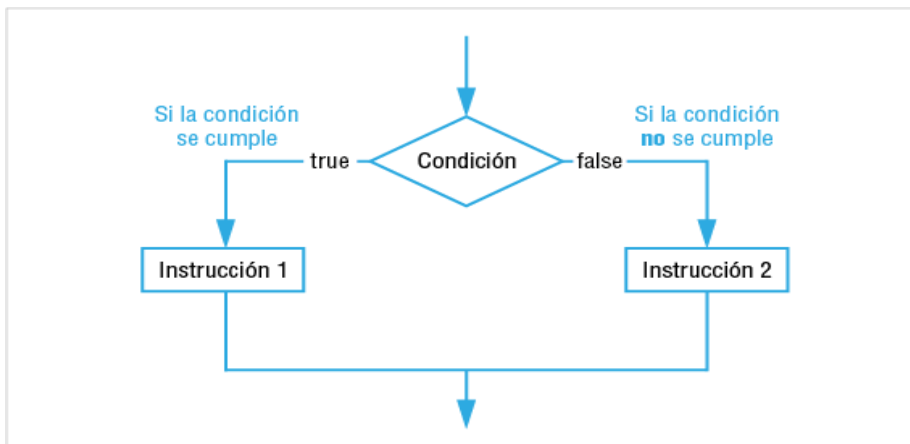


Diagrama Flujo condicional simple

Como vemos en el diagrama de arriba, cuando el programa llega a la condición existen dos opciones: que la respuesta sea **"true"** y se ejecute la **instrucción 1**, o que la respuesta sea **"false"** y se ejecute la **instrucción 2**. Dicho de otra manera, si la condición es verdadera ejecutar la **instrucción 1**, **si no**, ejecutar la **instrucción 2**.

Lo que en pseudocódigo sería

```
if (condición){
    código a ejecutar si la condición
    se cumple;
} else {
    código a ejecutar si la condición
    NO se cumple;
}
```


Practiquemos con un ejemplo

```
var x = 6;

if (x == 5){
  document.write("x es igual a 5");
} else {
  document.write("x NO es igual a 5");
}
```

En el código de arriba podemos comprobar cómo al no cumplirse la condición (**x es igual a 5**), se ejecuta el código entre llaves **{. .}** que está después de **"else"**.

HACER: Ejemplo altausuario.html.

Operador condicional (ternario)

El operador condicional (ternario) es el único operador de JavaScript que toma tres operandos: una condición seguida de un signo de interrogación (?), luego una expresión para ejecutar si la condición es verdadera seguida de dos puntos (:) y finalmente la expresión para ejecutar si la condición es falsa. Además de false, posibles expresiones falsas son: null, NaN, 0, la cadena vacía ("") y undefined.

Este operador se usa con frecuencia como una alternativa a una declaración if...else.

```
condition ? exprIfTrue : exprIfFalse
```

Ejemplo:

```
// Si el valor pasado por parámetro es mayor que 100 entonces escribes "Super",
// en otro caso , escribes "Normal";
function decide (dimension) {
  if (dimension > 100) {
    console.log("Super");
  } else {
    console.log("Normal");
  }
}
```

Podemos reducir la función utilizando el operador condicional:

```
function decide2 (dimension) {
  let test_mayor_100 = dimension > 100;
  let decision = test_mayor_100 ? "Super" : "Normal";
  return decision;
}

function decide3 (dimension) {
  let test_mayor_100 = dimension > 100;
  return test_mayor_100 ? "Super" : "Normal";
}
```

Solo nos falta mostrar por consola el valor devuelto:

```
function decide4 (dimension) {  
    console.log(dimension > 100 ? "Super" : "Normal");  
}
```

Condiciones complejas con “if”

Utilizando “&&” y “||” podemos comprobar dos o más condiciones en el mismo “if”.

Hagamos una prueba con “&&”.

```
var num1 = 5;  
var num2 = 10;  
var num3 = 20;  
var num4 = 35;  
if ( (num1 > num2) && (num3 < num4) ){  
    document.write("Las dos condiciones se cumplen");  
} else {  
    document.write("Al menos una de las condiciones NO se cumple");  
}
```

En el ejemplo de arriba con “&&” deberían haber sido ciertas las dos condiciones para que el **if** se cumpliese.

Probemos ahora con “||”

```
var num1 = 5;  
var num2 = 10;  
var num3 = 20;  
var num4 = 35;  
if ( (num1 > num2) || (num3 < num4) ){  
    document.write("Al menos una condición se cumple");  
} else {  
    document.write("No se cumple ninguna condición");  
}
```

En este último ejemplo con “||” ha bastado que una condición sea cierta para que el **if** se cumpliese.

Declaración “If else” anidada

Se trata de meter **un “if” dentro de otro**, de tal manera que si la primera condición se cumple pasará a comprobar el segundo “if”.

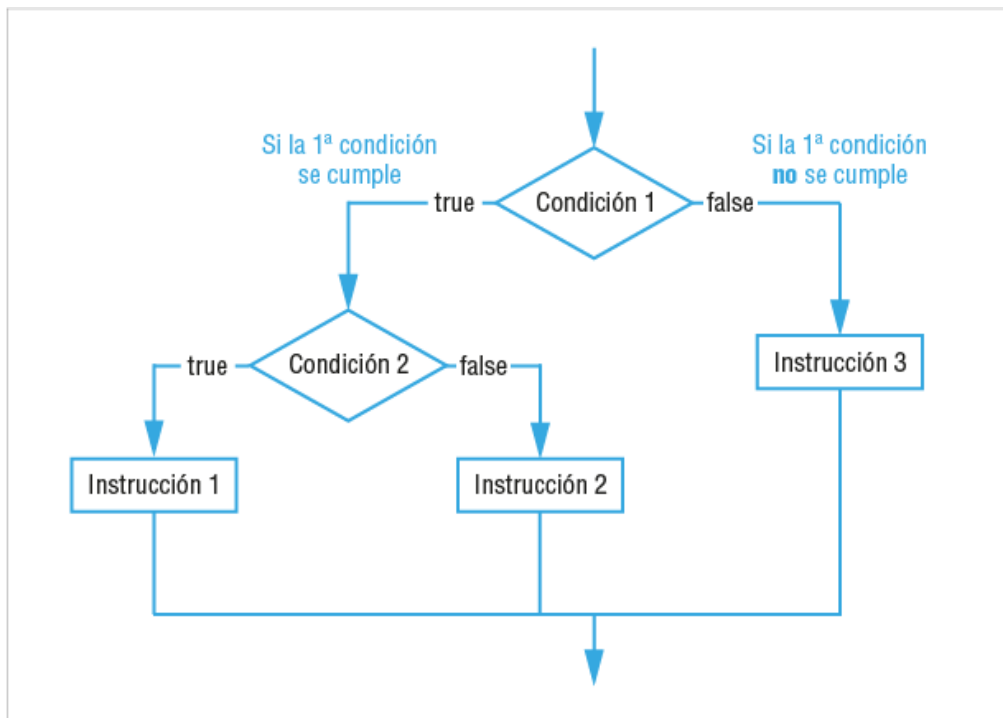


Diagrama Flujo Condicional Compleja

Ejemplo:

```

var distancia = 100;
if ( distancia > 10 ){
    if ( distancia < 40 ){
        document.write("est&aacute; cerca");
    } else {
        document.write("est&aacute; lejos");
    }
} else {
    document.write("est&aacute; muy cerca");
}

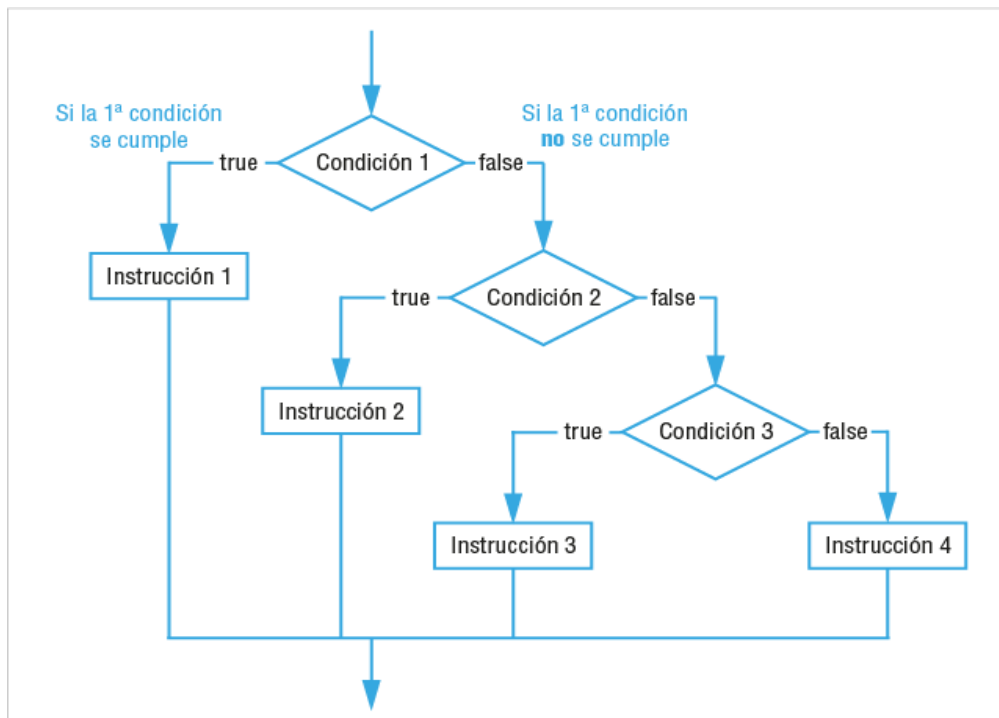
```

En este caso al ser cierta la primera condición ($\text{distancia} > 10$), se pasará a leer el bloque que contiene el primer "if" donde se encuentra el segundo "if". Al no cumplirse la condición del segundo "if" ($\text{distancia} < 40$) se ejecutará el bloque que contiene el "else" del segundo "if".

Declaración «else if»

En este apartado trataremos las dos últimas estructuras condicionales que nos faltan, **"else if"** y **"switch"**.

Otra forma de comprobar múltiples condiciones es utilizar varias sentencias **"if else" encadenadas** (en cascada), lo que se denomina "else if".



Flujo Condicional Múltiple (con if anidados)

Como se aprecia en el gráfico, primero nos encontramos con una condición si ésta no se cumple nos encontramos con otra condición, si esta última tampoco se cumple pasaremos a otra nueva condición, y así sucesivamente hasta que una condición se cumpla y se ejecute el bloque de instrucciones que le corresponda.

Sintaxis:

```

if (condición){
    código a ejecutar si la condición se cumple;
} else if (condición 2){
    código a ejecutar si la condición 2 se cumple;
} else if (condición 3){
    código a ejecutar si la condición 3 se cumple;
} else {
    código a ejecutar si la condición 3 tampoco se cumple;
}
  
```

Veámoslo con código.

```

var nota = 8;
if (nota <= 2){
    document.write("Muy deficiente");
} else if ((nota >= 3) && (nota < 5)){
    document.write("Insuficiente");
} else if ((nota >= 5) && (nota <= 6)){
    document.write("Suficiente");
} else if ((nota >= 7) && (nota <= 8)){
    document.write("Notable");
}
  
```

```

} else if ((nota > 8)){
    document.write("Sobresaliente");
} else {
    document.write("Debe hacer el examen");
}

```

Si ejecutamos el código anterior, se irán comprobando una a una todas las condiciones hasta encontrar una que pueda cumplir el valor de la variable "nota". En este caso que el valor es 8, se detendrá en la condición ((nota >= 7) && (nota <= 8)) mostrando en pantalla el texto "Notable".

Si bien esta estructura es totalmente correcta, para este tipo de situaciones se suele utilizar "switch", ya que es más eficaz y legible. Pasemos a verlo.

Múltiples '?'

Una secuencia de operadores de signos de interrogación ? puede devolver un valor que depende de más de una condición.

Por ejemplo:

```

let age = prompt('age?', 18);

let message = (age < 3) ? 'Hi, baby!' :
    (age < 18) ? 'Hello!' :
    (age < 100) ? 'Greetings!' :
    'What an unusual age!';

alert( message );

```

Puede ser difícil al principio entender lo que está pasando. Pero después de una mirada más cercana, podemos ver que es solo una secuencia ordinaria de pruebas:

1. El primer signo de interrogación comprueba si `age < 3`.
2. Si es verdadero, devuelve 'Hi, baby!'. De lo contrario, continúa con la expresión después de los dos puntos ":", comprobando `age < 18`.
3. Si eso es cierto, devuelve 'Hello!'. De lo contrario, continúa con la expresión después de los siguientes dos puntos ":", verificando `age < 100`.
4. Si eso es cierto, devuelve 'Greetings!'. De lo contrario, continúa con la expresión después de los últimos dos puntos ":", devolviendo 'What an unusual age!'.

Así es como se ve esto usando `if..else`:

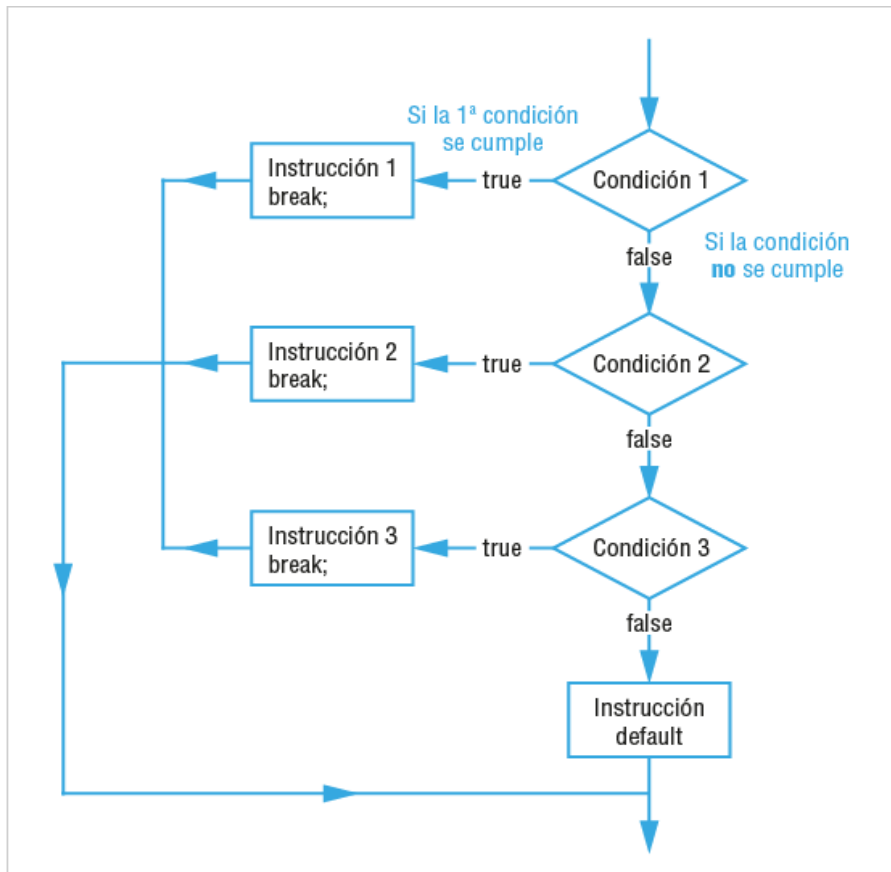
```

if (age < 3) {
    message = 'Hi, baby!';
} else if (age < 18) {
    message = 'Hello!';
} else if (age < 100) {
    message = 'Greetings!';
} else {
    message = 'What an unusual age!';
}

```

Declaración "switch"

Con "switch" podemos también evaluar condiciones múltiples, pero su estructura es ligeramente diferente.



Flujo Condicional Múltiple (con switch)

Se van comprobando diferentes condiciones una a una, si hay coincidencia se ejecuta la declaración que aparece a la izquierda y se sale del "switch", en caso contrario se pasa a comprobar la siguiente condición y así sucesivamente. En caso de que ninguna condición se cumpla se ejecutará la instrucción por defecto que aparece al final.

Esta sería la sintaxis

```
switch (expresión a evaluar){  
    case valor1 :  
        instrucción a ejecutar;  
        break;  
    case valor2 :  
        instrucción a ejecutar;  
        break;  
    case valor3 :  
        instrucción a ejecutar;  
        break;  
    default :  
        instrucción si no hay ninguna coincidencia;
```

La declaración comienza con la palabra clave **"switch"**, a continuación entre paréntesis se coloca la **expresión a evaluar**, es decir, el valor que tiene que coincidir con alguno de los valores que aparecen en los case.

En la siguiente línea a continuación de la palabra clave **"case"**, viene **el valor que se va a comparar con la expresión del comienzo**. En el caso que haya coincidencia se ejecutará la instrucción que aparece debajo y seguidamente con **"break"** se sale de la declaración "Switch", en caso contrario se pasa al siguiente "case" y así sucesivamente hasta encontrar una coincidencia.

En el caso de no haber encontrado ninguna coincidencia al llegar a **"default"** se ejecutará la instrucción que este contiene. Utilizar "default" es opcional.

Veamos un ejemplo basándonos en el caso propuesto anteriormente para "else if".

```
var nota = 4;

switch (nota) {
  case 0 :
  case 1 :
  case 2 :
    document.write("Muy deficiente");
    break;
  case 3 :
  case 4 :
    document.write("Insuficiente");
    break;
  case 5 :
  case 6 :
    document.write("Suficiente");
    break;
  case 7 :
  case 8 :
    document.write("Notable");
    break;
  case 9 :
  case 10 :
    document.write("Sobresaliente");
    break;
  default :
    document.write("Debe hacer el examen");
}
```

En este código se irán comparando los valores de los diferentes "case" con "nota" hasta encontrar una coincidencia. En este caso la coincidencia llegará en el "case" con valor 4, que hará que se muestre en pantalla el texto "Insuficiente".

En el ejemplo vemos cómo varios case pueden compartir una misma instrucción a ejecutar.

Ejemplo:

```
// colores : "blanco" : 10, "verde": 15 , "azul" : 20;

function priceByColor (color) {
  var price;
  switch(color.toLowerCase()) {
    case 'blanco':
      price = 10;
      break;
    case 'verde':
      price = 15;
      break;

    case 'azul':
      price = 20;
      break;
    default:
      price = 0;
  }

  return price;
}
```

Hacemos una versión simplificada utilizando una estructura de datos:

```
function priceByColor2 (color) {
  let pricesByColor = {
    'blanco' : 10,
    'verde' : 15,
    'azul' : 20
  };

  return pricesByColor[color.toLowerCase()];
}
```

Iteraciones

- Bucles
 - While / do while
 - For
 - For
 - forEach
 - for..of
 - for..in
- Map / Reduce / Filter / fill / repeat

Bucle "while"

En este capítulo vamos a ver **los bucles**, estructuras que nos servirán para repetir un conjunto de sentencias.

Básicamente un bucle es una estructura que ejecuta un grupo de instrucciones repetidamente mientras se cumple una condición. Cuando esa condición no se cumple se sale de la estructura para continuar con el flujo normal.

Javascript nos proporciona varios tipos de bucle; **«while»**, **«do while»**, **«for»**. También tenemos «for in» pero éste lo veremos más adelante.

Un bucle «while» repite un bloque de código mientras una condición sea verdadera.

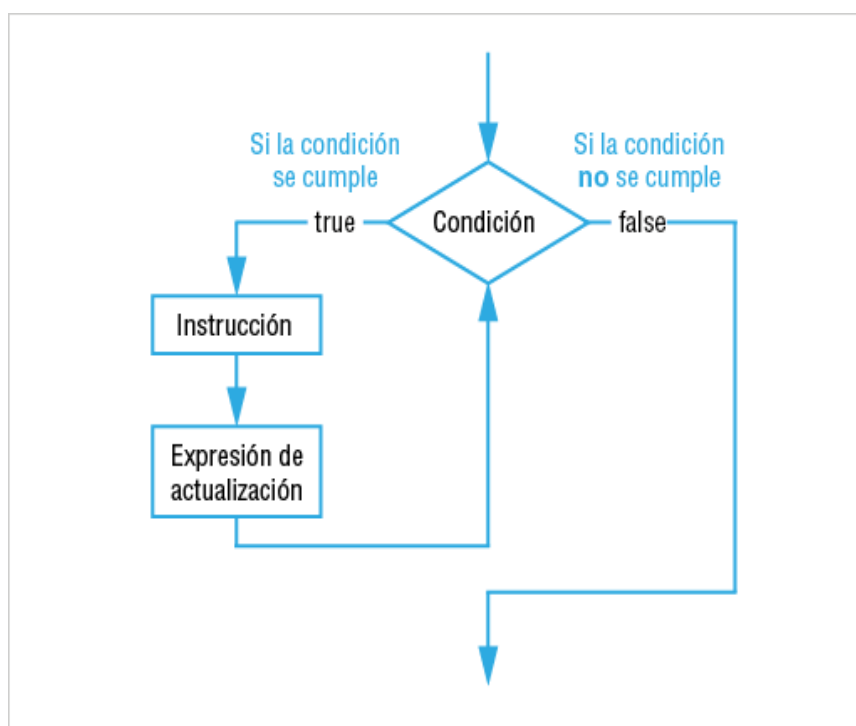


Diagrama Flujo bucle while

Sigamos el diagrama de arriba. Primero nos encontramos con la condición, en el caso de que nos devuelva «true» (la condición se cumple) se ejecuta 1º la instrucción y en segundo lugar la expresión de actualización. Finalmente se vuelve a comprobar la condición.

La **expresión de actualización** nos sirve para variar en cada repetición el valor de la variable, de tal manera que en algún momento la condición se deje de cumplir y devuelva false, de lo contrario estaríamos creando un bucle infinito.

Cuando la condición no se cumpla, se abandonará el bucle y continuará el flujo normal.

Veamos la sintaxis.

```
while (expresión a evaluar){  
    instrucción a ejecutar si la condición se cumple;  
    actualización de la variable;  
}
```

Sintaxis while

Como siempre la mejor manera de entenderlo es con un **ejemplo**.

```
var vuelta = 1; // Inicializamos la variable con el valor "1"  
document.write("Comienza el bucle. ")  
while (vuelta<11) {  
    document.write("Esta es la vuelta: " + vuelta + "<br/>");  
    vuelta = ++vuelta; //Actualizamos el valor de vuelta incrementándolo  
    en 1  
}  
  
document.write("Hemos salido del bucle");
```

En el código de arriba estamos diciendo:

- **Mientras** (while) la variable «vuelta» tenga un valor menor de «1», escribe en pantalla («Esta es la vuelta: » + vuelta + «
»).
- A continuación **actualiza** el valor de «vuelta» incrementándolo en 1, así en la siguiente vuelta su valor será 2.
- Este bucle será repetido hasta que el valor de «vuelta» llegue a «11», en ese momento dejará de ejecutarse el bucle y continuará con la siguiente instrucción que escribirá en pantalla «Hemos salido del bucle».

Bucle “do while”

El bucle «do while» es muy parecido a «while» pero su característica es que **ejecutará el código al menos una vez**. Recordemos que «while» primero evalúa la condición y si esta no se cumple no ejecuta el bloque de código que contiene. Con «do while» primero se ejecuta el bloque de instrucciones y luego evalúa la condición para permitir que se vuelva a ejecutar o no.

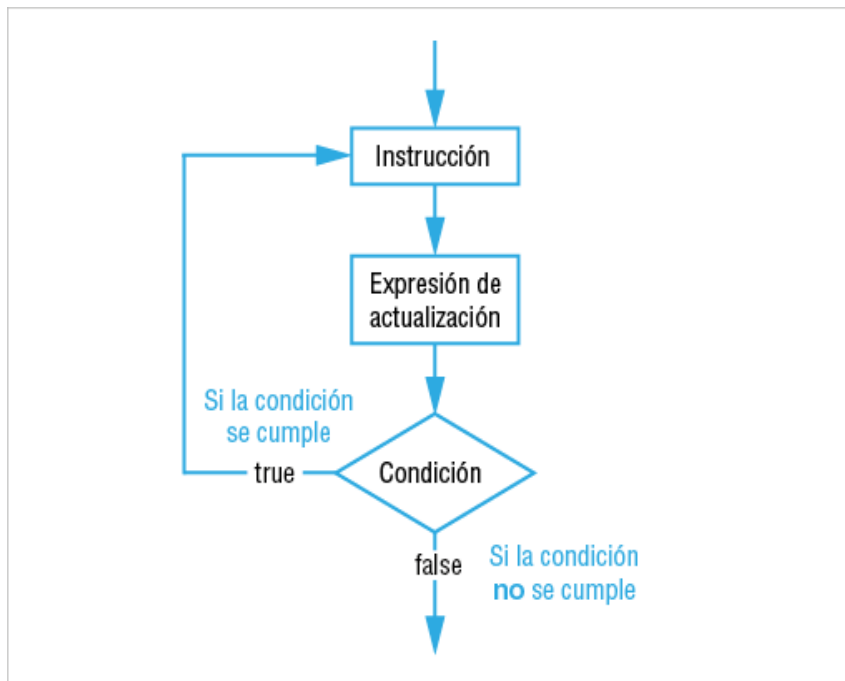


Diagrama Flujo bucle do while

Sigamos el diagrama de arriba. En primer lugar se ejecuta la instrucción y seguidamente se actualiza el valor de la variable. Cuando se llega a la condición, si ésta se cumple, se volverá al comienzo para ejecutar de nuevo el bucle, y si no se cumple se saldrá del bucle.

Vamos a ver la sintaxis.

```
do {  
    instrucción a ejecutar;  
    actualización de la variable;  
} while (expresión a evaluar);
```

Sintaxis do while

Clarifiquémoslo con un **ejemplo**.

```
var vuelta = 1; // Inicializamos la variable con el valor "1"  
  
document.write("Comienza el bucle. ")  
  
do {  
    document.write("Esta es la vuelta: " + vuelta + "<br/>");  
    vuelta = ++vuelta; //Actualizamos el valor de vuelta incrementándolo  
en 1  
} while (vuelta <11); // Mientras ésta expresión devuelva true se repetirá  
el bucle.  
  
document.write("Hemos salido del bucle");
```

Con este código le estamos ordenando:

- **Ejecuta** (do) lo siguiente. Escribe en el documento («Esta es la vuelta: » + vuelta + «
») y actualiza el valor de “vuelta” incrementándolo en 1.
- Repite esto **mientras** (while) el valor de “vuelta” sea menor de 11.

Como podemos ver es una estructura sencilla y similar a “while”, la diferencia radica en que ésta la usaremos cuando necesitemos que la instrucción que contiene el bucle se ejecute al menos una vez.

A las repeticiones de un bucle se les llama **iteraciones**.

Bucle “for”

La estructura “for” **utiliza una variable como contador** para limitar las iteraciones del bucle. El bloque que contiene el “for” se ejecutará hasta que el contador llegue a un determinado punto. Aunque pueda parecer un poco confuso de primeras, en seguida lo veremos con más claridad.

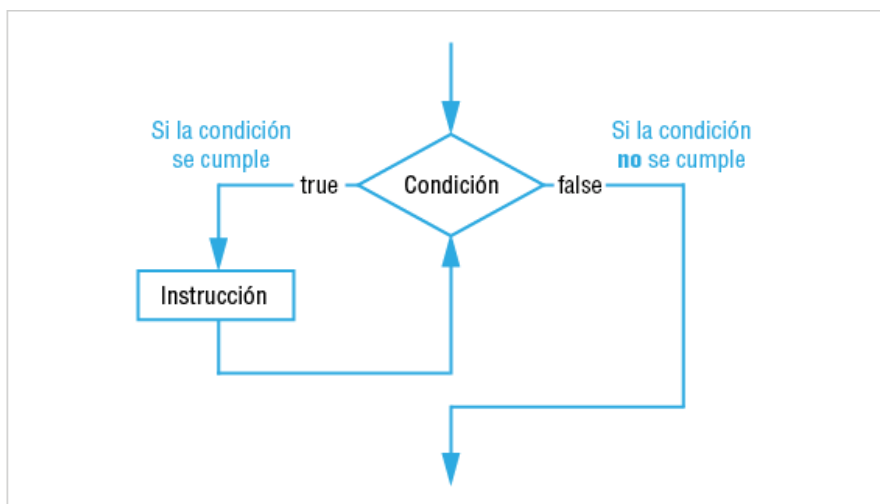


Diagrama flujo bucle for

Conozcamos los tres parámetros que necesitamos para que nuestro “for” funcione como un contador.



Sintaxis parámetros for

– Expresión de inicialización:

Aquí inicializamos una variable dándole el valor inicial con el cual comenzará el contador. Ejemplo. (contador = 0).

– Expresión de evaluación:

Aquí comprobaremos si la condición se cumple o no. Si se cumple se ejecutarán las instrucciones que contiene el bucle, y en caso contrario se saldrá del mismo. (contador < 10).

– Expresión de actualización:

Aquí incrementamos o decrementamos el valor de la variable declarada en la expresión de inicialización. De ese modo hacemos que en alguna de las vueltas la condición no se cumpla y salgamos del bucle.

Ahora veamos la sintaxis completa del "for".

```
for (inicialización; evaluación; actualización){  
    sentencia a ejecutar si la evaluación devuelve true;  
}
```

sintaxis for

Practiquemos con un **ejemplo**.

```
for (let i = 1; i < 11; i++) {  
    document.write("En esta vuelta \"i\" vale: " + i + "<br>");  
}
```

En el código de arriba le estamos ordenando:

- La variable «i» vale 1 (i = 1).
- Vas a repetir este bucle mientras «i» sea menor que 11 (i < 11).
- En cada vuelta del bucle vas a incrementar «i» en 1 (i++) y vas a escribir en el documento («En esta vuelta »i« vale: » + i + «
»).

En la expresión de inicialización hemos llamado «i» a la variable que utilizamos como contador.

RECORDATORIO:

Los bucles «while» y «for» comprueban la expresión de evaluación al comienzo del bucle. Por lo que si se evalúa como falsa no se ejecutará ni una sola vez.

En el caso de «do while», al evaluarse la expresión al final, como mínimo se ejecutará una vez.

Ruptura de bucle con «break»

Además de utilizarse en las estructuras «switch», como ya hemos visto, «break» también nos permite salir de un bucle antes de tiempo.

Vamos con un **ejemplo**.

```
for (let i = 1; i < 11; i++) {  
    document.write("En esta vuelta \"i\" vale: " + i + "<br>");  
    if (i == 6) {  
        break;  
    }  
}
```

En este caso, al introducir un condicional if que ejecuta un «break» cuando «i» sea igual a 6, al llegar a la iteración número 6 se sale del bucle.

Ruptura de bucle con «continue»

La declaración «continue» nos permite omitir **una** iteración de un bucle si se cumple una condición y proseguir el bucle en la iteración siguiente.

Veámoslo.

```
for (let i = 1; i < 11; i++) {  
    if (i == 6) {  
        continue;  
    }  
    document.write("En esta vuelta \"i\" vale: " + i + "<br>");  
}
```

En éste último código, en la vuelta que se cumple «i» es igual a 6, «continue» salta el código que se sitúa a continuación y pasa a la siguiente iteración.

Se denomina **«bloque de código»** al código que está entre llaves, «{ }». Las declaraciones «break» y «continue» son las únicas que pueden hacer saltar fuera de un bloque de código.

El bucle forEach

Este tipo de bucle fue una novedad que introdujo ECMAScript5. Pertenece a las funciones de la clase Array y nos permite iterar dentro de un array de una manera secuencial.

Ejemplo:

```
var miArray = [1, 2, 3, 4];  
miArray.forEach(function (elemento, index) {  
    console.log("El valor de la posición " + index + " es: " + elemento);  
});
```

```
// Devuelve lo siguiente
// El valor de la posición 0 es: 1
// El valor de la posición 1 es: 2
// El valor de la posición 2 es: 3
// El valor de la posición 3 es: 4
```

El bucle for...of

La sentencia for...of ejecuta un bloque de código para cada elemento de un objeto iterable, como lo son: String, Array

La herramienta básica para recorrer una colección es el bucle for...of:

```
const cats = ['Leopard', 'Serval', 'Jaguar', 'Tiger', 'Caracal', 'Lion'];
for (const cat of cats) {
  console.log(cat);
}
```

En este ejemplo, for (const cat of cats) dice:

1. Dada la colección cats, obtenga el primer elemento de la colección.
2. Asígnelo a la variable cat y luego ejecute el código entre llaves {}.
3. Obtenga el siguiente elemento y repita (2) hasta que haya llegado al final de la colección.

El bucle for...in

Herramienta para recorrer un bucle como en el bucle for...of, pero se utiliza para recorrer un objeto:

```
var obj = {cad1:'Pedro', cad2:'Jose'};
for (let k in obj) {
  console.log(obj[k]);
}
```

El operador coma

JavaScript usa una coma (,) para representar el operador de coma. Un operador de coma toma dos expresiones, las evalúa de izquierda a derecha y devuelve el valor de la expresión correcta.

Esta es la sintaxis del operador coma:

leftExpression, rightExpression

Por ejemplo:

```
let x = 10;  
let y = (x++, x + 1);  
console.log(x, y); //muestra 11 12
```

En este ejemplo, aumentamos el valor de x en uno (x++), sumamos uno a x (x+1) y asignamos x a y. Por lo tanto, x es 11, y y es 12.

Sin embargo, para que el código sea más explícito, puede usar dos declaraciones en lugar de una declaración con un operador de coma como este:

```
let x = 10;  
x++;  
let y = x + 1;  
console.log(x, y);
```

Este código es más explícito.

En la práctica, es posible que desee utilizar el operador de coma dentro de un bucle for para actualizar varias variables cada vez que se ejecuta.

El siguiente ejemplo usa el operador coma en un bucle for para mostrar una matriz de nueve elementos como una matriz de 3 filas y 3 columnas:

```
let board = [1, 2, 3, 4, 5, 6, 7, 8, 9];  
  
let s = '';  
for (let i = 0, j = 1; i < board.length; i++, j++) {  
  s += board[i] + ' ';  
  if (j % 3 == 0) {  
    console.log(s);  
    s = '';  
  }  
}
```

Mostrará por consola:

```
1 2 3  
4 5 6  
7 8 9
```


map() reduce() y filter()

JavaScript también tiene bucles más especializados para colecciones, y mencionaremos algunos de ellos aquí.

Puede usar **map()** para hacer algo con cada elemento de una colección y crear una nueva colección que contenga los elementos modificados.

Podemos simplemente devolver los elementos de una colección:

```
let list =[1,2,3,4,5,6]
list.map((el) => console.log(el))
```

O podemos por ejemplo devolver los elementos modificados en mayúsculas:

```
function toUpper(string) {
  return string.toUpperCase();
}

const cats = ['Leopard', 'Serval', 'Jaguar', 'Tiger', 'Caracal', 'Lion'];
const upperCats = cats.map(toUpper);
console.log(upperCats);
// [ "LEOPARD", "SERVAL", "JAGUAR", "TIGER", "CARACAL", "LION" ]
```

Aquí pasamos una función a `cats.map()`, y llamamos a la función `map()` una vez por cada elemento de la matriz, pasando el elemento. Luego agrega el valor de retorno de cada llamada de función a una nueva matriz y finalmente devuelve la nueva matriz. En este caso, la función que proporcionamos convierte el elemento a mayúsculas, por lo que la matriz resultante contiene todos nuestros gatos en mayúsculas:

```
[ "LEOPARD", "SERVAL", "JAGUAR", "TIGER", "CARACAL", "LION" ]
```

En el siguiente ejemplo, usamos el método de Array **map()** para duplicar cada valor en la matriz original:

```
const originals = [1, 2, 3];
const doubled = originals.map(item => item * 2);
console.log(doubled); // [2, 4, 6]
```

El método `map()` toma cada elemento de la matriz a su vez y lo pasa a la función dada. Luego toma el valor devuelto por esa función y lo agrega a una nueva matriz.

Entonces, en el ejemplo anterior, `item => item * 2`, la función de flecha es equivalente a:

```
function doubleItem(item) {
  return item * 2;
}
```

Recomendamos que se utilicen las funciones de flecha, ya que pueden hacer que el código sea más corto y más legible.

Puede usar **filter()** para probar cada elemento de una colección y crear una nueva colección que contenga solo elementos que coincidan:

```
function lCat(cat) {  
  return cat.startsWith('L');  
}  
const cats = ['Leopard', 'Serval', 'Jaguar', 'Tiger', 'Caracal', 'Lion'];  
const filtered = cats.filter(lCat);  
console.log(filtered);  
// [ "Leopard", "Lion" ]
```

Esto se parece mucho a map(), excepto que la función que pasamos devuelve un valor booleano : si devuelve true, entonces el elemento se incluye en la nueva matriz. Nuestra función prueba que el elemento comienza con la letra "L", por lo que el resultado es una matriz que contiene solo gatos cuyos nombres comienzan con "L":

```
[ "Leopard", "Lion" ]
```

Tenga en cuenta que map()y filter()se usan a menudo con expresiones de funciones , sobre las cuales aprenderemos en el módulo Funciones . Usando expresiones de función, podríamos reescribir el ejemplo anterior para que sea mucho más compacto:

```
const cats = ['Leopard', 'Serval', 'Jaguar', 'Tiger', 'Caracal', 'Lion'];  
const filtered = cats.filter((cat) => cat.startsWith('L'));  
console.log(filtered);  
// [ "Leopard", "Lion" ]
```

Ejemplo:

Función que muestre los elementos pares o impares según le indiquemos:

```
function oddOrEven (oddOk) {  
  //odd es par  
  return oddOk  
? function (el){  
    return (el % 2) == 0;  
: function (el){  
    return (el % 2) != 0;  
};  
}
```

Llamaremos a la función indicando como parámetro true (pares) o false (impares):

```
list.filter(oddOrEven(true))    //pares  
list.filter(oddOrEven(false))  //impares
```

Funciones

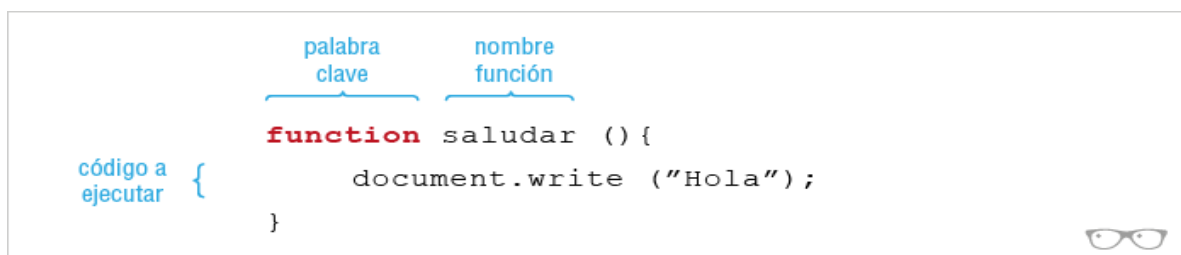
Vamos a ver cómo **las funciones** nos ayudan a reutilizar código para evitar tener que crearlo una y otra vez.

Introducción

Las funciones, no son más que un bloque de código creado para desarrollar una tarea en concreto. Lo útil de estas estructuras es que una vez creadas podemos llamarlas (invocarlas) cuando las necesitemos sin tener que volver a escribir todo el código.

Además de poder utilizarlas tantas veces como queramos, nos sirven para mantener el código bien estructurado pudiendo separar unas tareas de otras.

Sintaxis:



Sintaxis función

Practiquemos con un ejemplo similar.

```
function saludar () {  
    console.log("Hola");  
}  
saludar(); //Se llama a la función
```

En el código de arriba hemos comenzado con la palabra clave **«function»** y hemos escrito el nombre que queremos que tenga la función, «saludar».

A continuación hemos colocado dos paréntesis sin contenido **«()»**, ya veremos un poco más adelante para que se utilizan.

Para finalizar, hemos introducido entre las llaves **“{ }”** la tarea que queremos que ejecute. En concreto, con el método **«alert»** haremos que se muestre en pantalla una ventana de alerta con el texto «Hola».

De este modo ya tenemos una función preparada para poder usarla en cualquier momento. Para ello solo tenemos que **llamarla (invocarla)**, que es lo que hemos hecho en la última línea de código escribiendo su nombre y los dos paréntesis.

Nota de interés:

Las nomenclaturas de las funciones siguen las mismas reglas que las variables.

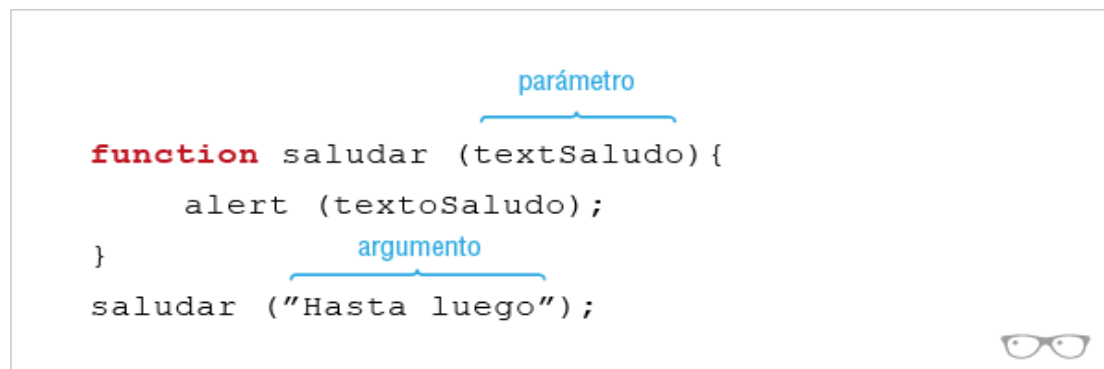
Una buena práctica para diferenciar fácilmente una variable de una función por el nombre, es utilizar verbos para las funciones, por ejemplo «saludar» en vez de «saludo».

Parámetros

Las funciones más sencillas como la del ejemplo anterior no necesitan que les proporcionemos información a la hora de llamarlas para que realicen su tarea. Pero a la mayoría de las funciones **deberemos facilitarles valores** para que puedan operar con ellos. Estos valores se los pasaremos a la función a través de los «parámetros».

En el ejercicio anterior hemos visto como al definir la función colocábamos dos paréntesis vacíos «()». Entre estos paréntesis es precisamente donde podemos colocar los parámetros.

```
function saludar (textoSaludo) {  
    alert (textoSaludo);  
}  
saludar ("Hasta luego");
```



Sintaxis función con parámetros

En el ejemplo de arriba hemos creado un parámetro llamado **«textoSaludo»**, hay que pensar en este parámetro como en una variable a la que posteriormente le vamos a dar un valor.

En el interior de la función hemos colocado un «alert» que mostrará el valor que le demos al parámetro «textoSaludo» cuando llamemos a la función.

En la última línea de código llamamos (invocamos) a la función saludar y **le pasamos un valor al parámetro**, concretamente el texto «Hasta luego». Con lo que mostrará un «alert» con dicho texto.

Nota de interés:

El término **parámetro** se utiliza al darles nombre cuando se define la función.

Los **argumentos** son los valores que se les da a los parámetros al llamar a la función.

Practiquemos con un pequeño ejercicio:

```
var miSuma;

function sumar (numero1, numero2) {
    miSuma = (numero1 + numero2);
    alert (miSuma);
}

sumar(25, 150); //Se llama a la función y mostrará 175
sumar(3, 20); //Se llama a la función y mostrará 23
sumar(2, 50); //Se llama a la función y mostrará 52
```

En el ejercicio, primero hemos declarado una variable que utilizamos dentro de la función, después **hemos definido la función con dos parámetros** llamados numero1 y numero2 separados por una coma.

Dentro de la función hemos asignado a la variable miSuma el valor de sumar numero1 y numero2, y a continuación hemos colocado un alert que mostrará el valor de «miSuma».

Finalmente **hemos llamado a la función pasándole dos argumentos** (25, 150), con lo que se deberá mostrar una alerta con la cifra «175». Y así con las sucesivas llamadas a la función.

Nota de interés:

Los parámetros no pueden tener el mismo nombre que las variables que utilicemos dentro de la función.

Podemos asignarle un **valor por defecto a los parámetros** y en caso de que no se le pase ningún valor en la llamada tomará los valores por defecto.

```
function saludar (name = "Mundo") {
    console.log("Hola " + name);
}

saludar(); //Se llama a la función mostrando Hola Mundo
saludar(Angelica); //Se llama a la función mostrando Hola Angelica
```

Declaración return

Las funciones no solo pueden recibir valores, a través de los parámetros, también pueden **devolvernos los valores** resultantes de las operaciones que ejecuten.

Para recuperar el resultado del bloque de código de una función, Javascript nos proporciona la palabra clave «return».

Veamos un ejemplo:

```
function sumar (numero1, numero2) {
    return (numero1 + numero2);
}
valorRetornado = sumar(25,150); //La función retornará un valor al ser
llamada y lo guardamos en "valorRetornado"

alert (valorRetornado);
```

Dentro de la función hemos escrito la palabra clave **«return»** seguida de la operación de la cual queremos obtener el resultado.

Posteriormente utilizamos la variable **“valorRetornado”** para almacenar el valor que retornará la función «sumar(25, 100)» al ser invocada.

La prueba de que “return” nos ha dado el valor resultante de la operación la podemos ver en el texto mostrado por el “alert”.

Nota de interés:

En el momento que se ejecuta la declaración «return» se sale automáticamente de la función. El resto de código que hubiera debajo del “return” hasta el final de la función se omitiría.

Ejemplo

Función que calcule el producto de 2 números y devuelva su resultado.

```
<!DOCTYPE html>
<html>
<body>
    <h2>JavaScript Functions</h2>
    <p>Esta función llama calcula el producto de 2 números y devuelve el
resultado:</p>
    <p id="demo"></p>
    <script>
        function myFunction(p1, p2) {
            return p1 * p2;
        }
        document.getElementById("demo").innerHTML = myFunction(4, 3);
    </script>
</body>
</html>
```

Ejemplo

Función que calcule de grados Fahrenheit y los pase a Celsius

```
<!DOCTYPE html>
<html>
<body>
    <h2>JavaScript Functions</h2>
    <p>Este ejemplo convierte grados Fahrenheit a Celsius:</p>
    <p id="demo"></p>
    <script>
        function toCelsius(f) {
            return (5/9) * (f-32);
        }
        document.getElementById("demo").innerHTML = toCelsius(77);
    </script>
</body>
</html>
```

Alcance de las variables

En este apartado vamos a tratar el alcance de las variables, y cómo esta característica nos condiciona a la hora de trabajar con funciones.

El «alcance» o también llamado «ámbito» de las variables (en inglés «scope»), determina el acceso que vamos a tener a las variables dependiendo en qué lugar del código hayan sido definidas.

Las variables en Javascript pueden ser de dos tipos según su alcance:

Variables locales

Son las variables que se declaran **dentro de una función** con lo que tan solo se puede acceder a ellas desde la función donde ha sido creada. Para que esto se cumpla la variable debe ser definida con la palabra clave «var», de lo contrario será tomada como global.

```
function mensaje () {  
    var varLocal = "LA VARIABLE LOCAL";  
    document.write ("Desde la función puedo usar " + varLocal);  
}
```

```
mensaje();
```

```
document.write ("Desde fuera no puedo usar" + varLocal); //nos dará error  
(podemos comprobarlo desde la consola)
```

El tiempo de vida de una variable local en Javascript comienza en el momento que es declarada y termina cuando la función ha sido ejecutada.

Variables globales

Como su nombre indica son de alcance global, lo que significa que están disponibles para que accedamos a ellas desde cualquier parte del código.

Para crear una variable global simplemente tenemos que declararla **fuera de una función**, de ésta manera cualquier función podrá acceder a ella.

```
var varGlobal = "LA VARIABLE GLOBAL";
```

```
function mensaje () {  
    document.write ("Desde la función puedo usar " + varGlobal);  
}
```

```
mensaje();
```

```
document.write ("</br> Y por supuesto desde fuera también puedo usar " +  
varGlobal);
```

Son útiles cuando queremos que varias funciones compartan valores.

Recordar, que como hemos comentado anteriormente, si definimos una variable dentro de una función pero sin la palabra clave «var» (no la declaramos, solo le asignamos un valor) será tomada como global.

Nota:

Los parámetros son variables locales de una función.

En el caso de que tengamos una variable local definida con el mismo nombre que una variable global, la variable local prevalecerá sobre la global dentro de la función a la que pertenece.

Si definimos una variable global dentro de una función, con el mismo nombre que otra variable global que ya estaba definida, simplemente se actualizará el valor.

De todas formas es recomendable evitar la duplicidad de nombres para no tener problemas. También **se recomienda** definir variables locales cuando su uso se vaya a reducir al ámbito de una función específica, y utilizar variables globales cuando se necesite compartir variables entre diferentes funciones.

Otras características

El alcance de las variables nos ofrece una posibilidad muy interesante cuando trabajamos con las funciones, ya que **nos permite guardar datos privados** y trabajar con ellos dentro de las funciones y que no se pueda acceder a ellos desde fuera.

Así una variable local no sobrescribirá accidentalmente a otra variable definida con el mismo nombre en otro lugar del código, ni se crearán posibles errores por este tipo de duplicidades.

Hay que tener en cuenta que podemos estar trabajando con archivos externos de Javascript que hayamos cargado, y las variables globales que éstos contengan también serán globales para nuestro código, ya que compartirán el mismo espacio global. Por lo que deberemos no abusar de las variables globales para evitar duplicados.

Funciones anidadas

Nos sirve para organizar el código, así podemos dividir en varias funciones un proceso complejo y luego ejecutarlas desde una función.

Practiquemos con un ejemplo.

```
var precioProd1 = 100;
var precioProd2 = 100;

function calcularIva (precio) {
    return (precio*1.21);
}
```



```
function calcularIvaReducido (precio) {
    return (precio*1.10);
}

function calcularTotal (){
    return (calcularIva(precioProd1) + calcularIvaReducido(precioProd2));
//Estamos utilizando las funciones anteriores
}

document.write (calcularTotal());
```

Primero hemos creado dos funciones llamadas «calcularIva» y «calcularIvaReducido» las cuales calculan diferentes tipos de IVA.

A continuación hemos creado una función que suma precios añadiéndole el IVA correspondiente por medio de las funciones anteriores. De esta manera tenemos el código bien organizado y más fácil de entender y modificar en un futuro.

Es recomendable que cada función solo realice una tarea concreta.

Funciones recursivas de JavaScript

Una función recursiva es una función que se llama a sí misma hasta que no lo hace. Y esta técnica se llama recursividad.

Suponga que tiene una función llamada recurse(). es una función recursiva recurse() si se llama a sí misma dentro de su cuerpo, así:

```
function recurse() {
    if(condition) {
        // stop calling itself
        //...
    } else {
        recurse();
    }
}
```

Una función recursiva siempre tiene una condición para dejar de llamarse a sí misma. De lo contrario, se llamará a sí mismo indefinidamente.

Ejemplo:

Función que cuente hacia atrás desde un número específico hasta 1.

```
function countDown(fromNumber) {
    console.log (fromNumber);
}

countDown(3);
```

Funciones anónimas y funciones de flecha

Se llama función anónima a aquella que no tiene nombre.

```
function() {  
  alert('hello');  
}
```

No se puede acceder a una función anónima después de su creación inicial. Por lo tanto, suele necesitar asignarlo a una variable.

```
let show = function() {  
  console.log('Soy una función anónima');  
};  
show();
```

A menudo verá funciones anónimas cuando una función espera recibir otra función como parámetro. En este caso, el parámetro de la función a menudo se pasa como una función anónima y, a veces, es posible que desee pasarle argumentos, como este:

```
let person = {  
  firstName: 'John',  
  lastName: 'Doe'  
};  
  
(function () {  
  console.log(person.firstName + ' ' + person.lastName);  
})(person);
```

Funciones de flecha

Las funciones de flecha de ES6 brindan una forma alternativa de escribir una sintaxis más corta en comparación con la expresión de la función.

Por ejemplo, esta función:

```
let show = function () {  
  console.log('Soy una función anónima');  
};
```

se puede acortar usando la siguiente función de flecha:

```
let show = () => console.log('Soy una función anónima');
```

El siguiente ejemplo define una expresión de función que devuelve la suma de dos números:

```
let add = function (x, y) {  
  return x + y;  
};  
console.log(add(10, 20)); // 30
```

Si una función de flecha tiene dos o más parámetros, utilice la siguiente sintaxis:

`(p1, p2, ..., pn) => expression;`

La siguiente expresión:

=> expression

es equivalente a la siguiente expresión:

=> { return expression; }

El siguiente ejemplo es equivalente a la expresión `add()` de la función anterior, pero usa una función de flecha en su lugar:

```
let add = (x, y) => x + y;
console.log(add(10, 20)); // 30;
```

En este ejemplo, la función de flecha tiene una expresión `x + y`, por lo que devuelve el resultado de la expresión. Sin embargo, si usa la sintaxis de bloque, debe especificar la palabra clave `return`:

```
let add = (x, y) => { return x + y; };
```

Por ejemplo, para ordenar una matriz de números en orden descendente, utilice el `sort()` método del objeto de matriz de la siguiente manera:

```
let numbers = [4,2,6];
numbers.sort(function(a,b) {
    return b - a;
});
console.log(numbers); // [6,4,2]
```

El código es más conciso con la sintaxis de la función de flecha:

```
let numbers = [4,2,6];
numbers.sort((a,b) => b - a);
console.log(numbers); // [6,4,2]
```

Paso de parámetros por valor o por referencia

En JavaScript, todos los argumentos de función siempre se pasan por valor. Significa que JavaScript copia los valores de las variables en los argumentos de la función.

Los cambios que realice en los argumentos dentro de la función no reflejan las variables que pasan fuera de la función. En otras palabras, los cambios realizados en los argumentos no se reflejan fuera de la función.

```
function square(x) {
    x = x * x;
    return x;
}

let y = 10;
let result = square(y);

console.log(result); // 100
console.log(y); // 10 -- no cambia
```

Por este motivo es una buena práctica no trabajar dentro de la función sobre los argumentos recibidos sino asignar esos valores a variables y modificar el valor de las variables.

Paso por valor de los valores de referencia

Los valores de referencia también se pasan por valores. Internamente, el motor de JavaScript crea la referencia al objeto y hace que la variable haga referencia al mismo objeto.

Por ejemplo:

```
let person = {  
  name: 'John',  
  age: 25,  
};  
  
function increaseAge(obj) {  
  obj.age += 1;  
}  
  
increaseAge(person);  
console.log(person);
```

Parece que JavaScript pasa un objeto por referencia porque el cambio en el objeto se refleja fuera de la función. Sin embargo, éste no es el caso.

De hecho, cuando pasa un objeto a una función, está pasando la referencia de ese objeto, no el objeto real. Por lo tanto, la función puede modificar las propiedades del objeto a través de su referencia.

Sin embargo, no puede cambiar la referencia pasada a la función.

Introducción al DOM

El DOM (Modelo de Objetos del Documento) es un estándar del W3C (World Wide Web Consortium) que nos **permite el acceso y la modificación** de los diferentes elementos de nuestro documento HTML.

«El DOM es una interface que permite a programas y scripts acceder dinámicamente y actualizar el contenido, estructura y estilo de un documento.»

Cuando creamos un documento HTML, éste se compone de diferentes etiquetas que el navegador debe interpretar para poder mostrar los elementos visuales que queremos que el usuario vea en pantalla.

Al mismo tiempo que el navegador realiza la tarea de interpretar las etiquetas, **confecciona una estructura (el DOM)** en la que organiza jerárquicamente todos los elementos que conforman nuestra página web (<html>, <body>, <a>, . . .).

Esta estructura que se presenta como un diagrama de árbol está compuesta por nodos que pueden ser padres, hijos o hermanos de otros nodos. También existen diferentes tipos de nodo según su contenido.

En la representación del DOM que tenemos a continuación cada rectángulo es un nodo. Cada nodo del árbol es un objeto. Aunque existen 12 tipos de nodo, en el ejemplo hemos utilizado algunos de los más comunes.

Document: Es la raíz del árbol, el nodo de donde salen todos los demás.

Element: Los nodos que contienen las etiquetas HTML y de donde pueden salir otros nodos.

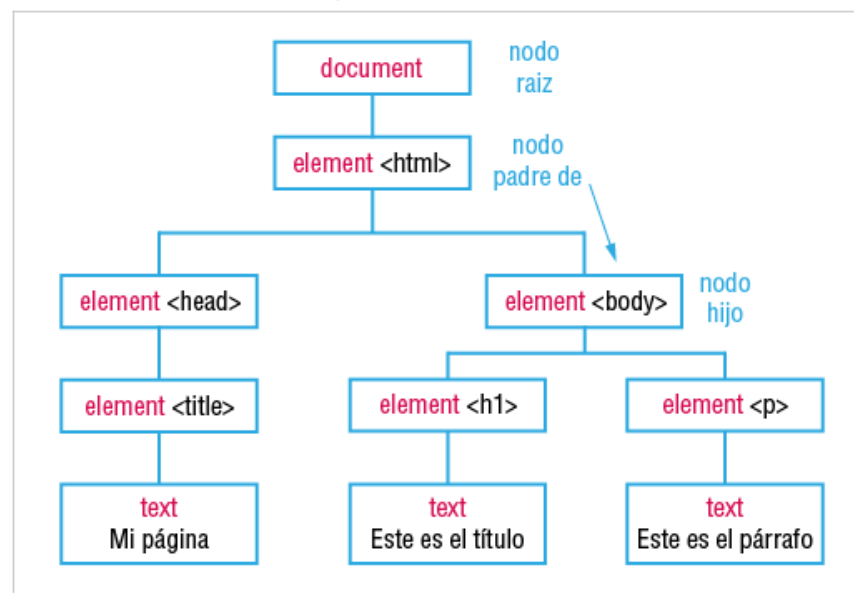
Attr: Representa a los atributos de las etiquetas con sus pares «atributo=valor».

Text: Representa el texto encerrado por una etiqueta.

Documento HTML

```
<html>
  <head>
    <title> Mi página </title>
  </head>
  <body>
    <h1> Este es el título </h1>
    <p> Este es el párrafo </p>
  </body>
</html>
```

Presentación DOM



Ejemplo DOM

El navegador construye el DOM automáticamente y a partir del momento en que el documento HTML se carga y la página web se muestra en pantalla, desde el código Javascript **vamos a poder acceder a los diferentes nodos**, acceder al documento HTML.

Concretamente, **a través del DOM podremos:**

- Cambiar todos los elementos HTML de la página.
- Cambiar todos los atributos HTML.
- Cambiar todos los estilos CSS.

- Añadir o eliminar elementos y atributos existentes.
- Responder a los eventos ocurridos en la página.
- Crear nuevos eventos.

El objeto **document**

Ahora que ya sabemos que «document» está en lo más alto de la jerarquía del DOM, vamos a conocerlo con más detalle.

Según vayamos avanzando iremos viendo que en Javascript casi todo es un objeto, y **«document»** es el objeto que contiene el resto de objetos que conforman la página web. Con lo cual, para acceder y modificar cualquier parte de nuestra página, siempre deberemos hacerlo a través del objeto document.

Recordemos que **en el DOM cada cosa es un nodo**:

- El mismo documento es un nodo de tipo document.
- Todos los elementos HTML son nodos de elemento.
- Todos los atributos son nodos de atributo.
- Los textos dentro de los elementos HTML son nodos de texto.
- Los comentarios son nodos de comentarios.

Como el resto de objetos, document posee diferentes **propiedades y métodos**:

«title»: Esta es una de sus **propiedades** y como podrás imaginar corresponde al título de la página que contiene la etiqueta <title>.

«write»: Este **método** es uno de los muchos métodos que nos proporciona document. Con él podemos escribir en el documento HTML.

Vemos algunas propiedades y métodos muy básicos para acceder y modificar una web a través del DOM:

document.getElementById

Si un elemento tiene el atributo id, podemos obtener el elemento usando el método document.getElementById(id), sin importar dónde se encuentre.

Con innerHTML modificamos el contenido con style el estilo del elemento.

Por ejemplo:

```
<div id="elem">
  <div id="elem-content">Elemento</div>
</div>

<script>
  // obtener el elemento
  let elem = document.getElementById('elem');

  // hacer que su fondo sea rojo
  elem.style.background = 'red';
</script>
```

querySelector y querySelectorAll

La llamada a `elem.querySelector(css)` devuelve todos los elementos dentro de `elem` que coinciden con el selector CSS dado.

La llamada a `elem.querySelector(css)` devuelve el primer elemento para el selector CSS dado.

Es el resultado es el mismo que `elem.querySelectorAll(css)[0]`, pero este último busca todos los elementos y elige uno, mientras que `elem.querySelector` sólo busca uno. Así que es más rápido y también más corto de escribir.

Introducción a los eventos

Un evento es una señal de que algo ocurrió. Todos los nodos del DOM generan dichas señales (pero los eventos no están limitados sólo al DOM).

Vemos los eventos del DOM más utilizados:

Eventos del mouse:

- `click` – cuando el mouse hace click sobre un elemento (los dispositivos touch lo generan con un toque).
- `contextmenu` – cuando el mouse hace click derecho sobre un elemento.
- `mouseover` / `mouseout` – cuando el cursor del mouse ingresa/abandona un elemento.
- `mousedown` / `mouseup` – cuando el botón del mouse es presionado/soltado sobre un elemento.
- `mousemove` – cuando el mouse se mueve.

Eventos del teclado:

- `keydown` / `keyup` – cuando se presiona/suelta una tecla.

Eventos del elemento form:

- `submit` – cuando el visitante envía un `<form>`.

- focus – cuando el visitante se centra sobre un elemento, por ejemplo un `<input>`.

Eventos del documento:

- DOMContentLoaded --cuando el HTML es cargado y procesado, el DOM está completamente construido

Eventos del CSS:

- transitionend – cuando una animación CSS concluye.

Hay muchos más eventos.

Controladores de eventos

Para reaccionar con eventos podemos asignar un handler (controlador) el cual es una función que se ejecuta en caso de un evento.

Los handlers son una forma de ejecutar código JavaScript en caso de acciones por parte del usuario.

Hay muchas maneras de asignar un handler. Vamos a verlas empezando por las más simples.

Atributo HTML

Un handler puede ser establecido en el HTML con un atributo llamado *on<event>*.

Por ejemplo, para asignar un handler click para un input podemos usar onclick, como:

Solo HTML:

```
<input type="button" onclick="alert('Click!')" value="Click me">
```

Al hacer click, el código dentro de onclick se ejecuta.

Propiedad del DOM

Podemos asignar un handler usando una propiedad del DOM *on<event>*.

HTML + JS

```
<input type="button" id="button" value="Botón">
<script>
    button.onclick = function() {
        alert('¡Click!');
    };
</script>
```

Si el handler es asignado usando un atributo HTML entonces el navegador lo lee, crea una nueva función desde el contenido del atributo y lo escribe en la propiedad del DOM.

Como solo hay una propiedad onclick, no podemos asignar más de un handler.

addEventListener

El problema fundamental de las formas ya mencionadas para asignar handlers es que no podemos asignar multiples handlers a un solo evento.

Digamos que una parte de nuestro código quiere resaltar un botón al hacer click, y otra quiere mostrar un mensaje en el mismo click.

La sintaxis para agregar un handler:

```
element.addEventListener(event, handler, [options]);
```

event

Nombre del evento, por ejemplo: "click".

handler

La función handler.

options

Un objeto adicional, opcional, con las propiedades:

- once: si es true entonces el listener se remueve automáticamente después de activarlo.
- capture: la fase en la que se controla el evento, que será cubierta en el capítulo Propagación y captura. Por razones históricas, options también puede ser false/true, lo que es igual a {capture: false/true}.
- passive: si es true entonces el handler no llamará a preventDefault(), esto lo explicaremos más adelante en Acciones predeterminadas del navegador.

Para remover el handler, usa removeEventListener:

```
element.removeEventListener(event, handler, [options]);
```

Múltiples llamadas a addEventListener permiten agregar múltiples handlers:

```
<input id="elem" type="button" value="Haz click en mí"/>
<script>
  function handler1() {
    alert('¡Gracias!');
  };
  function handler2() {
    alert('¡Gracias de nuevo!');
  }
  elem.onclick = () => alert("Hola");
  elem.addEventListener("click", handler1); // Gracias!
  elem.addEventListener("click", handler2); // Gracias de nuevo!
</script>
```

Objeto del evento

Cuando un evento ocurre, el navegador crea un objeto del evento que coloca los detalles dentro y los pasa como un argumento al handler.

handleEvent

Podemos asignar no solo una función, sino un objeto como handler del evento usando `addEventListener`. Cuando el evento ocurre, el método `handleEvent` es llamado.

Por ejemplo:

```
<button id="elem">Haz click en mí</button>

<script>
  let obj = {
    handleEvent(event) {
      alert(event.type + " en " + event.currentTarget);
    }
  };

  elem.addEventListener('click', obj);
</script>
```

Como podemos ver, cuando `addEventListener` recibe como handler a un objeto, llama a `obj.handleEvent(event)` en caso de un evento.

Resumen

Hay tres formas de asignar handlers:

- Atributos HTML: `onclick="..."`.
- Propiedades del DOM: `elem.onclick = function`.
- Métodos: `elem.addEventListener(event, handler[, phase])` para agregar ó `removeEventListener` para remover.

Formularios

La programación de aplicaciones que contienen formularios web siempre ha sido una de las tareas fundamentales de JavaScript. De hecho, una de las principales razones por las que se inventó el lenguaje de programación JavaScript fue la necesidad de validar los datos de los formularios directamente en el navegador del usuario. De esta forma, se evitaba recargar la página cuando el usuario cometía errores al rellenar los formularios.

No obstante, la aparición de las aplicaciones AJAX convirtió el tratamiento de formularios como la principal actividad de JavaScript. Ahora, el principal uso de JavaScript es el de las comunicaciones asíncronas con los servidores y el de la manipulación dinámica de las aplicaciones. De todas formas, el manejo de los formularios sigue siendo un requerimiento imprescindible para cualquier programador de JavaScript.

Propiedades de formularios y elementos

JavaScript dispone de numerosas propiedades y funciones que facilitan la programación de aplicaciones que manejan formularios. En primer lugar, cuando se carga una página web, el navegador crea automáticamente un array llamado `forms` y que contiene la referencia a todos los formularios de la página.

Para acceder al array `forms`, se utiliza el objeto `document`, por lo que `document.forms` es el array que contiene todos los formularios de la página. Como se trata de un array, el acceso a cada formulario se realiza con la misma sintaxis de los arrays. La siguiente instrucción accede al primer formulario de la página:

```
document.forms[0];
```

Además del array de formularios, el navegador crea automáticamente un array llamado `elements` por cada uno de los formularios de la página. Cada array `elements` contiene la referencia a todos los elementos (cuadros de texto, botones, listas desplegables, etc.) de ese formulario. Utilizando la sintaxis de los arrays, la siguiente instrucción obtiene el primer elemento del primer formulario de la página:

```
document.forms[0].elements[0];
```

La sintaxis de los arrays no siempre es tan concisa. El siguiente ejemplo muestra cómo obtener directamente el último elemento del primer formulario de la página:

```
document.forms[0].elements[document.forms[0].elements.length-1];
```

Aunque esta forma de acceder a los formularios es rápida y sencilla, tiene un inconveniente muy grave. ¿Qué sucede si cambia el diseño de la página y en el código HTML se cambia el orden de los formularios originales o se añaden nuevos formularios? El problema es que *"el primer formulario de la página"* ahora podría ser otro formulario diferente al que espera la aplicación.

En un entorno tan cambiante como el diseño web, es muy difícil confiar en que el orden de los formularios se mantenga estable en una página web. Por este motivo, siempre debería evitarse el acceso a los formularios de una página mediante el array `document.forms`.

Una forma de evitar los problemas del método anterior consiste en acceder a los formularios de una página a través de su nombre (atributo `name`) o a través de su atributo `id`. El objeto `document` permite acceder directamente a cualquier formulario mediante su atributo `name`:

```
var formularioPrincipal = document.formulario;  
var formularioSecundario = document.otro_formulario;
```

```
<form name="formulario" >  
  ...  
</form>
```

```
<form name="otro_formulario" >  
  ...  
</form>
```

Accediendo de esta forma a los formularios de la página, el script funciona correctamente aunque se reordenen los formularios o se añadan nuevos formularios a la página. Los elementos de los formularios también se pueden acceder directamente mediante su atributo `name`:

```
var formularioPrincipal = document.formulario;  
var primerElemento = document.formulario.elemento;
```

```
<form name="formulario">  
  <input type="text" name="elemento" />  
</form>
```

Obviamente, también se puede acceder a los formularios y a sus elementos utilizando las funciones DOM de acceso directo a los nodos. El siguiente ejemplo utiliza la habitual función `document.getElementById()` para acceder de forma directa a un formulario y a uno de sus elementos:

```
var formularioPrincipal = document.getElementById("formulario");
var primerElemento = document.getElementById("elemento");

<form name="formulario" id="formulario" >
  <input type="text" name="elemento" id="elemento" />
</form>
```

Independientemente del método utilizado para obtener la referencia a un elemento de formulario, cada elemento dispone de las siguientes propiedades útiles para el desarrollo de las aplicaciones:

- `type`: indica el tipo de elemento que se trata. Para los elementos de tipo `<input>` (`text`, `button`, `checkbox`, etc.) coincide con el valor de su atributo `type`. Para las listas desplegables normales (elemento `<select>`) su valor es `select-one`, lo que permite diferenciarlas de las listas que permiten seleccionar varios elementos a la vez y cuyo tipo es `select-multiple`. Por último, en los elementos de tipo `<textarea>`, el valor de `type` es `textarea`.
- `form`: es una referencia directa al formulario al que pertenece el elemento. Así, para acceder al formulario de un elemento, se puede utilizar `document.getElementById("id_del_elemento").form`
- `name`: obtiene el valor del atributo `name` de XHTML. Solamente se puede leer su valor, por lo que no se puede modificar.
- `value`: permite leer y modificar el valor del atributo `value` de XHTML. Para los campos de texto (`<input type="text">` y `<textarea>`) obtiene el texto que ha escrito el usuario. Para los botones obtiene el texto que se muestra en el botón. Para los elementos `checkbox` y `radiobutton` no es muy útil, como se verá más adelante

Validación de formularios

La principal utilidad de JavaScript en el manejo de los formularios es la validación de los datos introducidos por los usuarios. Antes de enviar un formulario al servidor, se recomienda validar mediante JavaScript los datos insertados por el usuario. De esta forma, si el usuario ha cometido algún error al rellenar el formulario, se le puede notificar de forma instantánea, sin necesidad de esperar la respuesta del servidor.

Notificar los errores de forma inmediata mediante JavaScript mejora la satisfacción del usuario con la aplicación (lo que técnicamente se conoce como "mejorar la experiencia de usuario") y ayuda a reducir la carga de procesamiento en el servidor.

Normalmente, la validación de un formulario consiste en llamar a una función de validación cuando el usuario pulsa sobre el botón de envío del formulario. En esta función, se comprueban si los valores que ha introducido el usuario cumplen las restricciones impuestas por la aplicación.

Aunque existen tantas posibles comprobaciones como elementos de formulario diferentes, algunas comprobaciones son muy habituales: que se rellene un campo obligatorio, que se seleccione el valor de una lista desplegable, que la dirección de email indicada sea correcta, que la fecha introducida sea lógica, que se haya introducido un número donde así se requiere, etc.

A continuación se muestra el código JavaScript básico necesario para incorporar la validación a un formulario:

```
<form action="" method="" id="" name="" onsubmit="return
validacion()">
...
</form>
```

Y el esquema de la función validacion() es el siguiente:

```
function validacion() {
  if (condicion que debe cumplir el primer campo del formulario) {
    // Si no se cumple la condicion...
    console.log('[ERROR] El campo debe tener un valor de...');
    return false;
  }
  else if (condicion que debe cumplir el segundo campo del
formulario) {
    // Si no se cumple la condicion...
```

```

        console.log('[ERROR] El campo debe tener un valor de...');
        return false;
    }
    ...
    else if (condicion que debe cumplir el último campo del
formulario) {
        // Si no se cumple la condicion...
        console.log('[ERROR] El campo debe tener un valor de...');
        return false;
    }

    // Si el script ha llegado a este punto, todas las condiciones
    // se han cumplido, por lo que se devuelve el valor true
    return true;
}

```

El funcionamiento de esta técnica de validación se basa en el comportamiento del evento `onsubmit` de JavaScript. Al igual que otros eventos como `onclick` y `onkeypress`, el evento `onsubmit` varía su comportamiento en función del valor que se devuelve.

Así, si el evento `onsubmit` devuelve el valor `true`, el formulario se envía como lo haría normalmente. Sin embargo, si el evento `onsubmit` devuelve el valor `false`, el formulario no se envía. La clave de esta técnica consiste en comprobar todos y cada uno de los elementos del formulario. En cuando se encuentra un elemento incorrecto, se devuelve el valor `false`. Si no se encuentra ningún error, se devuelve el valor `true`.

Por lo tanto, en primer lugar se define el evento `onsubmit` del formulario como:

```
onsubmit="return validacion()"
```

Como el código JavaScript devuelve el valor resultante de la función `validacion()`, el formulario solamente se enviará al servidor si esa función devuelve `true`. En el caso de que la función `validacion()` devuelva `false`, el formulario permanecerá sin enviarse.

Dentro de la función `validacion()` se comprueban todas las condiciones impuestas por la aplicación. Cuando no se cumple una condición, se devuelve `false` y por tanto el formulario no se envía. Si se llega al final de la función, todas las condiciones se han cumplido correctamente, por lo que se devuelve `true` y el formulario se envía.

La notificación de los errores cometidos depende del diseño de cada aplicación. En el código del ejemplo anterior simplemente se muestran mensajes mediante la función `console.log()` indicando el error producido. Las aplicaciones web mejor diseñadas muestran cada mensaje de error al lado del elemento de formulario correspondiente y también suelen mostrar un mensaje principal indicando que el formulario contiene errores.

Una vez definido el esquema de la función `validacion()`, se debe añadir a esta función el código correspondiente a todas las comprobaciones que se realizan sobre los elementos del formulario. A continuación, se muestran algunas de las validaciones más habituales de los campos de formulario.

Expresiones regulares en JS

Una expresión regular es una combinación de caracteres normales con caracteres especiales. Las Expresiones Regulares son patrones que permiten buscar coincidencias con combinaciones de caracteres dentro de cadenas de texto.

Con la utilización de caracteres especiales se consigue encontrar coincidencias con los retornos de carro, los tabuladores, el inicio o el final de las palabras, las repeticiones de caracteres...

Utilizar la página de referencia <https://regex101.com/> para la generación de expresiones regulares.

Ejemplos de uso de expresiones regulares para la validación de formularios:

– Número de teléfono nacional (sin espacios)

- Ejemplo: 954556817
- Exp. Reg.: `/^\d{9}$/` o también `/^[0-9]{9}$/`

Comienza (^) por una cifra numérica (\d) de la que habrá 9 ocurrencias ({9}) y aquí acabará la cadena (\$).

NOTA: La expresión "\d" equivale a la expresión "[0-9]", y representa a un carácter de una cifra numérica, es decir, '0' o '1' o '2' o '3' ... o '9'.

– Número de teléfono internacional

- Ejemplo: (+34)954556817
- Exp. Reg.: `/^\(\+\d{2,3}\)\d{9}$/`

Comienza (^) por un paréntesis (\()), le sigue un carácter + (\+), después una cifra numérica (\d) de la que habrá 2 o 3 ocurrencias ({2,3}), después

le sigue un paréntesis de cierre ()), luego viene una cifra numérica de la que habrá 9 ocurrencias ({9}), y aquí acabará la cadena (\$).

NOTA: Puesto que los caracteres: (,), +, *, -, \, {, }, |, etc... tienen significados especiales dentro de una expresión regular, para considerarlos como caracteres normales que debe incluir una cadena deben de ir precedidos del carácter de barra invertida \.

– Fecha con formato DD/MM/AAAA

- Ejemplo: 09/01/2006
- Exp. Reg.: `/^\d{2}\V\d{2}\V\d{4}$/`

Comienza (^) por una cifra numérica (\d) de la que habrá 2 ocurrencias ({2}), después una barra (\V), seguida de 2 cifras numéricas, otra barra, 4 cifras numéricas, y aquí acabará la cadena (\$).

– Código postal

- Ejemplo: 41012
- Exp. Reg.: `/^\d{5}$/`

Únicamente contiene 5 cifras numéricas.

– Email

- Ejemplo: usuario@servidor.com
- Exp. Reg.: `/^(.+\\@.+\\.+)$/`

Comienza (^) por caracteres cualesquiera que no sean salto de línea (.) de los que habrá al menos una ocurrencia (+), después el carácter arroba (\@), seguido de al menos un carácter que no podrá ser el salto de línea (.), después viene el carácter punto (\.), seguido de al menos un carácter donde ninguno podrá ser el salto de línea (.), y aquí acabará la cadena (\$).

– Número entero

- Ejemplo: -123
- Exp. Reg.: `/^(\+|\-)?\d+$/` o también `/^[+-]?\d+$/` o también `/^[+-]?[0-9]+$/`

Comienza (^) opcionalmente (?) por el carácter + o por el carácter -, por lo que puede que incluso no aparezcan ninguno de los 2; seguidamente vienen caracteres de cifras numéricas (\d) de los que al menos debe introducirse uno (+), y aquí acabará la cadena (\$).

– Número real

- Ejemplo: -123.35 o 7,4 o 8
- Exp. Reg.: `/^[+-]?\d+([\.,]\d+)?$/`

Comienza (^) opcionalmente (?) por el carácter + o por el carácter -, por lo que puede que incluso no aparezcan ninguno de los 2; seguidamente vienen caracteres de cifras numéricas (\d) de los que al menos debe introducirse uno (+), y, opcionalmente, aparecerá un punto o coma decimal seguido de al menos una cifra numérica, y aquí acabará la cadena (\$).

– Letra minúscula

- Exp. Reg.: [a-z]

– Fecha

- Formato: dd/mm/aaaa
- Exp. Reg.: `^\d{1,2}\d{1,2}\d{2,4}$`

– Hora

- Formato: hh:mm:ss
- Exp. Reg.: `^(0[1-9]|1\d|2[0-3]):([0-5]\d):([0-5]\d)$`

Ejemplos de validaciones en un formulario

```
<html>
<head>
<title>Ejemplo de validación de un formulario usando expresiones regulares</title>
<script>
function ValidaCampos(formulario) {
    var expresion_regular_telefono = /^d{9}$;/; // 9 cifras numéricas.
    var expresion_regular_dni = /^d{8}[a-zA-Z]$;/; // 8 cifras numéricas más un
    carácter alfabético.

    // Usaremos el método "test" de las expresiones regulares:
    if(expresion_regular_telefono.test(formulario.telefono.value)==false) {
        alert('Campo TELEFONO no válido.');
```

```

        </script>
</head>
<body>
    <form name="formulario" action="recoger_datos.php" onSubmit="return
    ValidaCampos(this)">
        DNI: <input type="text" name="dni" size="9" maxlength="9"> <br>
        Teléfono: <input type="text" name="telefono" size="9" maxlength="9"> <br>
        <input type="submit" value="Enviar" name="enviar">
    </form>
</body>
</html>

```

El siguiente ejemplo valida campos de teléfono y direcciones de correo electrónico:

```

function ValidaCampos(formu) {
    //expresión regular para teléfonos
    //permite campos vacíos y guiones
    var er_tlfono = /^(^([0-9\s\+|-]))+$/;
    //expresión regular para emails
    var er_email = /^(.+@.+\.+)$/;
    //comprueba campo tlfono de formu
    //usa el método test de expresión regular
    if(!er_tlfono.test(formu.tlfono.value)) {
        alert('Campo TELEFONO no válido.');
```

return false; //no submit

```

    }
    //comprueba el campo email de formu
    //usa método test de la expresión regular
    if(!er_email.test(formu.email.value)) {
        alert('Campo E-MAIL no válido.');
```

return false; //no submit

```

    }
    return trae; //pasa al submit
}

```

1 Validar un campo de texto obligatorio

Se trata de forzar al usuario a introducir un valor en un cuadro de texto o textarea en los que sea obligatorio. La condición en JavaScript se puede indicar como:

```
valor = document.getElementById("campo").value;
if( valor == null || valor.length == 0 || /^s+$/.test(valor) ) {
    return false;
}
```

Para que se dé por completado un campo de texto obligatorio, se comprueba que el valor introducido sea válido, que el número de caracteres introducido sea mayor que cero y que no se hayan introducido sólo espacios en blanco.

La palabra reservada `null` es un valor especial que se utiliza para indicar "ningún valor". Si el valor de una variable es `null`, la variable no contiene ningún valor de tipo objeto, array, numérico, cadena de texto o booleano.

La segunda parte de la condición obliga a que el texto introducido tenga una longitud superior a cero caracteres, esto es, que no sea un texto vacío.

Por último, la tercera parte de la condición (`/^s+$/.test(valor)`) obliga a que el valor introducido por el usuario no sólo esté formado por espacios en blanco. Esta comprobación se basa en el uso de "expresiones regulares", un recurso habitual en cualquier lenguaje de programación pero que por su gran complejidad no se van a estudiar. Por lo tanto, sólo es necesario copiar literalmente esta condición, poniendo especial cuidado en no modificar ningún carácter de la expresión.

2 Validar un campo de texto con valores numéricos

Se trata de obligar al usuario a introducir un valor numérico en un cuadro de texto. La condición JavaScript consiste en:

```
valor = document.getElementById("campo").value;
if( isNaN(valor) ) {
    return false;
}
```

Si el contenido de la variable `valor` no es un número válido, no se cumple la condición. La ventaja de utilizar la función interna `isNaN()` es que simplifica las comprobaciones, ya que JavaScript se encarga de tener en cuenta los decimales, signos, etc.

A continuación se muestran algunos resultados de la función `isNaN()` :

```
isNaN(3);           // false
isNaN("3");         // false
isNaN(3.3545);      // false
isNaN(32323.345);   // false
isNaN(+23.2);       // false
isNaN("-23.2");     // false
isNaN("23a");       // true
isNaN("23.43.54");  // true
```

3 Validar que se ha seleccionado una opción de una lista

Se trata de obligar al usuario a seleccionar un elemento de una lista desplegable. El siguiente código JavaScript permite conseguirlo:

```
indice = document.getElementById("opciones").selectedIndex;
if( indice == null || indice == 0 ) {
    return false;
}
```

```
<select id="opciones" name="opciones">
  <option value="">- Selecciona un valor -</option>
  <option value="1">Primer valor</option>
  <option value="2">Segundo valor</option>
  <option value="3">Tercer valor</option>
</select>
```

A partir de la propiedad `selectedIndex`, se comprueba si el índice de la opción seleccionada es válido y además es distinto de cero. La primera opción de la lista (- Selecciona un valor -) no es válida, por lo que no se permite el valor `0` para esta propiedad `selectedIndex`.

4 Validar una dirección de email

Se trata de obligar al usuario a introducir una dirección de email con un formato válido. Por tanto, lo que se comprueba es que la dirección parezca válida, ya que no se comprueba si se trata de una cuenta de correo electrónico real y operativa. La condición JavaScript consiste en:

```
valor = document.getElementById("campo").value;
if( !( /\w+ ( [-+.'] \w+ ) * @ \w+ ( [-.] \w+ ) * \. \w+ ( [-.] \w+ ) /.test(valor) ) )
{
    return false;
}
```

La comprobación se realiza nuevamente mediante las expresiones regulares, ya que las direcciones de correo electrónico válidas pueden ser muy diferentes. Por otra parte, como el estándar que define el formato de las direcciones de correo electrónico es muy complejo, la expresión regular anterior es una simplificación. Aunque esta regla valida la mayoría de direcciones de correo electrónico utilizadas por los usuarios, no soporta todos los diferentes formatos válidos de email.

5 Validar una fecha

Las fechas suelen ser los campos de formulario más complicados de validar por la multitud de formas diferentes en las que se pueden introducir. El siguiente código asume que de alguna forma se ha obtenido el año, el mes y el día introducidos por el usuario:

```
var ano = document.getElementById("ano").value;
var mes = document.getElementById("mes").value;
var dia = document.getElementById("dia").value;

valor = new Date(ano, mes, dia);

if( !isNaN(valor) ) {
    return false;
}
```

La función `Date(ano, mes, dia)` es una función interna de JavaScript que permite construir fechas a partir del año, el mes y el día de la fecha. Es muy importante tener en cuenta que el número de mes se indica de 0 a 11, siendo 0 el mes de Enero y 11 el mes de Diciembre. Los días del mes siguen una numeración diferente, ya que el mínimo permitido es 1 y el máximo 31.

La validación consiste en intentar construir una fecha con los datos proporcionados por el usuario. Si los datos del usuario no son correctos, la fecha no se puede construir correctamente y por tanto la validación del formulario no será correcta.

6 Validar un número de DNI

Se trata de comprobar que el número proporcionado por el usuario se corresponde con un número válido de Documento Nacional de Identidad o DNI. Aunque para cada país o región los requisitos del documento de identidad de las personas pueden variar, a continuación se muestra un ejemplo genérico fácilmente adaptable. La validación no sólo debe comprobar que el número esté formado por ocho cifras y una letra, sino

que también es necesario comprobar que la letra indicada es correcta para el número introducido:

```
valor = document.getElementById("campo").value;
var letras = ['T', 'R', 'W', 'A', 'G', 'M', 'Y', 'F', 'P', 'D',
'X', 'B', 'N', 'J', 'Z', 'S', 'Q', 'V', 'H', 'L', 'C', 'K', 'E',
'T'];

if( !(/^\d{8}[A-Z]$/.test(valor)) ) {
    return false;
}

if(valor.charAt(8) != letras[(valor.substring(0, 8))%23]) {
    return false;
}
```

La primera comprobación asegura que el formato del número introducido es el correcto, es decir, que está formado por 8 números seguidos y una letra. Si la letra está al principio de los números, la comprobación sería `/^[A-Z]\d{8}$/. Si en vez de ocho números y una letra, se requieren diez números y dos letras, la comprobación sería /^\d{10}[A-Z]{2}$/ y así sucesivamente.`

La segunda comprobación aplica el algoritmo de cálculo de la letra del DNI y la compara con la letra proporcionada por el usuario. El algoritmo de cada documento de identificación es diferente, por lo que esta parte de la validación se debe adaptar convenientemente.

7 Validar un número de teléfono

Los números de teléfono pueden ser indicados de formas muy diferentes: con prefijo nacional, con prefijo internacional, agrupado por pares, separando los números con guiones, etc.

El siguiente script considera que un número de teléfono está formado por nueve dígitos consecutivos y sin espacios ni guiones entre las cifras:

```
valor = document.getElementById("campo").value;
if( !(/^\d{9}$/.test(valor)) ) {
    return false;
}
```

Una vez más, la condición de JavaScript se basa en el uso de expresiones regulares, que comprueban si el valor indicado es una sucesión de nueve números consecutivos. A continuación se muestran otras expresiones

regulares que se pueden utilizar para otros formatos de número de teléfono:

Número	Expresión regular	Formato
900900900	<code>/^\d{9}\$/</code>	9 cifras seguidas
900-900-900	<code>/^\d{3}-\d{3}-\d{3}\$/</code>	9 cifras agrupadas de 3 en 3 y separadas por guiones
900 900900	<code>/^\d{3}\s\d{6}\$/</code>	9 cifras, las 3 primeras separadas por un espacio
900 90 09 00	<code>/^\d{3}\s\d{2}\s\d{2}\s\d{2}\$/</code>	9 cifras, las 3 primeras separadas por un espacio, las siguientes agrupadas de 2 en 2
(900) 900900	<code>/^\(\d{3}\)\s\d{6}\$/</code>	9 cifras, las 3 primeras encerradas por paréntesis y un espacio de separación respecto del resto
+34 900900900	<code>/^\+\d{2,3}\s\d{9}\$/</code>	Prefijo internacional (+ seguido de 2 o 3 cifras), espacio en blanco y 9 cifras consecutivas

8 Validar que un checkbox ha sido seleccionado

Si un elemento de tipo *checkbox* se debe seleccionar de forma obligatoria, JavaScript permite comprobarlo de forma muy sencilla:

```
elemento = document.getElementById("campo");  
if( !elemento.checked ) {
```

```
    return false;
}
```

Si se trata de comprobar que todos los *checkbox* del formulario han sido seleccionados, es más fácil utilizar un bucle:

```
formulario = document.getElementById("formulario");
for(var i=0; i<formulario.elements.length; i++) {
    var elemento = formulario.elements[i];
    if(elemento.type == "checkbox") {
        if(!elemento.checked) {
            return false;
        }
    }
}
}
```

9 Validar que un *radiobutton* ha sido seleccionado

Aunque se trata de un caso similar al de los *checkbox*, la validación de los *radiobutton* presenta una diferencia importante: en general, la comprobación que se realiza es que el usuario haya seleccionado algún *radiobutton* de los que forman un determinado grupo. Mediante JavaScript, es sencillo determinar si se ha seleccionado algún *radiobutton* de un grupo:

```
opciones = document.getElementsByName("opciones");

var seleccionado = false;
for(var i=0; i<opciones.length; i++) {
    if(opciones[i].checked) {
        seleccionado = true;
        break;
    }
}
if(!seleccionado) {
    return false;
}
```

Recorre todos los *radiobutton* que forman un grupo y comprueba elemento por elemento si ha sido seleccionado. Cuando se encuentra el primer *radiobutton* seleccionado, se sale del bucle.

EJEMPLO

Crear el siguiente formulario:

Registro

Nombre

adsa afad

Apellido

sdsad cadf

email

aaa@jckx.com

Password

.....

Comentarios

dhhgdfhg

Acepta las [condiciones](#)

☒

Enviar

Construir un validador para un formulario de registro que valide cada uno de los campos cuando el usuario ha terminado de introducir datos en cada uno de ellos. Si el campo no cumple las restricciones, se mostrará una alerta al usuario, pero se le permitirá seguir introduciendo datos en el resto de campos.

Las restricciones a cumplir son las siguientes:

- El nombre, email y comentarios son campos obligatorios.
- El campo email debe ser una dirección de email válida.
- El texto introducido en el campo de comentarios no debe exceder los 50 caracteres.
- El password debe tener una longitud mínima de 6 caracteres, y contener al menos una letra minúscula, una letra mayúscula y un dígito.

Validar también que no puedan dejarse en blanco.

NOTA: puedes utilizar tanto validaciones HTML como lo aprendido en Javascript.

En el momento en el que el usuario pulse en enviar el formulario, se validarán todos los campos. En caso de que se informe correctamente guardar la información en un objeto y obtener el JSON correspondiente, en caso contrario mostrar un mensaje de error.

```

<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Formulario, validaciones</title>
  <link rel="stylesheet" href="formularios.css">
  <script defer src="./formularios.js"></script>
</head>

<body>
  <h1>Formularios, Validaciones</h1>
  <form id="Ej1" action="">
    <fieldset>
      <legend>
        <h2>Registro</h2>
      </legend>
      <div class="Ej1">
        <label for="nombre"><strong>Nombre</strong></label>
        <input type="text" id="nombre">

        <label for="apellido"><strong>Apellido</strong></label>
        <input type="text" name="apellido" id="apellido">

        <label for="email"><strong>email</strong></label>
        <input type="text" id="email">

        <label for="password"><strong>Password</strong></label>
        <input type="password" name="password" id="password">

        <label for="comentario"><strong>Comentarios</strong></label>
        <textarea name="Comentario" id="comentario" cols="30"
rows="7"></textarea>
        <div>
          <label for="condiciones">Acepta las <a
href="pagina_condiciones.html">condiciones</a></label>
          <input type="checkbox" name="condiciones" id="condiciones"
/>
        </div>
        <button type="submit" value="Enviar">Enviar</button>
      </div>
    </fieldset>
  </form>
</body>

</html>

```

```

const $FORM = document.getElementById('Ej1');
const $NAME = document.getElementById('nombre');
const $SURNAME = document.getElementById('apellido');
const $EMAIL = document.getElementById('email');
const $PASSWORD = document.getElementById('password');
const $COMMENTS = document.getElementById('comentario');
const $ACCEPTED = document.getElementById('condiciones');

function handleSubmit(e) {
    e.preventDefault();

    const name = $NAME.value;
    const surname = $SURNAME.value;
    const email = $EMAIL.value;
    const password = $PASSWORD.value;
    const comentario = $COMMENTS.value;
    const accepted = $ACCEPTED.checked;

    if (name == false || isValidName(name) == false) {
        alert('El nombre debe ser un valor válido : \n -Debe estar relleno \n -Debe estar compuesto por 1 o 2 palabras');
    }
    else {
        if (email == false || isValidEmail(email) == false) {
            alert('El email debe ser un valor válido : \n -Debe estar relleno \n -Debe estar compuesto por un @ y un ".algo"');
        }
        else {
            if (comentario == false || isValidComent(comentario) == false) {
                alert('El comentario tiene que tener un formato válido : \n -Debe estar relleno \n -Debe de tener un total de 50 caracteres');
            }
            else {
                if (isValidPassword(password) == false) {
                    alert('El password debe tener una longitud mínima de 6 caracteres, y contener al menos una letra minúscula, una letra mayúscula y un dígito');
                }
                else {
                    var regUsuario = {nombre:name, apellido:surname, email:email, comentario:comentario, password:password};
                    console.log(regUsuario);
                    var newUser = JSON.stringify(regUsuario);
                    console.log(newUser);
                }
            }
        }
    }
}

```

```

    }
  }
}

function isValidName(name) {
  const validacion =
/^(([\wáéíóúÁÉÍÓÚ]+)|([\wáéíóúÁÉÍÓÚ]+\s[\wáéíóúÁÉÍÓÚ]+))$/;
  return validacion.test(name);
}

function isValidEmail(email) {
  const validacion = /^(.+@.+\.+.+)$/;
  return validacion.test(email);
}

function isValidPassword(password) {
  const validacion = /^(?=\w*\d)(?=\w*[A-Z])(?=\w*[a-z])\S{6,}$/;
  return validacion.test(password);
}

function isValidComent(comentario) {
  const validacion = /^[s\S]{1,50}$/;
  return validacion.test(comentario);
}

$FORM.addEventListener('submit', handleSubmit);

```

Almacenamiento en el lado cliente

Se han desarrollado funcionalidades que permiten el almacenamiento de información en el lado del cliente a través de Javascript.

Las cookies presentan limitaciones de modo que se restringen al manejo de cadenas de texto sin integración en un sistema de datos.

La API Web Storage permite un manejo de los datos más potentes que las cookies, que facilita el almacenamiento de información útil para la navegación web.

La información almacenada queda relacionada con un sitio o aplicación web, de forma que no puede ser utilizada por un sitio o aplicación distinta a la que creó los datos.

Cookies

Las cookies surgieron como necesidad ante algunas ausencias tecnológicas del protocolo HTTP.

Una cookie es un fichero de texto que se almacena en el ordenador del cliente.

Son un fichero propio de cada navegador.

Surgieron con el fin de mantener la información de los carritos de compra virtuales a través de la web.

Mantener opciones de visualización:

- Las cookies son utilizadas en ocasiones para mantener unas preferencias de visualización.
- Algunas páginas como Google, permiten que el usuario haga una configuración de su página de entrada en el buscador, a través de las cookies el servidor reconoce ciertos aspectos que el usuario configuró y conserva el aspecto.

Almacenar variables:

- El servidor puede utilizar las cookies para almacenar variables que se necesiten utilizar en el navegador.
- Un ejemplo sería, una página en la que nos solicitan unos datos, en la siguiente nos solicitan otros datos y así hasta la página final. Los datos de las páginas anteriores se irán almacenando en las cookies hasta que se finaliza el ciclo del formulario y el usuario envía los datos al servidor.
- Antes del envío al servidor se recuperarán todos los campos del formulario que están guardados en las cookies.

Realizar un seguimiento de la actividad del usuario:

- En ocasiones los servidores hacen uso de las cookies para almacenar ciertas preferencias y hábitos que el usuario tiene a la hora de navegar.
- Con esta información, el servidor personaliza sus servicios y publicidad orientándolo a cada cliente en particular.
- Estos fines no son del todo lícitos si la entidad que realiza estas actividades no avisa al usuario de que está realizando estas acciones.

Autenticación:

- Autenticar a los usuarios es uno de los usos más habituales de las cookies.
- A través de las cookies, el navegador guarda los datos del usuario, al realizar una petición al servidor, el navegador envía las cookies junto con la petición.
- Las cookies tienen una caducidad, cuando pasa un periodo de tiempo establecido, estas desaparecen junto con el fichero de texto que guarda el navegador.
- Habitualmente, como mecanismo de seguridad, las aplicaciones Web aplican un tiempo máximo de inactividad, por ejemplo de 15 minutos, tras el cual las cookies caducan, si no se produjo movimiento en la navegación de la aplicación web.

Formato de registro de una cookie

- Dominio del servidor que creó la cookie.
- Información de si es necesaria una conexión http segura para acceder a la cookie.
- Trayectoria de las URL que podrán acceder a la cookie.
- Fecha de caducidad de la cookie.
- Nombre de una entrada de datos.
- Cadena de texto (valor) asociada a ese nombre de entrada de datos.

Grabar una cookie:

```
document.cookie = "nombreCookie = datosCookie"  
[; expires=horaFormatoGMT] // Ej. expires=01 Jan 70 00:00:00 GMT;  
[; path=ruta] //Ruta actual de nuestra página web.  
[; domain=nombreDominio] //Por defecto, el que creó la cookie.  
[; secure] // Si no se pone, puede ser accesible por cualquier programa  
que se ajuste al dominio y path
```

Borrar una cookie:

Se le asigna una fecha de caducidad anterior a la actual.

Ejemplos:

```
document.cookie = "usuario=Pepe; expires =1 Nov 2022, 23:59:59 GMT";  
document.cookie = "contador=0"
```


Si la cookie no existe, la crea; si existe, la sobrescribe.

Función para **inicializar una cookie**:

```
document.cookie = "usuario=Pepe; expires =1 Nov 2022, 23:59:59 GMT";
```

```
function setCookie (cname, cvalue, exdays) {  
    let d = new Date (); // Se extrae la fecha actual  
    //Se le suman los días indicados por parámetro  
    d.setTime(d.getTime() + exdays * 24 * 60 * 60 * 1000);  
    let expires = "expires=" + d.toUTCString (); // Se pasa a formato GMT  
    document.cookie = cname + "=" + cvalue + "; " + expires; //Se crea la  
    cookie  
}
```

Leer una cookie:

```
//Devuelve toda la cadena con todas las variables;
```

```
let x = document.cookie;
```

Leer un valor concreto de la cookie:

```
function getCookie(cname) {  
    let name = cname + "=";  
    let ca = document.cookie.split(";"); //Divide la cookie separando por ;  
    for (let i = 0; i < ca.length; i++) {  
        //Recorre cada trozo de la cookie  
        let c = ca[i];  
        while (c.charAt(0) == " ") {  
            c = c.substring(1);  
        }  
        if (c.indexOf(name) == 0 ) //Si encuentra el campo buscado  
            return c.substring(name.length, c.length) //lo devuelve  
    }  
    return ""; //Si no devuelve vacío  
}
```

Comprobar si una cookie está inicializada:

```
function checkCookie() {  
    let username = getCookie("username"); //extrae el valor función anterior  
    if (username != "") { //Si no está vacío  
        alert(("Bienvenido de nuevo, " + username)); //Se le saluda  
    } else {  
        username = prompt("Introduzca su nombre: ", "");  
        if (username != "" && username != null) {  
            setCookie("username", username, 365); //Se inicializa la variable  
        }  
    }  
}
```

API Web Storage

Con Web Storage se puede almacenar información útil para la navegación web.

Tiene las siguientes propiedades:

- La cantidad de datos que se pueden guardar es limitada.
- Los datos almacenados no se transmiten, solo se almacenan en el navegador.
- No tienen fecha de expiración.
- Almacenan datos clave valor.
- Las claves valor están asociadas a un dominio.

Existen dos posibilidades de almacenamiento:

- `sessionStorage`: destinada a almacenar información durante el tiempo de vida de una sesión web (ej. Mantenimiento de los productos en un carrito de compra).
- `localStorage`: destinada a mantener los datos de forma permanente (ej. Mantener un informe de negocio).

SESSION STORAGE

Resumen de métodos y propiedades de `sessionStorage`:

Método o propiedad de <code>sessionStorage</code>	Descripción <code>aprenderaprogramar.com</code>
<code>sessionStorage.setItem('clave', 'valor');</code>	Guarda la información valor a la que se podrá acceder invocando a clave. Por ejemplo clave puede ser nombre y valor puede ser Francisco.
<code>sessionStorage.getItem('clave')</code>	Recupera el value de la clave especificada. Por ejemplo si clave es nombre puede recuperar "Francisco".
<code>sessionStorage[clave]=valor</code>	Igual que <code>setItem</code>
<code>sessionStorage.length</code>	Devuelve el número de items guardados por el objeto <code>sessionStorage</code> actual
<code>sessionStorage.key(i)</code>	Cada item se almacena con un índice que comienza por cero y se incrementa unitariamente por cada item añadido. Con esta sintaxis rescatamos la clave correspondiente al item con índice i.
<code>sessionStorage.removeItem(clave)</code>	Elimina un item almacenado en <code>sessionStorage</code>
<code>sessionStorage.clear()</code>	Elimina todos los items almacenados en <code>sessionStorage</code> , quedando vacío el espacio de almacenamiento.

PERSISTENCIA DE LOS DATOS CON SESSION STORAGE

Los datos con `sessionStorage` son accesibles mientras dura la sesión de navegación. Los datos no son recuperables si:

- a) Se cierra el navegador y se vuelve a abrir.
- b) Se abre una nueva pestaña de navegación independiente y se sigue navegando en esa pestaña.
- c) Se cierra la ventana de navegación que se estuviera utilizando y se abre otra.

LOCALSTORAGE

Con sessionStorage se puede tratar adecuadamente el flujo de información durante sesiones de navegación. ¿Pero qué ocurre si quisiéramos almacenar esa información más allá de una sesión y que estuviera disponible tanto si se cierra el navegador y se vuelve a abrir como si se continúa navegando en una ventana distinta de la inicial?

Esto trata de ser respondido por localStorage. Este objeto tiene las mismas propiedades y métodos que sessionStorage, pero su persistencia va más allá de la sesión.

Sin embargo localStorage tiene limitaciones importantes: el almacenamiento de los datos depende, en última instancia, de la voluntad del usuario. Aunque la forma en que se almacenan los datos depende del navegador, podemos considerar que una limpieza de caché del navegador hará que se borren los datos almacenados con localStorage. Si el usuario no limpia la caché, los datos se mantendrán durante mucho tiempo. En cambio hay usuarios que tienen configurado el navegador para que la caché se limpie en cada ocasión en que cierran el navegador. En este caso la persistencia que ofrece localStorage es similar a la que ofrece sessionStorage.

No podemos confiar el funcionamiento de una aplicación web a que el usuario limpie o no limpie caché, por tanto deberemos seguir trabajando con datos del lado del servidor siempre que deseemos obtener una persistencia de duración indefinida.

Comprobar si el navegador es compatible:

```
if (typeof Storage !== "undefined") {  
    //LocalStorage disponible  
} else {  
    //LocalStorage no soportado en este navegador  
}
```

Guardar datos:

```
// Guardar  
localStorage.setItem("Alixar", "DAW2");
```

Recuperar datos:

```
// Conseguir elemento  
localStorage.getItem("Alixar");
```

Guardar objetos en el LocalStorage:

```
localStorage.setItem("usuario", JSON.stringify(mi_objeto));
```

Recuperar objetos del localStorage:

```
// Conseguir el elemento
```

```
JSON.parse(localStorage.getItem("usuario"));
```

Borrar o vaciar el localStorage (para un elemento concreto):
`localStorage.removeItem("titulo");`

Borrar o vaciar el localStorage (el del dominio entero):
`localStorage.clear();`

Para el caso del sessionStorage, se utilizan exactamente los mismos métodos y propiedades que para el objeto localStorage, pero sobre el objeto sessionStorage:

```
sessionStorage.setItem("Alixar", "DAW2");  
sessionStorage.setItem("usuario", JSON.stringify(mi_objeto));  
sessionStorage.getItem("Alixar");  
sessionStorage.removeItem("titulo");  
sessionStorage.clear();
```

EL EVENTO STORAGE

Dado que localStorage permite que se reconozcan datos desde distintas ventanas ¿Cómo detectar en una ventana que se ha producido un cambio en los datos? Para ello se definió el evento storage: este evento se dispara cuando tiene lugar un cambio en el espacio de almacenamiento y puede ser detectado por las distintas ventanas que estén abiertas.

Para crear una respuesta a este evento podemos escribir:
`window.addEventListener("storage", nombreFuncionRespuesta, false);`
Donde *nombreFuncionRespuesta* es el nombre de la función que se invocará cuando se produzca el evento.

DIFERENCIAS ENTRE COOKIES Y STORAGE

Los objetos storage juegan un papel en alguna medida similar a las cookies, pero por otro lado hay diferencias importantes:

- a) Las cookies están disponibles tanto en el servidor como en el navegador del usuario, los objetos storage sólo están disponibles en el navegador del usuario.
- b) Las cookies se concibieron como pequeños paquetes de identificación, con una capacidad limitada (unos 4 Kb). Los objetos storage se han concebido para almacenar datos a mayor escala (pudiendo comprender cientos o miles de datos con un espacio de almacenamiento típicamente de varios Mb).

Tener en cuenta que de una forma u otra, ni las cookies ni los objetos storage están pensados para el almacenamiento de grandes volúmenes de información, sino para la gestión de los flujos de datos propios de la navegación web.