

Desarrollo Web en Entorno Cliente

UD 02. Sintaxis Javascript ES6

Actualizado Septiembre 2021

Nomenclatura

A lo largo de este tema se utilizarán distintos símbolos para distinguir elementos importantes dentro del contenido. Estos símbolos son:

 **Importante**

 **Atención**

 **Interesante**

ÍNDICE DE CONTENIDO

1. Introducción	4
2. Hola mundo y comentarios	4
3. Variables, constantes y arrays	5
3.1. Ámbito de variables y constantes	5
3.2. Variables	5
3.3. Tipos de datos	6
3.4. Coerción	7
3.5. Arrays	9
3.6. Clonando arrays (y objetos)	9
3.7. Conversiones entre tipos	10
3.8. Constantes	10
3.9. Modo estricto "use strict"	11
4. Entrada y salida en navegadores	12
4.1. alert	12
4.2. console.log	12
4.3. confirm	12
4.4. prompt	12
5. Operadores	13
5.1. Operadores de asignación	13
5.2. Operadores aritméticos	13
5.3. Operadores de comparación	14
5.4. Operadores lógicos	16
6. Estructuras de control	17

6.1. Instrucciones if/else	17
6.2. Ramificaciones anidadas	18
7. Estructura repetitivas (Bucles)	18
7.1. Bucle for	18
7.2. Bucle while	19
7.3. Bucle do-while	20
7.4. Instrucciones BREAK y CONTINUE	21
8. Funciones	22
9. Funciones flecha (arrow functions)	24
10. Clases en Javascript	25
11. Más cosas interesantes de Javascript ES6	26
12. Introducción a la manipulación del DOM	26
13. Material adicional	27
14. Bibliografía	27

UD02. SINTAXIS JAVASCRIPT ES6

1. INTRODUCCIÓN

Javascript es un lenguaje de programación que inicialmente nació como un lenguaje que permitía ejecutar código en nuestro navegador (cliente), ampliando la funcionalidad de nuestros sitios web.

Una de las versiones más extendidas de Javascript moderno, es la llamada por muchos **JavaScript ES6** (ECMAScript 6), también llamado ECMAScript 2015 o incluso por algunos llamado directamente Javascript 6. Al crear esta pequeña guía nos hemos basado en esta versión. Si ya sabes Javascript pero quieres conocer novedades de Javascript ES6, te recomendamos este curso <https://didacticode.com/curso/curso-javascript-es6/>

Actualmente esa es una de sus principales funciones, pero dada su popularidad el lenguaje ha sido portado a otros ámbitos, entre los que destaca el popular NodeJS <https://nodejs.org/> que permite la ejecución de Javascript como lenguaje escritorio y lenguaje servidor.

Aunque en este módulo nos centramos en la ejecución de Javascript en el navegador, lo aprendido puede ser utilizado para otras implementaciones de Javascript.

2. HOLA MUNDO Y COMENTARIOS

Para añadir JavaScript se usa la etiqueta SCRIPT.

Este puede estar en cualquier lugar de la página. El código se ejecuta en el lugar donde se encuentra de forma secuencial a como el navegador lo va encontrando.

```
<SCRIPT>
Aquí va el código
// Esto es un comentario en Javascript de una línea
</SCRIPT>
```

En Javascript los comentarios se pueden hacer con // para una línea y con /* */ para varias líneas.

Asimismo, desde Javascript es posible escribir mensajes a la consola de desarrollo mediante la orden "console.log(texto)".

Este ejemplo podría ser un pequeño hola mundo que al ejecutarse se mostrará en la consola de desarrollo. Ejemplo con "console.log" y comentario multilínea.

```
<SCRIPT>
console.log ("Hola mundo");
/* Esto es un comentario en
Javascript multilínea */
</SCRIPT>
```

Otra vía para mostrar información al usuario desde una ventana, es el comando “alert(texto)”.

```
<SCRIPT>
alert("Hola mundo");
</SCRIPT>
```

Hay una forma mucho más práctica y ordenada de usar código Javascript. Se pueden incluir uno o varios ficheros con código Javascript en nuestro documento HTML. Se puede incluir tantos como se desee. Esta es la forma más adecuada para trabajar con código Javascript.

Un ejemplo de inclusión de ficheros.

```
<script src="rutaDelArchivo1.js"/>
<script src="rutaDelArchivo2.js"/>
<script src="rutaDelArchivo3.js"/>
```

3. VARIABLES, CONSTANTES Y ARRAYS

En este punto hablaremos de variables, constantes y arrays. Antes de empezar, comentar que es recomendable mantener un estilo de nombramiento de variables. Aquí un enlace sobre formas de nombrar variables:

<https://medium.com/@alonsus91/convenci%C3%B3n-de-nombres-desde-el-camelcase-hasta-el-kebab-case-787e56d6d023>

Durante el curso, usaremos el estilo “CamelCase”

https://es.wikipedia.org/wiki/Camel_case

3.1 Ámbito de variables y constantes

Antes de comenzar, vamos a hablar del ámbito de variables (también llamado “scope” en inglés). El ámbito de variables en un lenguaje de programación indica en qué lugares del programa puede ser accedida una variable/constante. Al comentar cada uno de los tipos, definiremos en qué ámbito existen.

3.2 Variables

Las variables son elementos del lenguaje que permiten almacenar distintos valores en cada momento. Se puede almacenar un valor en una variable y consultar este valor posteriormente. También podemos modificar su contenido siempre que queramos.

Para declarar las variables en JavaScript podemos utilizar **let** o **var**, según el ámbito donde queramos que sea accesible.

- **let:** let permite declarar una variable que sea accesible únicamente dentro del bloque donde se ha declarado (llamamos bloque al espacio delimitado por { }).
- **var:** var permite declarar una variable que sea accesible por todos los lugares de la función donde ha sido declarada. Si una variable con var se declara fuera de cualquier función, el ámbito de esta son todas las funciones del código.
- **Variables sin declarar:** Javascript nos permite usar variables no declaradas. Si hacemos esto, será equivalente a declararlas con **var** fuera del código, es decir, serán variables accesibles por cualquier función.

```
function ejemplo(){
  ejemplo=3; // Equivale a declararla fuera de la funcion como var
  if (ejemplo === 3){
    var variable1 = 1;
    let variable2 = 2;
  }
  console.log(variable1); // variable1 existe en este lugar
  console.log(variable2); // variable2 no existe en este lugar
}
```

❏ Importante: En general, recomendamos usar **let**. Deberías tener una buena razón para usar **var**, ya que su uso es peligroso ya que podría modificarse una variable desde un lugar que no controles por accidente. Estos problemas no ocurren únicamente en Javascript, sino en otros lenguajes y pueden hacer que tu programa sea complicado de depurar.

3.3 Tipos de datos

Los principales tipos de datos que puede contener variables en Javascript son:

- **Numéricos (tipo “number”):** puede contener cualquier tipo de número real (0.3, 1.7, 2.9) o entero (5, 3, -1).
- **Enteros grandes (tipo “bigint”):** pueden contener enteros con valores superiores a $2^{53} - 1$. Se pueden nombrar escribiendo una letra “n” al final del entero. No pueden utilizarse con la mayoría de operadores matemáticos de Javascript.
- **Booleanos (tipo “boolean”):** puede contener uno de los siguientes valores: true, false, 1 y 0.
- **Cadenas (tipo “string”):** cualquier combinación de caracteres (letras, números, signos especiales y espacios). Las cadenas se delimitan mediante comillas dobles o simples (“Lolo”, “lalO”). Para concatenar cadenas puede usarse el operador “+”.

El tipo de una variable puede comprobarse usando la estructura **typeof** `variable===”tipo”`.

```
let edad=23, nueva_edad, incremento=4;
const nombre="Rosa García";
console.log(typeof incremento===”number”)
nueva_edad=edad+incremento;
console.log(nombre+ " tras "+incremento +” años tendrá ”+ nueva_edad);
```

Asimismo, existen otros tipos que Javascript considera primitivos: “undefined”, “null”, “symbol”, “object” y “function”.

Todos los valores que NO son de un tipo básico son considerados objetos: arrays, funciones, valores compuestos, etc. Esta distinción es muy importante porque los valores primitivos y los valores objetos se comportan de distinta forma cuando son asignados y cuando son pasados como parámetro a una función.

Más información en

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

3.4 Coerción

Javascript es un lenguaje de tipado blando (es decir, al declarar una variable no se le asigna un tipo), aunque internamente Javascript si maneja tipos de datos.

En determinados momentos, resulta necesario convertir un valor de un tipo a otro. Esto en JS se llama “coerción”, y puede ocurrir de forma implícita o podemos forzarlo de forma explícita.

Por ejemplo

```
let numero = 5;
console.log(numero);
```

En este código, ocurre una coerción **implícita** de número (que es de tipo number) a un tipo string, de modo que puede ser impreso por consola. Podríamos realizar la conversión de forma **explícita** de la siguiente forma:

```
console.log(numero.toString());
```

Las coerciones implícitas ocurren muy a menudo en JS, aunque muchas veces no seamos conscientes de ello. Resulta muy importante entender cómo funcionan para poder deducir cuál será el resultado de una comparación.

```
let a = "2", b = 5;
console.log( typeof a + " " + typeof b); // string number
console.log( a + b ); // nos muestra 15
```

En los lenguajes de tipado duro (por ejemplo, Java) se nos prohíbe realizar operaciones entre distintos tipos de datos. Sin embargo, Javascript, no hace eso, ya que permite operar entre distintos tipos, siguiendo una serie de reglas:

- Javascript tiene el operador `===` y `!==` para realizar comparaciones estrictas, pero no posee esos operadores para desigualdades (`>`, `<`, `>=`, `<=`).
- Si es posible, JS prefiere hacer coerciones a tipo `number` por encima de otros tipos básicos. Por ejemplo, la expresión `("15" < 100)` se resolverá como `true` porque JS cambiará `"15"`, de tipo `string`, por `15` de tipo `number`.
 - Ten en cuenta que, si conviertes `"15"` a `string`, al compararlo con `"100"` la expresión se resolvería como `false`.
- A la hora de hacer coerción a `boolean`, los siguientes valores se convertirán en `false`: `undefined`, `null`, `0`, `""`, `NaN`. El resto de valores se convertirán en `true`.

Ejemplo coerción a `number`


```
// <, <=, >, >= también hacen coercion. No existe >= ni <=
let arr = [ "1", "10", "100", "1000" ];
for (let i = 0; i < arr.length && arr[i] < 500; i++) {
  console.log(i);
} // 0, 1, 2
```

Ejemplo donde no se hace coerción

```
var x = "10";
var y = "9";
console.log(x < y); // true, Los dos son String y los compara como cadena
```

Ejemplo de coerción con `undefined`

```
let altura; // variable no definida
console.log(altura ? true : false); // Al no estar definido, false
```

 **Interesante:** al realizar comparaciones, si usas `==` o `!=` para comparar los datos, Javascript realiza coerción. Si quieres que la comparación no convierta tipos y solo sea cierta si son del mismo tipo, debes usar `===` o `!==`. Esta es una buena práctica muy recomendada para que estas conversiones no nos jueguen malas pasadas.

Para más información <https://www.etnassoft.com/2011/04/06/coercion-de-datos-en-javascript/>

3.5 Arrays

Un array (también llamado vector o arreglo) es una variable que contiene diversos valores. Lo creamos simplemente con “[]” o con “new Array()” o inicializando los elementos. Podemos referenciar los elementos de un array indicando su posición.

Los arrays poseen una propiedad llamada “length” que podemos utilizar para conocer el número de elementos del array.

```
// Array definido 1 a 1
let miVector=[]; // let miVector=new Array(); es equivalente
miVector[0]=22;
miVector[1]=12;
miVector[2]=33;
//Array definido en una línea inicializando elementos
let otroArray=[1,2,"Cancamusa"]; // Valores dentro del array
console.log(miVector[1]);
console.log(otroArray[2]);
console.log(miVector + " "+miVector.length ); // array y tamaño
// Array bidimensional 5x4 declarado sin rellenar
// Para saber mas de map, mirar seccion de funciones flecha
let matrizBi = new Array(5).fill().map(x => new Array(4));
// Otro array bidimensional 3x5, relleno de ceros
let otraMatrizBi = [...Array(3)].map(x=>Array(5).fill(0))
```

3.6 Clonando arrays (y objetos)

Los arrays en Javascript internamente almacenan referencias a donde está alojado cada objeto en memoria, por lo cual copiar un array no es tan simple como hacer `miArray=miOtroArray`.

En Javascript ES6, se puede hacer una copia sencilla de los valores de un array unidimensional así:

```
let miArray= { ...miOtroArray };
```

Esta copia solo funciona con arrays unidimensionales, ya que con multidimensionales, solo copiará las referencias de memoria de cada uno de estos.

Se pueden hacer copias completas con métodos como este, que se basa en convertir el array a JSON y volver a obtenerlo desde JSON:

```
let miArrayMultidimensional =
JSON.parse(JSON.stringify(miOtroArrayMultidimensional));
```

Más información en <https://www.samanthaming.com/tidbits/70-3-ways-to-clone-objects/>

3.7 Conversiones entre tipos

Javascript no define explícitamente el tipo de datos de sus variables. Según se almacenen, pueden ser cadenas (Entre Comillas), enteros (sin parte decimal) o decimales (con parte decimal).

Elementos como la función “prompt” para leer de teclado con una ventana emergente del navegador, leen los elementos siempre como cadena. Para estos casos y otros, merece la pena usar funciones de conversión de datos.

```
let num="100"; //Es una cadena
let num2="100.13"; //Es una cadena
let num3=11; // Es un entero
let n=parseInt(num); // Almacena un entero. Si hubiera habido parte decimal la truncará
let n2=parseFloat(num); // Almacena un decimal
let n3=num3.toString(); // Almacena una cadena
```

3.8 Constantes

Las constantes son elementos que permiten almacenar un valor, pero ese valor una vez almacenado es invariable (permanece constante). Para declarar constantes se utiliza la instrucción “const”. Suelen ser útiles para definir datos constantes, como el número PI, el número de Euler, etc...

Su ámbito es el mismo que el de **let**, es decir, solo son accesibles en el bloque que se han declarado.

```
const PI=3.1416;
console.log(PI);
PI=3; // Esto falla
```

Aunque resulta posible definir arrays y objetos usando **const**, no es recomendable hacerlo, ya que es posible que su uso no sea el que pensamos.

Por ejemplo, al declarar un array/objeto, realmente lo que ocurre es que la variable almacena la dirección de memoria del objeto/array. Si lo declaramos usando **const**, lo que haremos es que no pueda cambiarse esa dirección de memoria, pero nos permitirá cambiar sus valores.

```
const miArray=[1,2,3]
console.log(miArray[0]); // Muestra el valor 1
miArray[0]=4;
console.log(miArray[0]); // Muestra el valor 4
miArray=[]; // Esto falla
```

3.9 Modo estricto "use strict"

Javascript ES6 incorpora el llamado "modo estricto". Si en algún lugar del código se indica la sentencia "use strict" indica que ese código se ejecutará en modo estricto:

- Escribir "use strict" fuera de cualquier función afecta a todo el código.
- Escribir "use strict" dentro de una función afecta a esa función.

Entre las principales características de "use strict" tenemos que **obligar a que todas las variables sean declaradas**.

"Use strict" es ignorado por versiones anteriores de Javascript, al tomarlo como una simple declaración de cadena.

Las principales características de "use strict" son:

- No permite usar variables/objetos no declarados (los objetos son variables).
- No permite eliminar (usando delete) variables/objetos/funciones.
- No permite nombres duplicados de parámetros en funciones.
- No permite escribir en propiedades de objetos definidas como solo lectura.
- Evita que determinadas palabras reservadas sean usadas como variables (eval, arguments, this, etc...)

Ejemplo 1:

```
"use strict";
pi=3.14;           // Da error
funcionPrueba();

function funcionPrueba() {
  piBIS=3.14;      // Da error
}
```

Ejemplo 2:

```
pi=3.14;           // Da error, use strict sólo se aplica a la función
funcionPrueba();

function funcionPrueba() {
  "use strict";
  piBIS=3.14;      // Da error
}
```

4. ENTRADA Y SALIDA EN NAVEGADORES

4.1 alert

El método `alert()` permite mostrar al usuario información literal o el contenido de variables en una ventana independiente. La ventana contendrá la información a mostrar y el botón Aceptar.

```
alert("Hola mundo");
```

4.2 console.log

El método `console.log` permite mostrar información en la consola de desarrollo. En versiones de Javascript de escritorio tipo NodeJS, permite mostrar texto “por pantalla”.

```
console.log("Otro hola mundo");
```

4.3 confirm

A través del método `confirm()` se activa un cuadro de diálogo que contiene los botones Aceptar y Cancelar. Cuando el usuario pulsa el botón Aceptar, este método devuelve el valor `true`; Cancelar devuelve el valor `false`. Con ayuda de este método el usuario puede decidir sobre preguntas concretas e influir de ese modo directamente en la página.

```
let respuesta;  
respuesta=confirm ("¿Desea cancelar la suscripción?");  
alert("Usted ha contestado que "+respuesta);
```

4.4 prompt

El método `prompt()` abre un cuadro de diálogo en pantalla en el que se pide al usuario que introduzca algún dato. Si se pulsa el botón Cancelar, el valor de devolución es `false/null`. Pulsando en Aceptar se obtiene el valor `true` y la cadena de caracteres introducida se guarda para su posterior procesamiento.

```
let provincia;  
provincia=prompt("Introduzca la provincia ","Valencia");  
alert("Usted ha introducido la siguiente información "+provincia);  
console.log("Operación realizada con éxito");
```

5. OPERADORES

Combinando variables y valores, se pueden formular expresiones más complejas. Las expresiones son una parte clave en la creación de programas. Para formular expresiones utilizamos los llamados operadores. Pasamos a comentar los principales operadores de Javascript.

5.1 Operadores de asignación

Los operadores de asignación se utilizan para asignar valores a las variables. Algunos de ellos, además de asignar el valor, también incluyen operaciones (Ejemplo += asigna y suma).

Operador	Descripción
=	Asigna a la variable de la parte izquierda el valor de la parte derecha.
+=	Suma los operandos izquierdo y derecho y asigna el resultado al operando izquierdo.
-=	Resta el operando derecho del operando izquierdo y asigna el resultado al operando izquierdo.
*=	Multiplica ambos operandos y asigna el resultado al operando izquierdo.
/=	Divide ambos operandos y asigna el resultado al operando izquierdo.

```
let num1=3;
let num2=5;
num2+=num1;
num2-=num1;
num2*=num1;
num2/=num1;
num2%=num1;
```

5.2 Operadores aritméticos

Los operadores aritméticos se utilizan para realizar cálculos aritméticos.

Operador	Descripción
+	Suma.
-	Resta.

*	Multiplicación.
/	División.
%	Calcula el resto de una división entera.

Además de estos operadores, también existen operadores aritméticos unitarios: incremento (++), disminución (--) y la negación unitaria (-).

Los operadores de incremento y disminución pueden estar tanto delante como detrás de una variable, con distintos matices en su ejecución. Estos operadores aumentan o disminuyen en 1, respectivamente, el valor de una variable.

Operador	Descripción (Suponiendo x=5)
y = ++x	Primero el incremento y después la asignación. x=6, y=6.
y = x++	Primero la asignación y después el incremento. x=6, y=5.
y = --x	Primero el decremento y después la asignación. x=4, y=4.
y = x--	Primero la asignación y después el decremento x=4, y=5.
y = -x	Se asigna a la variable "y" el valor negativo de "x", pero el valor de la variable "x" no varía. x=5, y= -5.

```
let num1=5, num2=8, resultado1, resultado2;
resultado1=((num1+num2)*200)/100;
resultado2=resultado1%3;
resultado1=++num1;
resultado2=num2++;
resultado1=--num1;
resultado2=num2--;
resultado1=-resultado2;
```

5.3 Operadores de comparación

Operadores utilizados para comparar dos valores entre sí. Como valor de retorno se obtiene siempre un valor booleano: *true* o *false*.

Operador	Descripción
===	Compara dos elementos, incluyendo su tipo interno. Si son de distinto tipo, no realiza conversión y devuelve false ya que siempre los considera diferentes. <u>Uso recomendado.</u>
!==	Compara dos elementos, incluyendo su tipo interno. Si son de distinto tipo, no realiza conversión y devuelve true ya que siempre los considera diferentes. <u>Uso recomendado.</u>
==	Devuelve el valor <i>true</i> cuando los dos operandos son iguales. Si los elementos son de distintos tipos, realiza una conversión. <u>No está recomendado su uso.</u>
!=	Devuelve el valor <i>true</i> cuando los dos operandos son distintos. Si los elementos son de distintos tipos, realiza una conversión. <u>No está recomendado su uso.</u>
>	Devuelve el valor <i>true</i> cuando el operando de la izquierda es mayor que el de la derecha.
<	Devuelve el valor <i>true</i> cuando el operando de la derecha es menor que el de la izquierda.
>=	Devuelve el valor <i>true</i> cuando el operando de la izquierda es mayor o igual que el de la derecha.
<=	Devuelve el valor <i>true</i> cuando el operando de la derecha es menor o igual que el de la izquierda.

```

let a=4;b=5,c="5";
console.log("El resultado de la expresión 'a==c' es igual a "+(a==c));
console.log("El resultado de la expresión 'a===c' es igual a "+(a===c));
console.log("El resultado de la expresión 'a!=c' es igual a "+(a!=c));
console.log("El resultado de la expresión 'a!==c' es igual a "+(a!==c));
console.log("El resultado de la expresión 'a==b' es igual a "+(a==b));
console.log("El resultado de la expresión 'a!=b' es igual a "+(a!=b));
console.log("El resultado de la expresión 'a>b' es igual a "+(a>b));
console.log("El resultado de la expresión 'a<b' es igual a "+(a<b));
console.log("El resultado de la expresión 'a>=b' es igual a "+(a>=b));
console.log("El resultado de la expresión 'a<=b' es igual a "+(a<=b));

```

5.4 Operadores lógicos

Los operadores lógicos se utilizan para el procesamiento de los valores booleanos. A su vez el valor que devuelven también es booleano: true o false.

Operador	Descripción
&&	Y "lógica". El valor de devolución es true cuando ambos operandos son verdaderos.
	O "lógica". El valor de devolución es true cuando alguno de los operandos es verdadero o lo son los dos.
!	No "lógico". Si el valor es true, devuelve false y si el valor es false, devuelve true.

Se muestra el resultado de distintas operaciones realizadas con operadores lógicos. (En el ejemplo se usa directamente los valores true y false en lugar de variables).

```
console.log("El resultado de la expresión 'false&&false' es igual a  
"+(false&&false));  
console.log("El resultado de la expresión 'false&&true' es igual a  
"+(false&&true));  
console.log("El resultado de la expresión 'true&&false' es igual a  
"+(true&&false));  
console.log("El resultado de la expresión 'true&&true' es igual a  
"+(true&&true));  
console.log("El resultado de la expresión 'false||false' es igual a  
"+(false||false));  
console.log("El resultado de la expresión 'false||true' es igual a  
"+(false||true));  
console.log("El resultado de la expresión 'true||false' es igual a  
"+(true||false));  
console.log("El resultado de la expresión 'true||true' es igual a  
"+(true||true));  
console.log("El resultado de la expresión '!false' es igual a  
"+(!false));
```

Para saber más de comparadores y expresiones, puedes consultar:

[https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Expressions and Operator
s](https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Expressions_and_Operator_s)

6. ESTRUCTURAS DE CONTROL

6.1 Instrucciones if/else

Para controlar el flujo de información en los programas JavaScript existen una serie de estructuras condicionales y bucles que permiten alterar el orden secuencial de ejecución. Estas son las instrucciones if y else.

La instrucción if permite la ejecución de un bloque u otro de instrucciones en función de una condición.

Sintaxis

```
if (condición) {  
  // bloque de instrucciones que se ejecutan si la condición se cumple  
}  
else{  
  // bloque de instrucciones que se ejecutan si la condición no se cumple  
}
```

Las llaves solo son obligatorias cuando haya varias instrucciones seguidas pertenecientes a la ramificación. Si no pones llaves, el if se aplicará únicamente a la siguiente instrucción.

Puede existir una instrucción if que no contenga la parte else. En este caso, se ejecutarán una serie de órdenes si se cumple la condición y si esto no es así, se continuaría con las órdenes que están a continuación del bloque if. Ejemplos de uso:

```
let diaSem;  
diaSem=prompt("Introduce el día de la semana ", "");  
if (diaSem === "domingo")  
{  
  console.log("Hoy es festivo");  
}  
else // Al no tener {}, es un "bloque de una instrucción"  
  console.log("Hoy no es domingo, a descansar!!");
```

Otro ejemplo

```
let edadAna,edadLuis;  
// Usamos parseInt para convertir a entero  
edadAna=parseInt(prompt("Introduce la edad de Ana",""));  
edadLuis=parseInt(prompt("Introduce la edad de Luis",""));  
  
if (edadAna > edadLuis){
```

```
    console.log("Ana es mayor que Luis.");
    console.log(" Ana tiene "+edadAna+" años y Luis "+ edadLuis);
}
else{
    console.log("Ana es menor o de igual edad que Luis.");
    console.log(" Ana tiene "+edadAna+" años y Luis "+ edadLuis);
}
```

6.2 Ramificaciones anidadas

Para las condiciones ramificadas más complicadas, a menudo se utilizan las ramificaciones anidadas. En ellas se definen consultas if dentro de otras consultas if, por ejemplo:

```
let edadAna,edadLuis;
// Convertimos a entero las cadenas
edadAna=parseInt(prompt("Introduce la edad de Ana",""));
edadLuis=parseInt(prompt("Introduce la edad de Luis",""));
if (edadAna > edadLuis){
    console.log("Ana es mayor que Luis.");
}
else{
    if (edadAna<edadLuis){
        console.log("Ana es menor que Luis.");
    }else{
        console.log("Ana tiene la misma edad que Luis.");
    }
}
console.log(" Ana tiene "+edadAna+" años y Luis "+ edadLuis);
```

7. ESTRUCTURA REPETITIVAS (BUCLES)

7.1 Bucle for

Cuando la ejecución de un programa llega a un bucle for:

- Lo primero que hace es ejecutar la “Inicialización del índice”, que solo se ejecuta una vez.
- A continuación analiza la condición de prueba y si esta se cumple ejecuta las instrucciones del bucle.
- Cuando finaliza la ejecución de las instrucciones del bucle se realiza la modificación del índice, se retorna a la cabecera del bucle for y se realiza de nuevo la condición de prueba.
- Si la condición se cumple se ejecutan las instrucciones y si no se cumple la ejecución continúa en las líneas de código que siguen posteriores al bucle.

Sintaxis

```
for (Inicialización del índice; Condición de prueba; Modificación en el índice){  
  // ...instrucciones...  
}
```

Ejemplo: números pares del 2 al 30

```
for (i=2;i<=30;i+=2) {  
  console.log(i);  
}  
console.log("Se han escrito los números pares del 2 al 30");
```

Ejemplo: Escribe las potencias de 2 hasta 3000

```
let aux=1;  
for (i=2;i<=3000;i*=2) {  
  console.log("2 elevado a "+aux+" es igual a "+i);  
  aux++;  
}  
console.log("Se han escrito las potencias de 2 menores de 3000");
```

7.2 Bucle while

Con el bucle while se pueden ejecutar un grupo de instrucciones mientras se cumpla una condición.

- Si la condición nunca se cumple, entonces tampoco se ejecuta ninguna instrucción.
- Si la condición se cumple siempre, nos veremos inmersos en el problema de los bucles infinitos, que pueden llegar a colapsar el navegador, o incluso el ordenador.
 - Por esa razón es muy importante que la condición deba dejar de cumplirse en algún momento para evitar bucles infinitos.

Sintaxis

```
while (condición){  
  //...instrucciones...  
}
```

Ejemplo: Escribe los números pares de 0 a 30

```
let i=2;
while (i<=30) {
    console.log(i);
    i+=2;
}
console.log("Ya se han mostrado los números pares del 2 al 30");
```

Ejemplo: Pregunta una clave hasta que se corresponda con una dada.

```
let auxclave="";
while (auxclave!="vivaY0"){
    auxclave=prompt("introduce la clave ", "claveSecreta")
}
console.log("Has acertado la clave");
```

7.3 Bucle do-while

La diferencia del bucle do-while frente al bucle while reside en el momento en que se comprueba la condición: el bucle do-while no la comprueba hasta el final, es decir, después del cuerpo del bucle, lo que significa que el bucle do-while se recorrerá, una vez, como mínimo, aunque no se cumpla la condición.

Sintaxis

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

Ejemplo: Preguntar por una clave hasta que se introduzca la correcta

```
let auxclave;
do {
    auxclave=prompt("introduce la clave ", "vivaYo")
} while (auxclave!="EstaeslaclaveJEJEJE")
console.log("Has acertado la clave");
```

7.4 Instrucciones BREAK y CONTINUE

En los bucles for, while y do-while se pueden utilizar las instrucciones break y continue para modificar el comportamiento del bucle.

La instrucción “break” dentro de un bucle hace que este se interrumpa inmediatamente, aun cuando no se haya ejecutado todavía el bucle completo. Al llegar la instrucción, el programa se sigue desarrollando inmediatamente a continuación del final del bucle.

Ejemplo: Pregunta por la clave y permitir tres respuestas incorrectas

```
let auxclave=true;
let numveces=0;
//Mientras no introduzca la clave y no se pulse Cancelar
while (auxclave !== "anonimo" && auxclave){
    auxclave=prompt("Introduce la clave ", "");
    numveces++;
    if (numveces === 3)
        break;
}
if (auxclave=="SuperClave"){
    console.log("La clave es correcta");
}else{
    console.log("La clave no es correcta correcta");
}
```

El efecto que tiene la instrucción “continue” en un bucle es el de hacer retornar a la secuencia de ejecución a la cabecera del bucle, volviendo a ejecutar la condición o a incrementar los índices cuando sea un bucle for. Esto permite saltarse recorridos del bucle.

Ejemplo: Presenta todos los números pares del 0 al 50 excepto los que sean múltiplos de 3

```
let i;
for (i=2;i<=50;i+=2){
    if ((i%3)===0)
        continue;
    console.log(i);
}
```

8. FUNCIONES

Una función es un conjunto de instrucciones que se agrupan bajo un nombre de función. Se ejecuta solo cuando es llamada por su nombre en el código del programa. La llamada provoca la ejecución de las órdenes que contiene.

Las funciones son muy importantes por diversos motivos:

- Ayudan a estructurar los programas para hacerlos su código más comprensible y más fácil de modificar.
- Permiten repetir la ejecución de un conjunto de órdenes todas las veces que sea necesario sin necesidad de escribir de nuevo las instrucciones.

Una función consta de las siguientes partes básicas:

- Un nombre de función.
- Los parámetros pasados a la función separados por comas y entre paréntesis.
- Las llaves de inicio y final de la función.
- Desde Javascript ES6, se pueden definir valores por defecto para los parámetros.

Sintaxis de la definición de una función

```
function nombrefuncion (parámetro1, parámetro2=valorPorDefecto...){  
  // instrucciones  
  //si la función devuelve algún valor añadimos:  
  return valor;  
}
```

Sintaxis de la llamada a una función

```
// La función se ejecuta siempre que se ejecute la sentencia.  
valorRetornado=nombrefuncion (parám1, parám2...);
```

Es importante entender la **diferencia entre definir una función y llamarla:**

- Definir una función es simplemente especificar su nombre y definir qué acciones realizará en el momento en que sea invocada, mediante la palabra reservada function.
- Para llamar a una función es necesario especificar su nombre e introducir los parámetros que queremos que utilice. Esta llamada se puede efectuar en una línea de órdenes o bien a la derecha de una sentencia de asignación en el caso de que la función devuelva algún valor debido al uso de la instrucción return.

La definición de una función se puede realizar en cualquier lugar del programa, pero se recomienda hacerlo al principio del código o en un fichero "js" que contenga funciones.

La llamada a una función se realizará cuando sea necesario, es decir, cuando se demande la ejecución de las instrucciones que hay dentro de ella.

Ejemplo: funciones que devuelve la suma de dos valores que se pasan por parámetros y que escriben el nombre del profesor.

```
// Definiciones de Las funciones
function suma (a,b){
    // Esta función devuelve un valor
    return a+b;
}

// Esta función muestra un texto, pero no devuelve un valor
function profe (){
    console.log("El profesor es muy bueno");
    // OJO: esto es un ejemplo, pero rara vez se realiza en una función real
}

// Código que se ejecuta
let op1=5, op2=25;
let resultado;
// Llamada a función
resultado=suma(op1,op2);
// Llamada a la función
console.log (op1+" "+op2+"="+resultado);
// Llamada a función
profe();
```

⚠ **Atención:** recordad que dentro de las funciones rara vez se utilizan funciones de entrada/salida. El 99.9% de las veces simplemente procesan la entrada por parámetros y devuelven un valor.

Desde Javascript ES6, las funciones soportan los llamados parámetros REST.

Los parámetros REST son un conjunto de parámetros que se almacenan como array en un “parámetro final” nombrado con **...nombreParametro**. Esto nos permite manejar la función sin tener que controlar el número de parámetros con los que esta es llamada.

Importante: sólo el último parámetro puede ser REST.

Ejemplo:

```
function pruebaParREST(a, b, ...masParametros) {  
  console.log("a: "+a+" b: "+ b + " otros: " + masParametros);  
}  
pruebaParREST("param1", "param1", "param3", "param4", "param5");
```

Para saber más de los parámetros REST

https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Funciones/parametros_rest

9. FUNCIONES FLECHA (ARROW FUNCTIONS)

Una función flecha (arrow function) es una alternativa compacta al uso de funciones tradicionales.

- Este tipo de funciones tienen sus limitaciones y deben ser utilizadas solo en algunos contextos donde sean útiles.
- No son adecuadas para ser utilizadas como métodos.

Soporta varias formas de sintaxis, a continuación indicamos las más típicas:

- (parametro1, parametro2, ...) => {sentencias}
- () => {sentencias}
- parámetro => sentencia

A veces, son utilizados con la **función “map” de los arrays**. Esta función crea un nuevo array formado por la aplicación de una función a cada uno de sus elementos:

https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/map

También es muy útil con la **función “reduce” de los arrays**. Esta función ejecuta una “función reductora” sobre cada elemento del array, devolviendo un único valor.

https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/reduce

Ejemplo:

```
let nombres = ['Pedro', 'Juan', 'Elena'];  
console.log(nombres.map(nom => nom.length));  
// Muestra el array con los valores [5, 4, 5]  
let sumaNombres= nombres.reduce((acumulador, elemento) => {  
  return acumulador + elemento.length;  
}, 0);  
// Muestra la suma de la longitud de los nombres  
console.log(sumaNombres);
```


Para saber más:

https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Funciones/Arrow_functions

10. CLASES EN JAVASCRIPT

Javascript ES6 permite una mejor definición de clases que en anteriores versiones de Javascript.

Mediante la palabra reservada “class”, permite definir clases, métodos, atributos, etc...

Recordad que los objetos en Javascript se guardan como referencias de memoria. En apartados anteriores hablamos de como clonar arrays y objetos.

Ejemplo:

```
class Punto {
  // Constructor de la clase
  constructor ( pX , pY ){
    this.pX = pX;
    this.pY = pY;
  }
  // Método estático para calcular distancia entre dos puntos
  static distancia ( a , b ) {
    const dx = a.pX - b.pX;
    const dy = a.pY - b.pY;

    return Math.sqrt ( dx * dx + dy * dy );
  }
  // Método indicado para ser usado como getter
  get coordX() {
    return this.pX;
  }
  // Método normal
  devuelveXporY () {
    return this.pX * this.pY;
  }
}
let p1 = new Punto(5, 5);
let p2 = new Punto(10, 10);
//Llamada método estático
console.log (Punto.distancia(p1, p2));
//Llamada método normal
console.log (p1.devuelveXporY());
// Al ser un getter, puede usarse como una propiedad
```

```
console.log (p1.coordX);
```

Para saber más: <https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Classes>

11. MÁS COSAS INTERESANTES DE JAVASCRIPT ES6

Este documento pretende ser una simple guía y no podemos incluir todas las novedades de Javascript ES6, pero desde aquí enlazamos las más interesantes por si queréis ampliar:

- Iteradores y generadores
 - https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Iterators_and_Generators
- Operadores de propagación (spread)
 - https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Operadores/Spread_operator
- Plantillas de cadenas de texto (template strings)
 - https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/template_strings
- For ... of
 - <https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Sentencias/for...of>
- Tail Call Optimization (hace más seguras funciones recursivas)
 - <https://hackernoon.com/es6-tail-call-optimization-43f545d2f68b>

Aquí tenéis otras referencias a nuevas características de Javascript ES6

<https://github.com/lukehoban/es6features>

12. INTRODUCCIÓN A LA MANIPULACIÓN DEL DOM

Con el fin de facilitar la realización de pequeños ejercicios, incluimos unos primeros conceptos de manipulación del DOM (Document Object Model). Conociendo la id de algún objeto HTML existente en la página podemos cambiar sus propiedades.

Cuando queremos cambiar una propiedad de un objeto presente en el documento, utilizamos su ID (nombre único que define a un elemento) y el método "document.getElementById()" para acceder al elemento en sí.

Ejemplo:

Para cambiar una imagen con id 'matrix', accedemos al elemento y modificamos la propiedad src :

```
document.getElementById('matrix').src = "mt05.jpg";
```

Esto se puede usar para cualquier propiedad existente en cualquier objeto HTML.

Una función Javascript genérica para cambiar una imagen sería:

```
function cambiarImagen (id, rutaImagen) {  
    document.getElementById(id).src=rutaImagen;  
}
```

13. MATERIAL ADICIONAL

[1] Curso de Git en Udacity

<https://www.udacity.com/course/how-to-use-git-and-github--ud775>

[2] Uso de la "Consola Web"

https://developer.mozilla.org/en-US/docs/Tools/Web_Console

14. BIBLIOGRAFÍA

[1] Javascript Mozilla Developer <https://developer.mozilla.org/es/docs/Web/JavaScript>

[2] Javascript ES6 W3C https://www.w3schools.com/js/js_es6.asp