# Sintaxis de Javascript

Como cualquier lenguaje, Javascript está formado por unos **elementos** determinados. Es lo que llamamos la **sintaxis de Javascript**, y sería equivalente a hablar de artículos, preposiciones o verbos en nuestro idioma.

Concretamente, la **sintaxis de Javascript** está formada por los siguientes elementos:

- Valores.
- Operadores.
- Expresiones.
- Palabras reservadas.
- Comentarios.

Además, al igual que en nuestro lenguaje los nombres propios comienzan con mayúscula y los comunes con minúscula, y no podemos introducir números o caracteres extraños en una palabra, Javascript también tiene normas para escribir sus **identificadores**.

# Sentencias en Javascript

Al igual que en nuestro idioma es necesario dividir el texto en frases o párrafos y tienen mucha importancia los signos de puntuación, en **Javascript** tenemos que conocer cómo deben escribirse las instrucciones. Es lo que llamamos **sentencias en Javascript**:

- Es recomendable utilizar el **punto y coma al finalizar cada instrucción** (¡pero ojo, no necesariamente cada línea!).
- Podemos agrupar **varias instrucciones en la misma línea** separándolas por puntos y comas.
- Los **espacios en blanco** entre elementos hacen el código más legible.
- Los **bloques de código** se limitan con llaves.

- Hay una serie de **palabras reservadas** que no pueden ser utilizadas como identificadores.

# Comentarios en Javascript

Mediante **comentarios en Javascript** podemos explicar para qué sirve una función, qué utilidad tiene una variable, cómo se realiza una operación aritmética compleja, etc. Estos comentarios pueden no ser importantes en el momento en que son escritos, pero es posible que, en un futuro tú u otro programador debáis volver a revisar el código y **los comentarios en Javascript pueden aportar información interesante**… ¡o no!

Por tanto, **comentar nuestro código** es una práctica que, aunque tiene sus detractores, nos permite aportar información de valor sobre lo que estamos escribiendo.

Además, aunque es más útil utilizar un depurador, podemos **eliminar la funcionalidad de trocitos de código** comentándolos. De esta manera podemos testear partes de nuestros programas para **aislar un error**… ¡acorralarlo hasta que demos con él!

En cualquier caso, recuerda que si escribes un **buen código, limpio, legible, comprensible, intuitivo**… ¡utilizar comentarios puede no ser necesario!

```
<!DOCTYPE html>
<html>
	<head>
		<meta charset="UTF-8">
	</head>

	<body>
	    <script>
		//Declaración de variables
		var a = 1; //Variable a
		var b = 2; //Variable b

		/*
		    Esta operación realiza
		    la suma de a y b
		*/
```

```
            //a = 2;

            var c = a + b;
            alert (c);
        </script>
    </body>
</html>
```

# Declaración de variables

Puedes declarar una variable de dos formas:

- Con la palabra clave var. Por ejemplo, var x = 42. Esta sintaxis se puede utilizar para declarar variables locales y globales, dependiendo del *contexto de ejecución*.
- A partir de la versión ES6 Con la palabra clave const o let. Por ejemplo, let y = 13. Esta sintaxis se puede utilizar para declarar una variable local con ámbito de bloque. (Ver el Ámbito de variables abajo.)

También puedes simplemente asignar un valor a una variable. Por ejemplo, x = 42.

El siguiente ejemplo crea una variable global no declarada. También genera una advertencia estricta de JavaScript. Las variables globales no declaradas a menudo pueden provocar un comportamiento inesperado. Por lo tanto, se desaconseja utilizar variables globales no declaradas.

**Evaluar variables**

Una variable declarada usando la instrucción var o let sin un valor asignado especificado tiene el valor de undefined.

Un intento de acceder a una variable no declarada da como resultado el disparo de una excepción ReferenceError:

```
var a;
console.log('El valor de a es ' + a); // El valor de a es
undefined

console.log('El valor de b es ' + b); // El valor de b es
undefined
var b;
// Esto puede desconcertarte hasta que leas 'Elevación de
variable' a continuación

console.log('El valor de c es ' + c); // Error de
referencia no detectado: c no está definida

let x;
console.log('El valor de x es ' + x); // El valor de x es
undefined

console.log('El valor de y es ' + y); // Error de
referencia no detectada: y no está definida
let y;
```

Puedes usar undefined para determinar si una variable tiene un valor. En el siguiente código, a la variable input no se le asigna un valor y la declaración if evalúa a true.

```
var input;
if (input === undefined) {
  doThis();
} else {
  doThat();
}
```

El valor undefined se comporta como false cuando se usa en un contexto booleano. Por ejemplo, el siguiente código ejecuta la función myFunction porque el elemento myArray es undefined:

```
var myArray = [];
if (!myArray[0]) myFunction();
```

El valor undefined se convierte en NaN cuando se usa en contexto numérico.

```
var a;
console.log(a + 2);  // Evalúa a NaN
```

Cuando evalúas una variable null, el valor nulo se comporta como 0 en contextos numéricos y como false en contextos booleanos. Por ejemplo:

```
var n = null;
console.log(n * 32); // Registrará 0 en la consola
```

**Ámbito de variables**

Cuando declaras una variable fuera de cualquier función, se denomina variable *global*, porque está disponible para cualquier otro código en el documento actual. Cuando declaras una variable dentro de una función, se llama variable *local*, porque solo está disponible dentro de esa función.

JavaScript anterior a ECMAScript 2015 (ES6) no tiene el ámbito de la declaración de bloque. Más bien, una variable declarada dentro de un bloque es local a la *función* en el que reside el bloque.

Por ejemplo, el siguiente código registrará 5, porque el ámbito de x es el contexto global (o el contexto de la función si el código es parte de una función). El ámbito de x no se limita al bloque de instrucciones inmediato.

```
if (true) {
  var x = 5;
}
console.log(x);  // x es 5
```

Este comportamiento cambia cuando se usa la declaración let (introducida en ECMAScript 2015).

```
if (true) {
  let y = 5;
}
console.log(y); // ReferenceError: y no está definida
```

```
<!DOCTYPE html>
<html>
     <head>
          <meta charset="UTF-8">
          <title>Ámbito de variables</title>
     </head>

     <body>
          <script>
               //VARIABLES LOCALES
               /* - Se define dentro de una función.
                  - Tienen ámbito local en la función.
                  - Son accesibles únicamente dentro de la
función.
                  - Podemos declarar variables con el mismo
nombre en diferentes funciones.
                  - Variable local desaparece cuando
finaliza la función. */
                  function saludar(){
                    var saludo = "Hola";
                    console.log(saludo);
```

```
                }
                saludar();

            //VARIABLES GLOBALES
            /* - Se definen fuera de las funciones.
                - Tienen ámbito global (en toda la
página).
                - Son accesibles desde fuera y dentro de
las funciones.
                - Variables globales desaparecen cuando
se sale de la página. */
                var despedida = "Adios";
                function despedir(){
                  console.log (despedida);
                }
                despedir();

            //VARIABLES AUTOMÁTICAMENTE GLOBALES
            // - Si asignamos un valor a una variable no
declarada se convierte en global.
            function preguntar(){
                 pregunta = "¿De qué color es el caballo
blanco de Santiago?";
            }
            preguntar();
            console.log(pregunta);

            //VARIABLES LOCALES Y GLOBALES CON EL MISMO
NOMBRE
            var miVariable = "Fuera";
            function ambito(){
                 var miVariable = "Dentro";
                 console.log(miVariable);
            }
            console.log(miVariable); //Devuelve Fuera
            ambito(); //Devuelve Dentro
            console.log(miVariable); //Devuelve Fuera
```

```
            //VARIABLE GLOBAL REDEFINIDA DENTRO DE UNA
FUNCIÓN
            var miVariable2 = "Fuera";
            function ambito2(){
                miVariable2 = "Dentro";
                console.log(miVariable2);
            }
            console.log(miVariable2); //Devuelve Fuera
            ambito2(); //Devuelve Dentro
            console.log(miVariable2); //Devuelve Dentro
        </script>
    </body>
</html>
```

**Elevación de variables (hoisting)**

https://www.freecodecamp.org/espanol/news/que-es-hoisting-alzar-en-javascript/

Otra cosa inusual acerca de las variables en JavaScript es que puedes hacer referencia a una variable declarada más tarde, sin obtener una excepción.

Este concepto se conoce como elevación. Las variables en JavaScript son, en cierto sentido, "elevadas" (o "izadas") a la parte superior de la función o declaración. Sin embargo, las variables que se elevan devuelven un valor de undefined. Entonces, incluso si la declaras e inicias después de usarla o hacer referencia a esta variable, devuelve undefined.

```
/**
 * Ejemplo 1
 */
console.log(x === undefined); // true
var x = 3;
```

```
/**
 * Ejemplo 2
 */
// devolverá un valor de undefined
var myVar = 'my value';

(function() {
  console.log(myVar); // undefined
  var myVar = 'valor local';
})();
```

Los ejemplos anteriores se interpretarán de la misma manera que:

```
/**
 * Ejemplo 1
 */
var x;
console.log(x === undefined); // true
x = 3;

/**
 * Ejemplo 2
 */
var myVar = 'my value';

(function() {
  var myVar;
  console.log(myVar); // undefined
  myVar = 'valor local';
})();
```

Debido a la elevación, todas las declaraciones var en una función se deben colocar lo más cerca posible de la parte superior de la función. Esta buena práctica aumenta la claridad del código.

En ECMAScript 2015, let y const se elevan pero no se inician. Hacer referencia a la variable en el bloque antes de la declaración de la variable da como resultado un ReferenceError, porque la variable está en una "zona muerta temporal" desde el inicio del bloque hasta que se procesa la declaración.

```
console.log(x); // ReferenceError
let x = 3;
```

# Modo estricto con use strict

El modo estricto, expresado como ***"use strict"***, es una expresión literal que se puede utilizar desde la versión de **ES6** y que, entre otras cosas, **no permite utilizar variables no declaradas**.

```
"use strict";
x = 3.14;       // This will cause an error because x is
not declared
```

Podríamos decir que, hasta ahora, hemos utilizado Javascript en modo "poco riguroso", o *"sloppy mode".* Es más, ahora podemos utilizar el **modo estricto** y el **modo poco riguroso** en el mismo programa, e incluso ir progresivamente cambiando nuestros programas de modo poco riguroso a estricto.

La expresión literal del **modo estricto** se define de la siguiente manera al comienzo de un script o una función.

Si se declara **dentro de una función** tendrá ámbito local (solo el código que está dentro de la función estará en modo estricto); si se declara **al principio de un script** tendrá ámbito global (todo el código del script se ejecutará en **modo estricto**).

```
x = 3.14;       // This will not cause an error.
myFunction();

function myFunction() {
  "use strict";
  y = 3.14;   // This will cause an error
}
```

```
"use strict";
myFunction();

function myFunction() {
  y = 3.14;   // This will also cause an error because y is not declared
}
```

https://www.w3schools.com/js/js_strict.asp

# Operadores y tipos de datos

https://www.w3schools.com/js/js_operators.asp
https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Expressions_and_Operators

https://manuais.iessanclemente.net/index.php/Tipos_de_datos_en_JavaScript

Operadores Aritméticos

Los **operadores aritméticos**, como su propio nombre indica, permiten realizar **operaciones matemáticas aritméticas**, es decir, las operaciones básicas: **suma, resta, multiplicación y división**. Además, en programación trabajamos con un operador muy interesante llamado módulo que nos sirve para obtener el **resto de una división entera**.

Los operadores aritméticos en Javascript pueden dividirse en dos grupos: **binarios** (requieren de dos operandos) o **unarios** (se aplican sobre un solo operando).

Por tanto, los **operadores aritméticos binarios** con los que podemos trabajar en Javascript son:

- Operador suma: +.
- Operador resta: -.
- Operador producto: *
- Operador división: /
- Operador módulo o resto: %

**operadores aritméticos unarios** son:

- Operador incremento: ++
- Operador decremento: --

**operadores de desplazamiento de bits(bitwise operators):**

| | | |
|---|---|---|
| Desplazamiento a la izquierda | a << b | Desplaza a en representación binaria b bits hacia la izquierda, desplazándose en ceros desde la derecha.(cada desplazamiento equivale a multiplicar por 2) |
| Desplazamiento a la derecha de propagación de signo | a >> b | Desplaza a en representación binaria b bits a la derecha, descartando los bits desplazados.(cada desplazamiento equivale a dividir por 2) |

Pero no es el único tipo de operadores que existen. También hay operadores de asignación, de cadena, de comparación, lógicos…

```
<!DOCTYPE html>
<html>
     <head>
          <meta charset="UTF-8">
          <title>Operadores aritméticos en
Javascript</title>
```

```
    </head>

    <body>
        <script>
        // OPERADORES ARITMÉTICOS: +, -, *, /, %, ++, --.

        //UTILIZACION
        var x = 5; var y = 2;
        var dosNumeros = 1 + 7;
        var dosVariables = x + y;
        var numeroVariable = 4 + x;

        alert("La variable dosNumeros contiene
"+dosNumeros);
        alert("La variable dosVariables contiene
"+dosVariables);
        alert("La variable numeroVariable contiene
"+numeroVariable);

        //EJEMPLOS DE OPERADORES BINARIOS
        var suma = x + y;
        var resta = x - y;
        var multiplicacion = x * y;
        var division = x / y;
        var modulo = x % y;

        alert ("La suma es "+suma);
        alert ("La resta es "+resta);
        alert ("El producto es "+multiplicacion);
        alert ("La division es "+division);
        alert ("El modulo es "+modulo);

        //EJEMPLOS DE OPERADORES UNARIOS
        var a = 8;
        alert ("El valor inicial de a es "+a);
        ++a;
        alert ("El valor de a después del incremento es
```

```
"+a);
          --a;
          alert ("El valor de a después del decremento es
"+a);

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

Operadores de asignación

Los **operadores de asignación** se utilizan para dar un valor a una variable. Podríamos pensar que solamente necesitamos un operador de asignación para esta operación. ¡Nada más lejos de la realidad! Podemos utilizar este tipo de operadores asociados a operaciones aritméticas.

Los seis **operadores de asignación** más importantes de Javascript son:

- Operador de asignación.
- Operador de asignación de adicción.
- Operador de asignación de sustracción.
- Operador de asignación de multiplicación.
- Operador de asignación de división.
- Operador de asignación de resto.

Además, hay otros **operadores de asignación**: de exponenciación, de desplazamiento, asociados a operaciones lógicas, etc.

```
<!DOCTYPE html>
<html>
     <head>
          <meta charset="UTF-8">
          <title>Operadores de asignación en
Javascript</title>
     </head>
```

```
    <body>
        <script>
            //OPERADORES DE ASIGNACIÓN: =, +=, -=, *=,
/=, %=

            //EJEMPLO DE OPERADOR DE ASIGNACION =
            var x = 10;

            //EJEMPLO DE OPERADOR DE ASIGNACION +=
            x+=5; // x = x + 5
            alert ("El valor de x es "+x);

            //EJEMPLO DE OPERADOR DE ASIGNACION *=
            var y = 10;
            y*=5; // y = y * 5
            alert ("El valor de y es "+y);

            //EJEMPLO DE OPERADOR DE ASIGNACION %=
            var z = 10;
            z%=5;
            alert ("El valor de z es "+z);
        </script>
    </body>
</html>
```

Operadores de cadena

Hay multitud de funciones que permiten realizar **operaciones con cadenas de texto**. Pero operadores, lo que se dice operadores, podríamos hablar de dos: el **operador suma** y el **operador de asignación de adición**.

El **operador suma**, al igual que ocurriría en el caso de los operadores aritméticos, permite concatenar dos cadenas. ¿Qué es concatenar? Unir una cadena con otra, o unir todas las cadenas que queramos entre sí.

El **operador de asignación de adición**, tal y como ya vimos sobre operadores de asignación, permite realizar conjuntamente la operación de asignación y la operación de concatenación en una sola sentencia.

```
<!DOCTYPE html>
<html>
     <head>
          <meta charset="UTF-8">
          <title>Operadores de cadena</title>
     </head>

     <body>
          <script>
          //OPERADOR +
          var nombre = "Ada";
          var apellido = 'Lovelace';
          var completo = nombre + " " + apellido;
          alert (completo);

          //OPERADOR DE ASIGNACION +=
          var texto = "ABCD";
          texto += "EFGH"; //texto = texto + "EFGH";
          alert(texto);

          //SUMAR NÚMEROS Y CADENAS
          var resultado = 4 + 5 + "Ada Lovelace";
          alert (resultado);
          </script>
     </body>
</html>
```

## Operadores de comparación

Los **operadores de comparación**, como podéis imaginar, permiten comparar dos expresiones y **devuelven un valor booleano**, es decir,

devuelven ***true*** o ***false*** (**verdadero** o **falso**) que representa la relación de sus valores.

Podemos hablar de dos grupos de **operadores de comparación**: los **operadores de igualdad** y los **operadores relacionales**.

Los vemos en el siguiente ejemplo

```
<!DOCTYPE html>
<html>
     <head>
          <meta charset="UTF-8">
          <title>Operadores de comparación</title>
     </head>

     <body>
          <script>
               var x = 5; // Tipo numérico

               //Operador de igualdad (==)
               alert("x==8 es "+(x==8)); //Devuelve false
               alert("x==5 es "+(x==5)); //Devuelve true
               alert("x=='5' es "+(x=='5')); //Devuelve
true

               //Operador de igualdad estricta (===)
               alert("x===8 es "+(x===8)); //Devuelve false
               alert("x===5 es "+(x===5)); //Devuelve true
               alert("x==='5' es "+(x==='5')); //Devuelve
false

               //Operador de desigualdad (!=)
               alert("x!=8 es "+(x!=8)); //Devuelve true
               alert("x!=5 es "+(x!=5)); //Devuelve false
               alert("x!='5' es "+(x!='5')); //Devuelve
false
```

```
            //Operador de desigualdad estricta (!==)
            alert("x!==8 es "+(x!==8)); //Devuelve true
            alert("x!==5 es "+(x!==5)); //Devuelve false
            alert("x!=='5' es "+(x!=='5')); //Devuelve
true

            //Operador de mayor que (>)
            alert("x>8 es "+(x>8)); //Devuelve false

            //Operador de menor que (<)
            alert("x<8 es "+(x<8)); //Devuelve true

            //Operador de mayor o igual que (>=)
            alert("x>=8 es "+(x>=8)); //Devuelve false

            //Operador de menor o igual que (<=)
            alert("x<=8 es "+(x<=8)); //Devuelve true
        </script>
    </body>
</html>
```

## Operadores lógicos

Los **operadores lógicos** devuelven un valor booleano (***true*** o ***false***) en función de que se cumpla o no una condición. Sus operandos, a diferencia del caso de la mayoría de los operadores de comparación, también son lógicos.

Hay tres **operadores lógicos**:

- **AND (&&)**: devuelve *true* (cierto) si los dos operandos son verdaderos.
- **OR (||)**: devuelve *true* (cierto) si al menos un operador es verdadero.
- **NOT (!)**: solo se aplica a un operando. Devuelve *true* (cierto) si su operando es falso y *false* (falso) si su operando es cierto.

```
<!DOCTYPE html>
<html>
     <head>
          <meta charset="UTF-8">
          <title>Operadores lógicos en Javascript</title>
     </head>

     <body>
          <script>
               //OPERADOR AND / Y (&&)
               var x = 4, y = 2;

               var resultado = (x < 5 && y > 1);
//Resultado true
               resultado = (x > 5 && y > 1); //Resultado
false
               alert (resultado);

               //OPERADOR OR / O (||)
               var resultado2 = (x > 5 || y > 1);
//Resultado true
               alert (resultado2);

               //OPERADOR NOT / NO (!)
               var resultado3 = !(x==y); //Resultado true
               alert (resultado3);
          </script>
     </body>
</html>
```

Operador ternario

El **operador ternario** realmente, podríamos decir que es un operador que permite realizar una **sentencia condicional** (es decir, un **if-then-else**) en una sola línea.

```
<!DOCTYPE html>
<html>
     <head>
          <meta charset="UTF-8">
          <title>Operador ternario</title>
     </head>

     <body>
          <script>
               // (CONDICION) ? RESULTADO_CIERTO :
RESULTADO_FALSO
               var edad = 22;
               var puedeVotar = (edad > 18) ? "Puede votar"
: "No puede votar";
               alert (puedeVotar);

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

Operadores de tipos

- ***Typeof*** devuelve una cadena que indica el tipo de operando que hay después sin evaluarlo.
- ***Instanceof*** verifica si un objeto se corresponde con el tipo de dato indicado a continuación.

```
<!DOCTYPE html>
<html>
     <head>
          <meta charset="UTF-8">
          <title>Operadores de tipos en Javascript</title>
```

```
    </head>

    <body>
        <script>
            //TYPEOF: nos devuelve el tipo de dato de
una variable o dato
            var cadena = "Ada Lovelace"; //Devuelve
string
            var numero = 3.14; //Devuelve number
            var bool = true; //Devuelve boolean
            var lista = [1, 2, 3, 4]; //Devuelve object
            var registro = {nombre: "Ada", apellido:
"Lovelace"}; //Devuelve object
            var fecha = new Date; //Devuelve object
            var miFuncion = function miFun () {alert
("Hola");}; //Devuelve function
            var sinDefinir; //Devuelve undefined
            var vacia = null; //Devuelve object

            //INSTANCEOF: nos devuelve true si un objeto
es una instancia de otro especificado
            var animales = ["perro", "gato", "hamster"];
            alert (animales instanceof Object);
//Devuelve true
            alert (animales instanceof Array);
//Devuelve true
            alert (animales instanceof Number);
//Devuelve false
        </script>
    </body>
</html>
```

Tipos de datos

Los principales **tipos de datos** que tiene Javascript:

- ***string***, o cadena.
- ***number***, o numérico.
- ***boolean***, o booleano.
- ***array***, o vectores/matrices.
- ***object***, u objetos.

y también tenemos tres valores importantes:

- Valor no definido (***undefined***).
- Cadena vacía.
- Valor nulo (***null***).

```
<!DOCTYPE html>
<html>
     <head>
          <meta charset="UTF-8">
          <title>Tipos de datos en Javascript</title>
     </head>

     <body>
          <script>
               // CADENAS o STRINGS
               var cadena = "Ada";
               var cadena2 = ' Lovelace';
               var cadena3 = "Ada 'Lovelace'";

               // NÚMEROS
               var entero = 3;
               var decimal = 3.56;
               var cientifica = 12e-5;

               // BOOLEANOS
               var cierto = true;
               var falso = false;

               // ARRAY
               var lista = [1, "Hola", 6, "Caracola"];
```

```
                // OBJETOS
                var animal = {nombre: "Lola", tipo:
"Hamster", raza: "Ruso", edad:1};

                // OPERADOR typeof
                alert (typeof cadena); //Devuelve string
                alert (typeof cierto); //Devuelve boolean
                alert (typeof animal); //Tanto arrays como
objetos devuelve object

                // UNDEFINED
                var objetoSinDefinir;
                alert (objetoSinDefinir); //Devuelve
undefined

                // string
                var cadena4 = "";
                alert ("Cadena: "+cadena4 + " tipo: "+typeof
cadena4);

                // NULL
                var animal = null;
                alert ("Animal: "+animal+ " tipo: "+typeof
animal);

                //NULOS vs NO DEFINIDOS
                typeof undefined; //Devuelve undefined
                typeof null; //Devuelve objeto
                null === undefined; //Devuelve false
                null == undefined; //Devuelve true

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

# Funciones

https://www.w3schools.com/js/js_functions.asp
https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Functions

Introducción

Una **función** es un grupo de sentencias o instrucciones que realizan una tarea o calculan un valor. Para utilizar una función, en primer lugar hay que definirla en algún lugar de nuestro programa y posteriormente, llamarla.

Una **función** debe seguir unas normas para ser definida, es decir, siempre debemos escribirla utilizando los mismos elementos aunque esto puede variar en javascript. Lo normal es que tenga la siguiente estructura:

```
function <nombre de la funcion> (<parametros opcionales separados por
comas>) {
        <instrucciones>
}
```

Pero aunque puede parecer muy sencillo, las funciones tienen multitud de variantes:

- Podemos escribir **funciones sin parámetros.**
- Las **funciones** pueden devolver un dato o no.
- Podemos escribir una función **sin paréntesis** y guardarla en una variable.
- Podemos escribir una **función sin nombre** (función anónima).

```
<!DOCTYPE html>
<html>
     <head>
          <meta charset="UTF-8">
          <title>Funciones - Introducción</title>
```

```
    </head>

    <body>
        <script>
        //SINTAXIS DE UNA FUNCION
        //function nombre_funcion
(<parámetros_separados_por_comas) { ... }

        //Función sin parámetros
        function saludo (){
            return "¡Hola!";
        }

        //Los paréntesis () invocan a la función
        var resultado_saludo = saludo();
        alert (resultado_saludo);

        //Sin paréntesis se puede guardar la función en
una variable
        var resultado_saludo2 = saludo;
        alert(resultado_saludo2);

        //Función con parámetros
        function producto (a, b){
            return a * b;
        }

        //Los paréntesis () invocan a la función
        var resultado_producto = producto(3, 4);
        alert (resultado_producto);

        //Podemos introducir en el alert diréctamente la
función
        alert (producto(5,6));

        //Sin paréntesis se puede guardar la función en
una variable o mostrar
```

```
          alert (producto);

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

Funciones anónimas

Las **funciones anónimas** no tienen nombre. Esto quiere decir que no es necesario poner el nombre de la función después de ***function***.

Debido a esto, se pueden pasar a otras funciones o asignar a variables. Cuando una función anónima se asigna a una variable, el nombre de la variable es el que usamos para llamar a la función.

buena información aquí : [funciones anonimas autoinvocadas]

tendría este aspecto:

```
function (<parametros opcionales separados por comas>) {
        <instrucciones>
}
```

```
<!DOCTYPE html>
<html>
     <head>
          <meta charset="UTF-8">
          <title>Funciones anónimas </title>
     </head>

     <body>
          <script>
          //Sintaxis de una función no anónima
          //function nombre (<parámetros>) {
<instrucciones> }
```

```
        //SINTAXIS DE UNA FUNCIÓN ANÓNIMA
        var producto = function (a, b) { return a * b;};
        var resultado = producto(3,6);
        alert (resultado);

        //EL CONSTRUCTOR FUNCTION()
        var miFuncion = new Function ("a", "b", "return
a*b;");
        var resultado2 = miFuncion(5,7);
        alert (resultado2);

        //FUNCIONES ANÓNIMAS AUTOINVOCADAS
        (function () { alert ("¡Hola!");}());

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

### Arrow Functions

A partir de ES6 disponemos también de las funciones flecha (Arrow functions).

Se trata de reemplazar eliminar la palabra function y añadir => antes de abrir las llaves:

```
const func = function () {
  return "Función tradicional.";
};

const func = () => {
  return "Función flecha.";
};
```

las funciones flechas tienen algunas ventajas a la hora de simplificar código bastante interesantes:

- Si el cuerpo de la función sólo tiene una línea, podemos omitir las llaves ({}).
- Además, en ese caso, automáticamente se hace un return de esa única línea, por lo que podemos omitir también el return.
- En el caso de que la función no tenga parámetros, se indica como en el ejemplo anterior: () =>.
- En el caso de que la función tenga un solo parámetro, se puede indicar simplemente el nombre del mismo: e =>.
- En el caso de que la función tenga 2 ó más parámetros, se indican entre paréntesis: (a, b) =>.
- Si queremos devolver un objeto, que coincide con la sintaxis de las llaves, se puede englobar con paréntesis: ({name: 'Manz'}).

Por lo tanto, el ejemplo anterior se puede simplificar aún más:

```
const func = () => "Función flecha."; // 0 parámetros:
Devuelve "Función flecha"
const func = e => e + 1; // 1 parámetro: Devuelve el valor
de e + 1
const func = (a, b) => a + b; // 2 parámetros: Devuelve el
valor de a + b
```

Ejemplos:

html:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>ES6 - Arrow functions o funciones flecha</title>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width,
initial-scale=1">
  <script src="06_ES6_ArrowFunctions.js"></script>

</head>
```

```
<body>
    <h1>ES6 - Arrow functions o funciones flecha</h1>
</body>
</html>
```

Javascript:

```
// ARROW FUNCTIONS O FUNCIONES FLECHA

/* Las funciones flecha son una alternativa compacta a una
función convencional.
   No son adecuadas para ser utilizadas como métodos,
   y no pueden ser usadas como constructores. */

// SINTAXIS: (param1, param2, ..., paramN) => { sentencias
}
//           (param1, param2, ..., paramN) => expresion
//             () => {return expresion}

//          Paréntesis opcionales con un solo parámetro
//          (parametro) => {sentencias}
//          parametro => {sentencias}

//          Paréntesis obligatorios si la función no tiene
parámetros
//          () => {sentencias}

var miFuncion = function () {
  return new Date();
}

var miFuncion = () => new Date();

var arraysConcatenados = function (array1, array2) {
  return array1.concat(array2);
}
console.log(arraysConcatenados([1,2],[3,4,5]));
```

```
var arraysConcatenados2 = (array1, array2) =>
array1.concat(array2);
console.log(arraysConcatenados2([1,2],[3,4,5]));
```

En el siguiente enlace tienes más información sobre las funciones, una cuestión interesante es la diferencia del tratamiento del objeto this en las funciones normales y las arrow function que comentaremos más adelante:

https://lenguajejs.com/javascript/fundamentos/funciones/

El manejo de this con las funciones de flecha también es diferente en comparación con las funciones normales.

En resumen, con las funciones de flecha no hay vinculación de this.

En funciones normales, this representaba el objeto que llamaba a la función, que podría ser la ventana, el documento, un botón o lo que sea.

Con las funciones de flecha, this siempre representa el objeto que definió la función de flecha.

Echemos un vistazo a dos ejemplos para comprender la diferencia.

Ambos ejemplos llaman a un método dos veces, primero cuando se carga la página y nuevamente cuando el usuario hace clic en un botón.

El primer ejemplo usa una función normal y el segundo ejemplo usa una función de flecha.

El resultado muestra que el primer ejemplo devuelve dos objetos diferentes (ventana y botón) y el segundo ejemplo devuelve el objeto Header dos veces.

Ejemplo
Con una función regular, this representa el objeto que llamó a la función:

```
class Header {
  constructor() {
    this.color = "Red";
  }

//Regular function:
  changeColor = function() {
    document.getElementById("demo").innerHTML += this;
  }
}

const myheader = new Header();

//The window object calls the function:
window.addEventListener("load", myheader.changeColor);

//A button object calls the function:
document.getElementById("btn").addEventListener("click",
myheader.changeColor);
```

Ejemplo;
Con una función de flecha, this representa el objeto Header sin importar quién llamó a la función:

```
class Header {
  constructor() {
    this.color = "Red";
  }

//Arrow function:
  changeColor = () => {
    document.getElementById("demo").innerHTML += this;
  }
}

const myheader = new Header();


//The window object calls the function:
window.addEventListener("load", myheader.changeColor);
```

```
//A button object calls the function:
document.getElementById("btn").addEventListener("click",
myheader.changeColor);
```

## Funciones. Parámetros y argumentos

En muchas ocasiones hablamos indistintamente de **parámetros y argumentos** para referirnos a los valores que “pasamos” entre paréntesis en una función.

Los parámetros son los nombres que aparecen en la definición de una función. Por su parte, los argumentos son los valores que le pasamos (y que, por tanto, recibe) una función.

Javascript permite que el número de parámetros de una función sea diferente del número de argumentos que se le pasan.

```
<!DOCTYPE html>
<html>
     <head>
          <meta charset="UTF-8">
          <title>Parámetros de funciones </title>
     </head>

     <body>
          <script>
          /* PARÁMETROS Y ARGUMENTOS:
               - Función puede tener cero o más parámetros.
               - Parámetros son los nombres que aparecen en
la definición de una función.
               - Argumentos con los valores que pasamos a
(y que recibe) una función

               REGLAS DE LOS PARÁMETROS:
               - No se especifica el tipo de los
parámetros.
```

```
                - No se verifican los tipos de los
argumentos.
                - No se comprueba el número de los
argumentos recibidos. */

                //PARÁMETROS POR DEFECTO
                //Cuando llamamos a una función con menos
argumentos de los declarados. Los valores que faltan no
están definidos.
                function suma (a, b){
                     if (b === undefined)
                          b = 0;
                     return a + b;
                }

                var resultado = suma (4);
                alert (resultado);

                //PARÁMETROS POR EXCESO
                //Cuando llamamos a una función con más
argumentos de los que ha sido declarada. Los valores que
nos llegan pueden capturarse a través de un objeto
(incluido en la función) llamado arguments.
                function valores (){
                     alert ("El número de argumentos es
"+arguments.length);
                     for (var i=0; i < arguments.length;
i++){
                          alert ("Argumento
"+i+"="+arguments[i]);
                     }
                }
                valores (4, 6, 8, 2, 7, 5);
          </script>
     </body>
</html>
```

Parámetros REST

A partir de la version ES6 disponemos de los parametros REST, ver funcionamiento y diferencias con arguments aquí:

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Functions/rest_parameters

# Estructuras de control

Condicionales. Sentencia if-else

entencia condicional: **if – else**, que ejecuta una o más instrucciones si una condición se evalúa como verdadera. En caso de que se evalúe como falsa podemos indicar que se ejecuten otras instrucciones o nada.

La sintaxis de este tipo de estructuras es la siguiente:

```
if (condición) sentencia1 [else sentencia2]
```

Si quisiéramos incluir más de una sentencia o instrucción deberíamos utilizar las llaves:

```
if (condición) {
sentencia1;
sentencia2;
...} [else {
sentencia3;
sentencia4;
}]
//¡Recuerda que todo lo que hay entre corchetes es opcional!
```

```
<!DOCTYPE html>
<html>
      <head>
            <meta charset="UTF-8">
            <title>Condicionales. Sentencias if...else</title>
      </head>

      <body>
            <script>
                  //IF: especifica un bloque de código que se ejecuta si
una condición es cierta.
                  /* SINTAXIS:
                  if (<condicion>){
                        //Instrucciones si la condición es verdadera
                  } */

                  var nota = 8;
                  if (nota >= 5){
                        alert ("Has aprobado");
                  }

                  //ELSE: especifica un bloque de código que se ejecuta
si una condición es falsa.
                  /* SINTAXIS:
                  if (<condicion>){
                        //Instrucciones si la condición es verdadera
                  } else {
                        //Instrucciones si la condición es falsa
                  }*/

                  var nota2 = 4;
                  if (nota2 >=5){
                        alert ("Has aprobado");
                  } else {
                        alert ("Has suspendido");
                  }

                  //ELSE IF: especifica una nueva condición si la
primera es falsa.
                  /* SINTAXIS:
                  if (<condicion>){
                        //Instrucciones si la condición 1 es verdadera
                  } else if (<condicion 2>) {
                        //Instrucciones si la condición 1 es falsa y la
condición 2 es verdadera.
```

```
                  } else {
                        //Instrucciones si la condición 1 es falsa y la
condición 2 es falsa.
                  } */

                  var nota3 = 9;
                  if (nota3 < 5) {
                        alert ("Has suspendido");
                  } else if (nota3 == 5){
                        alert ("Has sacado un suficiente");
                  } else if (nota3 == 6){
                        alert ("Has sacado un bien");
                  } else if (nota3 == 7 || nota3 == 8){
                        alert ("Has sacado un notable");
                  } else {
                        alert ("Has sacado un sobresaliente");
                  }


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

Condicionales. Sentencia switch

La sentencia switch, que equivaldría a hacer varios ***if – else if*** anidados. ***switch*** evalúa una condición comparando el valor de esa expresión con instancias case, es decir, va comprobando cuál de esas opciones se corresponde con la expresión, y ejecuta las instrucciones asociadas..

La sintaxis de este tipo de estructuras es la siguiente:

```
switch (expresión) {
  case valor1:
    //Instrucciones ejecutadas cuando el resultado de expresión coincide
con el valor1
    [break;]
  case valor2:
    //Instrucciones ejecutadas cuando el resultado de expresión coincide
con el valor2
    [break;]
  ...
```

```
  case valorN:
    //Instrucciones ejecutadas cuando el resultado de expresión coincide
con valorN
    [break;]
  default:
    //Instrucciones ejecutadas cuando ninguno de los valores coincide
con el valor de la expresión
    [break;]
}
```

***break*** y ***default*** permiten salir de la estructura sin seguir comprobando casos y la segunda para indicar cualquier valor que no está recogido anteriormente.

```
<!DOCTYPE html>
<html>
	<head>
		<meta charset="UTF-8">
		<title>Condicionales. Sentencia switch</title>
	</head>

	<body>
		<script>
			/* SWITCH:
			- Se utiliza para hacer diferentes acciones basadas en
diferentes condiciones.
			- La condición se evalúa una única vez.
			- Si hay alguna coincidencia con algún valor se ejecuta el
bloque de código correspondiente.
			- Break: sale del bloque switch.
			- Default: se ejecuta en caso de que la expresión no coincide
con ningún valor.
			*/

			/* SINTAXIS
			switch (<expresión>){
				case valor_1:
					//Instrucciones para valor_1
					break;
				case valor_n:
					//Instrucciones para valor_n
					break;
				default:
					//Instrucciones en caso de que no se cumpla
ninguna condición.
			}*/

			var nota = 11;
			var mensaje = "";
			switch (nota){
				case 0:
```

```
                        case 1:
                        case 2:
                        case 3:
                        case 4:
                               mensaje = "Suspenso";
                               break;
                        case 5:
                               mensaje = "Suficiente";
                               break;
                        case 6:
                               mensaje = "Bien";
                               break;
                        case 7:
                        case 8:
                               mensaje = "Notable";
                               break;
                        case 9:
                        case 10:
                               mensaje = "Sobresaliente";
                               break;
                        default:
                               mensaje = "El valor no es válido";
                  }
                  alert (mensaje);

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

Repeticiones. Bucle for

Consiste en tres expresiones opcionales entre paréntesis y separadas por punto y coma. La primera indica desde qué número, la segunda la condición que debe cumplir dicho número, y por último, el incremento o decremento en cada iteración del bucle.

La sintaxis de esta estructura es la siguiente:

```
for ([expresion-inicial]; [condicion]; [expresion-final])
sentencia-o-bloque-de-sentencias
```

Javascript, además, tiene diferentes opciones que permiten eliminar alguna de las expresiones entre paréntesis. Así, podemos:

- **Omitir la expresión inicial**: declarando antes la variable.

- **Omitir la condición:** añadiendo dentro del bloque de sentencias la condición y una ruptura con ***break***.
- **Omitir la expresión final:** incrementando o decrementando la variable dentro del bloque de sentencias.

Además un **bucle for** puede incluir más de una variable. Es decir, podemos incrementar o decrementar más de una variable simultáneamente en cada iteración.

```
<!DOCTYPE html>
<html>
	<head>
		<meta charset="UTF-8">
		<title>Repeticiones. Bucle for</title>
	</head>

	<body>
		<script>
			/* SINTAXIS:
				for (<sentencia_1>; <sentencia_2>; <sentencia_3>){
					//Código que se repite
				}
				- SENTENCIA 1: se ejecuta antes del que empiece el
bloque de código a repetir.
				- SENTENCIA 2: define la condición por la que el bloque
de código se va a ejecutar.
				- SENTENCIA 3: se ejecuta después de que acabe el bloque
de código a repetir. */

				var i;
				for (i = 1; i <= 5; i++){
					alert ("i = "+i);
				}

				// VARIACIONES
				//1. Si omitimos la primera sentencia debemos haber
inicializado la variable con anterioridad.
				var j = 1;
				for ( ; j <=5; j++){
					alert ("j = "+j);
				}

				//2. Si omitimos la segunda sentencia, dentro del código
a repetir debemos incluir una instrucción break.
				var k;
				for (k=1; ;k++){
					alert ("k = "+k);
					if (k == 5){
						break;
					}
```

```
                        }

                        //3. Si omitimos la tercera sentencia, debemos
incrementar o decrementar el valor de la variable dentro del bucle.
                        var l;
                        for (l=1; l<=5; ){
                              alert ("l = "+l);
                              l++;
                        }

                        //4. Podemos trabajar con dos o más valores dentro del
bucle for
                        var m, n;
                        for (m=1, n=10; m<=10, n>=1; m++, n--){
                              alert ("m = "+m+", n = "+n);
                        }

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

## Repeticiones. Bucle for in

Esta vez veremos una pequeña variante del for, que sirve para recorrer las propiedades de un objeto

La sintaxis de este tipo de estructura es la siguiente:

```
for (variable in objeto) { ...
```

```
<!DOCTYPE html>
<html>
      <head>
            <meta charset="UTF-8">
            <title>Repeticiones. Bucle for in</title>
      </head>

      <body>
            <script>
                  //Definimos un objeto animal
                  var animal = {nombre:"Lola", tipo:"Hamster",
raza:"Ruso", edad:1};
```

```
                var propiedad;

                for (propiedad in animal){
                      alert(animal[propiedad]);
                }
           </script>
      </body>
</html>
```

veremos también más adelante el bucle **for …. of (a partir de ES6)**que permite recorrer strings, arrays, maps y sets.

Repeticiones. Bucle while

El **bucle *while*** ejecuta una o varias instrucciones mientras una condición se evalúe como verdadera. Esta condición se evalúa ANTES de ejecutar la sentencia

```
while (condicion)
{ sentencia; sentencia; ... }
```

```
<!DOCTYPE html>
<html>
     <head>
          <meta charset="UTF-8">
          <title>Repeticiones. Bucle while</title>
     </head>

     <body>
          <script>
               /* SINTAXIS WHILE:
                    while (<condicion>){
                         //Código a ejecutar
                    } */

               var i = 0;
               while (i < 10){
                    alert ("i = "+i);
```

```
                    i++;
               }

               for(i = 0; i < 10; ){
                    alert ("i = "+i);
                    i++;
               }

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

Repeticiones. Bucle do-while

El **bucle *do-while*** funciona exactamente igual que while, con la diferencia que, en este caso, la condición se evalúa DESPUÉS de ejecutar la sentencia.

```
do { sentencia; sentencia; ... }
while (condicion);
```

```
<!DOCTYPE html>
<html>
     <head>
          <meta charset="UTF-8">
          <title>Repeticiones. Bucle do while</title>
     </head>

     <body>
          <script>
               /*SINTAXIS DO WHILE:
                    do {
                         // Código a ejecutar
                    } while (<condición>); */

                    var i = 0;

                    do{
```

```
                        alert ("i = "+i);
                        i++;
                    } while (i < 10);

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

Saltos. Break y continue

***Break*** y ***continue*** son dos instrucciones que permiten alterar el flujo del programa, es decir: en lugar de que las sentencias se ejecuten una detrás de otra, permiten saltar a otro punto de nuestro código.

Concretamente, ***break*** termina el bucle actual o sentencia switch y transfiere el control del programa a la siguiente instrucción que se encuentra después de la instrucción de terminación de éstos elementos.

En el caso de ***continue***, termina la ejecución de las sentencias de la iteración actual del bucle actual o la etiqueta y continua la ejecución del bucle en la siguiente iteración.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Saltos. Sentencias break y
continue</title>
    </head>

    <body>
        <script>
        /* BREAK: salir o saltar fuera un bucle,
finalizándolo.
            CONTINUE: saltar hasta la siguiente iteración
del bucle; permite saltar una o más iteraciones. */
```

```
          // Ejemplo de bucle con break
            var i;
            for (i = 0; i < 10; i++){
               if (i == 3){
                    break;
               }
               alert ("i = "+i);
            }

          //Ejemplo de bucle con continue
            var j;
            for (j = 0; j < 10; j++){
               if (j == 3 || j == 5){
                    continue;
               }
               alert ("j = "+j);
            }

            /*ETIQUETAS: preceder una sentencia con el
nombre de la etiqueta seguido de dos puntos.
            - Break sin etiquetas se puede utilizar en
bucles o switch.
            - Break con etiqueta también se puede utilizar
en un bloque de sentencias etiquetado.
            - Continue: la etiqueta es opcional. Siempre
debe estar dentro un bucle.
             */

            //Ejemplo de break con etiquetas
            var animales = ["perro", "gato", "hamster"];

            ver_animales:
            {
               alert (animales[0]);
               alert (animales[1]);
               break ver_animales;
```

```
            alert (animales[2]);
        }

        //Ejemplo de continue con etiquetas
        var k;
        bucle:
            for (k = 0; k < 10; k++){
            if (k == 3 || k == 5){
                continue bucle;
            }
            alert ("k = "+k);
        }

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