

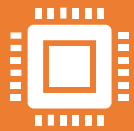
UNIDAD 2: Sintaxis de JavaScript

Desarrollo Web en Entorno Cliente

Índice

1. Introducción
2. Hola Mundo y comentarios
3. Variables, constantes y arrays
 1. Ámbito de Variables y constantes
 2. Variables
 3. Constantes
 4. Tipos de datos
 5. Coerción
 6. Arrays
 7. Conversiones entre tipos
 8. Constantes
 9. Modo Estricto
4. Entrada y salida
5. Operadores
 1. Asignación
 2. Aritméticos
 3. Comparación
 4. Lógicos
6. Estructuras de control
7. Estructuras repetitivas
8. Funciones
 1. Función flecha
9. Clases

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 por muchos Javascript ES6 (ECMAScript 6), también llamado ECMAScript 2015 o incluso por algunos llamado directamente Javascript 6.



Introducción /2

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.

Versionados: 1 al año

- https://en.wikipedia.org/wiki/ECMAScript_version_history

VERSIONES

- 1.1 4th Edition (abandoned)
- 1.2 5th Edition – ECMAScript 2009
- 1.3 6th Edition – ECMAScript 2015
- 1.4 7th Edition – ECMAScript 2016
- 1.5 8th Edition – ECMAScript 2017
- 1.6 9th Edition – ECMAScript 2018
- 1.7 10th Edition – ECMAScript 2019
- 1.8 11th Edition – ECMAScript 2020
- 1.9 12th Edition – ECMAScript 2021
- 1.10 13th Edition – ECMAScript 2022
- 1.11 EC Next

Cómo ejecutarlo...

- 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.
- Si el código referencia HTML que no está cargado, no funcionará cuando se ejecute
- Pueden haber tantos `<script>` como queramos; se suele recomendar su carga al final del `<body>` para no enlentecer la carga visual de la página

```
<script LANGUAGE= "JavaScript" >
```

Aquí va el código

```
//Esto es un comentario en JavaScript de una línea
```

```
</script>
```

Comentarios y mensajes a consola

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

- Ejemplo con “console.log” y comentario multilínea.

```
<script LANGUAGE= "JavaScript" >  
    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 LANGUAGE= "JavaScript" >  
    alert("Hola mundo");  
</script>
```

Inclusión en un fichero externo al HTML

- Se pueden incluir uno o varios ficheros con código Javascript en nuestro documento HTML. Se puede incluir tantos como se desee.
 - Es la forma más adecuada para trabajar con código Javascript.

```
<script type="text/javascript"  
src="rutaDelArchivo1.js"/>  
<script type="text/javascript"  
src="rutaDelArchivo2.js"/>
```

Variables, constantes y arrays

- Es recomendable mantener un estilo de nombramiento de variables.
 - Durante el curso, usaremos el estilo “CamelCase”
- 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.

Declaración y Ambito:

- Dos formas de declararlas
 - `let` : permite declarar una variable que sea accesible únicamente dentro del bloque donde se ha declarado (delimitado por `{ }`).
 - `var` : permite declarar una variable que sea accesible por todos los lugares de la función donde ha sido declarada.
- Variables sin declarar:
 - Javascript nos permite usar variables no declaradas.
- Ámbito:
 - Si una variable con `var` se declara fuera de cualquier función, el ámbito de esta son todas las funciones del código.

Ejemplo de Ambito

```
function ejemplo (){
    ejemplo= 3 ; // Variable no declarada
    if (ejemplo === 3 ){
        var variable1 = 1;
        let variable2 = 2;
    }
    console.log(variable1); // variable1
    existe aquí
    console.log(variable2); // variable2 no
    existe aquí
}
```

Principales Tipos de Datos

Numéricos (tipo "number"):

- Contiene 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} -
- Se pueden nomenciar escribiendo una letra "n" al final del entero.
- No pueden utilizarse con la mayoría de operadores matemáticos de Javascript.

Booleanos (tipo "boolean"):

- Contiene 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","laO").

- Para concatenar cadenas puede usarse el operador "+"

Comprobación del tipo de una variable

- `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);
```



Otros tipos


- Javascript considera primitivos:
 - “undefined”, “null”, “symbol”, “object” y “function”
 - Todos los valores que NO son de un tipo básico, son considerados objetos: arrays, funciones,
 - Esta distinción es muy importante
 - Los valores primitivos y los valores objetos se comportan de distinta forma cuando son asignados y cuando son pasados como parámetro a una función. valores compuestos, etc.
-

Coerción

- Javascript es un lenguaje de tipado blando
 - Al declarar una variable no se le asigna un tipo, aunque internamente Javascript si maneje 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, ¿Dónde ocurre la Coerción?

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

Aquí se convierte el numero a string



Ejemplo de Coerción

- Por ejemplo: ¿Que saldría con esta suma?

```
let a = "2", b = 5 ;  
console.log( typeof a + " " + typeof b);  
console.log( a + b ); //¿?
```

- 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
-

Reglas de Coerción

- 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`.

Ejemplos de Coerción

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 `!==`.

- 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
```

Arrays

- Un array (también llamado vector) 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

Ejemplos de Arrays

```
// Array definido 1 a 1
let miVector=[];
let miVector2=new Array();
miVector[0]= 22 ;
miVector[1]= 12 ;
miVector[2]= 33 ;
//Array en una línea inicializando elementos
let otroArray=[ 1 , 2 , "Cancamusa" ];
console.log(miVector[1]);
console.log(otroArray[2]);
```

Ejemplos de Arrays /2

```
// array y tamaño
console .log(miVector + " " +miVector.length );
// Array bidimensional 5x4 declarado sin rellenar
// Para saber mas de map, mirar sección de funciones fleche
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
))
```

Clonando arrays (y objetos)

- Los arrays en Javascript internamente almacenan referencias a donde está alojado cada objeto en memoria
- 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)
);
```

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.

Ejemplos

```
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
```

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 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 ; // ERROR, no se puede asignar a una constante
```

¿Definir un array como const?

- Aunque resulta posible definir arrays y objetos usando const, no es recomendable hacerlo
 - 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
```

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 obliga a que todas las variables sean declaradas .
- “Use strict” es ignorado por versiones anteriores de Javascript, al tomarlo como una simple declaración de cadena.

Características de “use strict”

- 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.)

"use strict" en funciones

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

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

```
pi= 3.14 ; // NO da error, use strict sólo  
se aplica a la función  
funcionPrueba();
```

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

Entrada y salida en navegadores

- 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");`
- 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");`

Entrada y salida en navegadores

- A través del método `confirm()` se activa un cuadro de diálogo que contiene los botones Aceptar y Cancelar.

- Aceptar, devuelve el valor `true`;
- Cancelar devuelve el valor `false`.

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

- El método `prompt()` abre un cuadro de diálogo en pantalla en el que se pide al usuario que introduzca algún dato.

- Cancelar, devuelve `false/null`.
- Aceptar devuelve `true`
 - 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");
```


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.
- Los principales operadores de Javascript son:
 - aritméticos,
 - asignación,
 - comparación
 - lógicos.

Operadores aritméticos

- Se utilizan para realizar cálculos aritméticos.
- + Suma.
- - Resta.
- * Multiplicación.
- / División.
- % Calcula el resto de una división entera.
- También existen operadores aritméticos unitarios: incremento (++), disminución (--) y la negación unitaria (-).

Operadores de incremento y disminución

- Los operadores de incremento y disminución pueden estar tanto delante como detrás de una variable.
- Estos operadores aumentan o disminuyen en 1, respectivamente, el valor de una variable.
- (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. $x=5, y= -5$.

Operadores de Asignación

- Se utilizan para asignar valores a las variables.
 - = 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

Operadores de comparación

- Como valor de retorno se obtiene siempre un valor booleano: true o false.

=== Compara dos elementos, incluyendo su tipo interno. Devuelve false si son diferentes.

!== Compara dos elementos, incluyendo su tipo interno. Devuelve true si son diferentes.

== Devuelve el valor true cuando los dos operandos son iguales.

!= Devuelve el valor true cuando los dos operandos son distintos.

> Devuelve el valor true cuando el operando de la izquierda es mayor que el de la derecha.

< Devuelve el valor true cuando el operando de la derecha es menor que el de la izquierda.

>= Devuelve el valor true cuando el operando de la izquierda es mayor o igual que el de la derecha. ?

<= Devuelve el valor true cuando el operando de la derecha es menor o igual que el de la izquierda.

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.
- && 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.

Estructuras de Control: if/else

- La instrucción if/else 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.
-

Ejemplo if/else

- 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.

```
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!!"
```


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);
}
```

Ramificaciones Anidadas

```
let edadAna, edadLuis; // Convertiremos 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);
```

Para las condiciones ramificadas más complicadas, a menudo se utilizan las ramificaciones anidadas.

En ellas se definen consultas if dentro de otras consultas if.

Estructuras Repetitivas:

Bucle while

- Sintaxis:

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

Ejemplos While

- 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" );
```

- Pregunta una clave hasta que se corresponda con una dada.

```
let auxclave = "" ;
while
(auxclave!="vivaYO"){

    auxclave=prompt("introduc
e la clave" ,
"claveSecreta" )

}
console.log( "Has
acertado la clave" );
```

Estructuras Repetitivas: Bucle do-while

- Sintaxis

```
do {  
    // ...instrucciones...  
} while (condición);
```
- 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
 - Esto significa que el bucle do-while se recorrerá, una vez, como mínimo.

Ejemplo do-while:
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");
```

Estructuras Repetitivas: Bucle for

- Sintaxis

```
for (Inicialización del índice; Condición;  
Modificaíndice){  
    // ...instrucciones...  
}
```

- Equivale a:

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

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

- Ejecuta una única vez la “Inicialización del índice”
- Analiza la condición de prueba y si esta se cumple ejecuta las instrucciones del bucle.
- Modifica el 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.

Ejemplos for

- 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" );
```

- 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");
```




Instrucción break

- 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.
-

Pregunta por la clave y permitir tres respuestas incorrectas

```
let auxclave= true ;
let numveces= 0 ;
//Mientras no introduzca la clave y no 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");
}
```

Instrucción continue

- 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);  
}
```

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.



Diferencia entre definir una función y llamarla

- Definir una función es especificar su nombre y definir qué acciones realizará en el momento en que sea invocada, mediante la palabra reservada function.

- Sintaxis de la definición de una función:

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

- 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
-

Llamada a una función

- 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.
 - Sintaxis de la llamada a una función

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

Funciones REST

- 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” nomencado 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.

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



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`
-

Funciones especiales de arrays

- 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  
nombre  
console .log(sumaNombres);
```

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

Ejemplos de Clase

```
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);
```