Características generales de JavaScript

Curso 2025/2026

Paco Segura

¿Qué es JavaScript?

- Lenguaje de programación
  - Alto nivel
  - Interpretado → No tiene sentido compilar en Web (demasiados dispositivos, se requiere actuar en tiempo real, etc.)
  - Débilmente *tipado* →no se indican los tipos de las variables al declararlas (usar typeof para conocer tipos).
  - No es de propósito general → Desarrollo en cliente web / nativo.
  - Basado en objetos.
  - Base de la mayoría de *frameworks* de desarrollo en cliente.

JavaScript en cliente (navegador)

- **Típicamente:**
  - Nació para ser ejecutado dentro de un navegador (Netscape) y rápidamente se extendió a otros navegadores
  - En el navegador se emplea para realizar pequeños cálculos
    - Procesar interacciones del usuario con la Web y responder a ellas.
    - Modificación de la estructura o estilo de una Web al vuelo.
    - Peticiones de datos a servidor.
    - Algún pequeño cálculo que no requiere de datos almacenados en el servidor.
    - Juegos interactivos.

JavaScript fuera de cliente

- JavaScript ha crecido mucho en popularidad → Extensión a otros ámbitos fuera del cliente
- Ejemplos:
  - Juegos HTML 5 + CSS + JS multiplataforma (Phaser)
  - Aplicaciones móviles (React Native, Ionic)
  - Bases de datos (MongoDB)
  - APIs para dar servicio/datos.

Características generales de JavaScript

A continuación vamos a ver las características generales de JavaScript que ya habéis utilizado para programar: cómo declarar variables, utilizar sentencias condicionales, estructuras de control de flujo, arrays, funciones…

Declaración de variables

- Antes:

```
var x = 'Hola';
```

- Ahora:

```
const x = 'Hola';

let y = 'Hola';
```

Uso de **let**

Si *let* lo usamos para declarar variables en un ámbito global, funcionaría igual que *var*.

Sin embargo, cuando usamos *let* para declarar una variable dentro de un bloque (bucles), esa variable declarada y sus valores solo existen dentro de dicho bloque.

Pasa lo mismo con *const*.

Probar:

```
let i = 43;

for (let i = 0; i < 10; i++) {
    console.log(i);
}

console.log(i);
```

let vs const

- Ambos son formas de declarar variables en JS.
- Especialmente interesantes en bloques de código locales, ya que no machacan valores globales o otras variables visibles.
- ¿Cuándo *uso* let frente a *const* y viceversa?
  - Si sospecho o sé que el valor de esa variable puede cambiar → *let*
  - Si sé que el valor de esa variable *nunca cambiará* → *const*
    - Si intento cambiar el valor de una variable const lanza un error (TypeError).
    - Esto es útil para debuggear.

Uso de == y === (!= y !==)

- JavaScript cuenta con dos comparadores de igualdad:
  - Igualdad débil ==
    - Si se utiliza con dos tipos de datos iguales (4 == 3), compara los valores y devuelve true solo si los valores son idénticos.
    - Si se utiliza con dos tipos de datos diferentes (4 == true) intenta convertir el tipo de datos más complejo al dato más simple. Tras la conversión, si ambos valores son iguales, devuelve true.
  - Igualdad fuerte ===
    - Dos objetos solos son iguales si tienen el mismo tipo y el mismo valor. No se intenta convertir.
    - Si no estáis muy seguros de qué comportamiento queréis, esta opción es más segura.

Sentencias condicionales: **if**

USO: Cuando se requieren acciones distintas dependiendo de la condición que se dé.

Si -> condición

Acción

En cambio si -> otra condición

Otra acción

Sintaxis:

```
if (i < 0) {
    console.log('Negativo');
} else if (i > 0) {
    console.log('Positivo');
} else {
    console.log('Cero');
}
```

Sentencias condicionales: **switch**

USO: Cuando tengo una expresión de la que conozco exactamente los resultados posibles y cada uno de ellos requiere que se ejecuten acciones distintas.

Las declaraciones se ejecutan cuando el resultado de la expresión coincide con el valor de cada *case*. Si ninguno de los valores coincide, se ejecuta *default*.

Sintaxis:

```
switch (i) {
    case 0:
        console.log('suspendido');
        break;
    case 1:
        console.log('suspendido');
        break;
    case 2:
        console.log('suspendido');
        break;
    case 3:
        console.log('suspendido');
        break;
    case 4:
        console.log('suspendido');
        break;
    default:
        console.log('aprobado');
        break;
}
```

Bucles: **for**

USO: Cuando tengo que repetir un conjunto de instrucciones un número de veces.

Bucle **for** → Normalmente conozco el número de veces a ejecutar el bloque de código.

Sintaxis:

```
for (let i = 0; i < 10; i++) {
    console.log(i);
}
```

Bucles: **for** (2)

Requiere de una colección de elementos o un objeto iterable.

Requiere de una variable local al bucle que cada iteración toma el valor de uno de los elementos de la colección/objeto.

Muy conveniente para iterar sobre elementos de una colección.

Sintaxis:

```
let myArray = [1, 2, 3, 4, 5];

for (let value of myArray) {
    console.log(value);
}
```

Bucles: **while**

USO: Cuando tengo que repetir un conjunto de instrucciones un número de veces.

Bucle **while** → Sé que quiero ejecutar el conjunto de instrucciones, pero desconozco el número de veces. Sé que lo quiero ejecutar mientras se cumpla una condición.

Sintaxis:

```
while(i < 5){
    console.log(i);
    i += 1;
}
```

Funciones:

USO: para definir un patrón de código con una entrada y salida.

Muy importantes en JavaScript. Hablaremos de ellas más adelante.

Sintaxis:

```
function multiply(a, b) {
    return a * b;
}
```

Arrays:

Estructura de datos dinámica donde los valores tienen una posición de 0 a n-1.

push → Añade al final

pop → Quita del principio.

Podemos acceder por índice.

Sintaxis:

```
let a = []
let b = [1, 2, 3];
a.push(5);
a.push(7);
```

Sintaxis:

```
// Array
let c = new Array(2);

//En cada posición de 'c' guardamos un nuevo array
c[0] = new Array(2);
c[1] = new Array(2);

//Accedemos a la fila 0, columna 1
c[0][1];
```