# JavaScript

## *Tipos de datos, variables y operadores*

CertiDevs

# Índice de contenidos

# 1. Sintaxis Básica

- JavaScript es case-sensitive, lo que significa que diferencia entre mayúsculas y minúsculas.
- Las instrucciones en JavaScript se terminan con punto y coma `;` (aunque es opcional en la mayoría de los casos, se recomienda usarlo para evitar errores).
- Los comentarios de una sola línea comienzan con `//`.
- Los comentarios multilínea se encierran entre /* y */.

# 2. Variables

- Las variables se declaran con las palabras clave `var`, `let` o `const`.
- `var` se usa para declarar variables en versiones anteriores de JavaScript (ES5 y anteriores).
- `let` y `const` se introdujeron en ES6 y son preferibles por su comportamiento de alcance de bloque.
- `const` se utiliza para declarar variables cuyo valor no cambiará en el futuro.

## 2.1. Nombres de Variables

Al elegir nombres para variables, sigue estas reglas y convenciones:

- Los nombres de las variables deben comenzar con una letra, un guion bajo (_) o un signo de dólar ($).
- Los nombres de las variables pueden contener letras, números, guiones bajos y signos de dólar.
- No se permiten espacios ni caracteres especiales en los nombres de las variables.
- Los nombres de las variables son sensibles a mayúsculas y minúsculas (es decir, variable y Variable son diferentes).

Además, sigue las convenciones de nombres para mejorar la legibilidad y el mantenimiento del código:

- Utiliza nombres descriptivos y significativos para las variables.
- Utiliza la notación de camelCase para nombres de variables de varias palabras (por ejemplo, `miVariable`, en vez de `mi variable`).
- Evita nombres de variables demasiado cortos o nombres que no describan claramente el propósito de la variable.

## 2.2. Asignación y Reasignación de Variables

- Para asignar un valor a una variable, utiliza el operador de asignación (`=`).
- Puedes reasignar valores a variables `var` y `let` usando el operador de asignación.

```
var variableVar = 'Valor inicial';
```

```
let variableLet = 'Valor inicial';

variableVar = 'Nuevo valor';
variableLet = 'Nuevo valor';
console.log(variableVar); // 'Nuevo valor'
console.log(variableLet); // 'Nuevo valor'

// No puedes reasignar valores a variables const.
const variableConst = 'Valor inicial';
variableConst = 'Nuevo valor'; // TypeError: Assignment to constant variable.
```

## 2.3. Variables y Alcance (Scope)

Entender el alcance de las variables es fundamental para trabajar con ellas en JavaScript:

- Alcance **global**: Una variable declarada fuera de cualquier función o bloque tiene un alcance global y es accesible desde cualquier parte del código.
- Alcance de **función**: Las variables declaradas dentro de una función con var están disponibles solo dentro de esa función y no son accesibles fuera de ella.
- Alcance de **bloque**: Las variables declaradas dentro de un bloque (por ejemplo, dentro de una declaración if o un bucle for) con let o const están disponibles solo dentro de ese bloque.

```
var globalVar = 'Soy una variable global';

function ejemploScope() {
    var localVar = 'Soy una variable local';
    let localLet = 'Soy una variable local con let';
    const localConst = 'Soy una variable local con const';

    if (true) {
        let bloqueLet = 'Soy una variable de bloque';
        const bloqueConst = 'Soy una variable de bloque con const';
        console.log(bloqueLet); // 'Soy una variable de bloque'
        console.log(bloqueConst); // 'Soy una variable de bloque con const'
    }

    console.log(localVar); // 'Soy una variable local'
    console.log(localLet); // 'Soy una variable local con let'
    console.log(localConst); // 'Soy una variable local con const'
}
ejemploScope();
console.log(globalVar); // 'Soy una variable global'
console.log(localVar); // ReferenceError: localVar is not defined
```

## 2.4. Inicialización de Variables

- Las variables declaradas con `var` se inicializan automáticamente con el valor `undefined`.
- Las variables declaradas con `let` y `const` no se inicializan automáticamente y deben ser inicializadas antes de su uso, de lo contrario, se generará un error de referencia.

```
var variableVar;
console.log(variableVar); // undefined

let variableLet;
console.log(variableLet); // undefined
const variableConst; // SyntaxError: Missing initializer in const declaration
```

## 2.5. Variables Temporales Muertas (Temporal Dead Zone)

Las variables declaradas con let y const tienen un concepto llamado "zona muerta temporal" (Temporal Dead Zone, TDZ). La TDZ es el espacio entre el inicio de un bloque de código y la declaración de una variable let o const en ese bloque. En la TDZ, la variable está en un estado no inicializado y no se puede acceder.

```
function ejemploTDZ() {
    console.log(variableLet); // ReferenceError: variableLet is not defined (TDZ)
    let variableLet = 'Valor de variableLet';
    console.log(variableLet); // 'Valor de variableLet'
}
ejemploTDZ();
```

## 2.6. Elevación (Hoisting)

El concepto de elevación (hoisting) es importante para entender el comportamiento de las variables en JavaScript:

- Las declaraciones de variables var se elevan al inicio de su ámbito, pero no su asignación.
- Las declaraciones de variables let y const no se elevan.

```
function ejemploHoisting() {
    console.log(variableVar); // undefined
    var variableVar = 'Valor de variableVar';

    console.log(variableLet); // ReferenceError: variableLet is not defined
    let variableLet = 'Valor de variableLet';
    console.log(variableConst); // ReferenceError: variableConst is not defined
    const variableConst = 'Valor de variableConst';
```

```
}
ejemploHoisting();
```

Es importante recordar las diferencias entre var, let y const, así como comprender cómo se comportan con respecto al alcance, la elevación (hoisting) y la zona muerta temporal (TDZ)

# 3. Tipos de Datos

- JavaScript es un lenguaje de tipado dinámico, lo que significa que no es necesario especificar el tipo de dato al declarar una variable.
- Los tipos de datos básicos son: `String`, `Number`, `Boolean`, `null`, `undefined` y `Object`.

A continuación, se enumeran y explican los principales tipos de datos en JavaScript:

## 3.1. String

- Las cadenas de texto (strings) representan secuencias de caracteres.
- Pueden crearse utilizando comillas simples ', comillas dobles " o comillas invertidas (template literals) `.
- Las cadenas de texto pueden concatenarse utilizando el operador + o interpolación de variables en las template literals.

```
const str1 = 'Hola';
const str2 = "mundo";
const str3 = `Esto es una template literal`;
const concatenacion = str1 + ' ' + str2;
const interpolacion = `${str1} ${str2}`;
```

Las cadenas en JavaScript son objetos y, como tal, tienen métodos útiles para manipular y trabajar con texto. Algunos de estos métodos incluyen:

- `charAt(index)`: devuelve el carácter en la posición index.
  concat(string1, string2, ...): concatena las cadenas proporcionadas.
- `includes(searchString)`: devuelve true si la cadena incluye searchString, de lo contrario, devuelve false.
- `indexOf(searchString)`: devuelve el índice de la primera aparición de searchString en la cadena, o -1 si no se encuentra.
- `lastIndexOf(searchString)`: devuelve el índice de la última aparición de searchString en la cadena, o -1 si no se encuentra.
- `replace(searchValue, replaceValue)`: devuelve una nueva cadena con la primera aparición de searchValue reemplazada por replaceValue.
- `slice(beginIndex, endIndex)`: devuelve una subcadena desde beginIndex hasta (pero no incluyendo) endIndex.

- `split(separator)`: divide la cadena en un array de subcadenas utilizando el separator.
- `startsWith(searchString)`: devuelve true si la cadena comienza con searchString, de lo contrario, devuelve false.
- `substring(startIndex, endIndex)`: devuelve una subcadena desde startIndex hasta (pero no incluyendo) endIndex.
- `toLowerCase()`: devuelve una cadena con todos los caracteres en minúsculas.
- `toUpperCase()`: devuelve una cadena con todos los caracteres en mayúsculas.
- `trim()`: elimina los espacios en blanco del principio y final de la cadena.

## 3.2. Number

- Los números en JavaScript incluyen tanto enteros como decimales.
- Pueden representarse en notación decimal, hexadecimal (prefijo 0x), octal (prefijo 0o) y binaria (prefijo 0b).
- JavaScript utiliza el estándar IEEE 754 para representar números, lo que puede llevar a imprecisiones en operaciones aritméticas.

```
const decimal = 42;
const decimalConDecimales = 3.1415;
const hexadecimal = 0xFF;
const octal = 0o52;
const binario = 0b101010;
```

## 3.3. Boolean

- Los booleanos representan valores de verdad, es decir, true (verdadero) y false (falso).
- Resultan de comparaciones, operaciones lógicas o pueden asignarse directamente a una variable.

```
const verdadero = true;
const falso = false;
const comparacion = 5 < 10; // true
```

## 3.4. null

- El valor null representa la ausencia intencional de valor en una variable.
- Se utiliza para indicar que una variable no tiene ningún valor asignado.

```
const vacio = null;

var testVar = null;
```

```
console.log(testVar); //shows null
console.log(typeof testVar); //shows object
```

## 3.5. undefined

- El valor undefined indica que una variable ha sido declarada pero no se le ha asignado ningún valor.
- Si intentas acceder a una propiedad que no existe en un objeto, también obtendrás undefined.

```
let sinValor;
console.log(sinValor); // undefined
console.log({}.propiedadInexistente); // undefined

var testVar;
console.log(testVar); //shows undefined
console.log(typeof testVar); //shows undefined
```

## 3.6. Object

- Los objetos son una colección de propiedades, que son pares clave-valor.
- Pueden representar estructuras de datos complejas y se pueden crear utilizando literales de objeto o la palabra clave new.

```
const objLiteral = {
clave1: 'valor1',
clave2: 'valor2'
};

const objConstructor = new Object();
objConstructor.clave1 = 'valor1';
objConstructor.clave2 = 'valor2';
```

Además de estos tipos de datos primitivos, JavaScript también incluye estructuras de datos más avanzadas, como arrays, sets, maps y otros objetos predefinidos.

# 4. Operadores

Los operadores en JavaScript son símbolos que se utilizan para realizar operaciones en valores y variables. Se pueden clasificar en varias categorías:

- Los operadores aritméticos incluyen: , -, *, /, % (módulo), ** (exponente) y +/-- (incremento/decremento).
- Los operadores de comparación incluyen: ==, ===, !=, !==, <, >, ⇐, >=.

- Los operadores lógicos incluyen: && (AND), || (OR), ! (NOT).

## 4.1. Operadores Aritméticos

- Los operadores aritméticos realizan operaciones matemáticas básicas.
- Incluyen: suma (), resta (-), multiplicación (*), división (/), módulo (%), exponente (**), incremento (+) y decremento (--).

```
const suma = 5 + 3; // 8
const resta = 5 - 3; // 2
const multiplicacion = 5 * 3; // 15
const division = 5 / 3; // 1.6666...
const modulo = 5 % 3; // 2
const exponente = 5 ** 3; // 125
let x = 5;
x++; // x = 6
x--; // x = 5
```

## 4.2. Operadores de Comparación

- Los operadores de comparación se utilizan para comparar dos valores.
- Incluyen: igual (==), estrictamente igual (===), no igual (!=), estrictamente no igual (!==), menor que (<), mayor que (>), menor o igual que (⇐), mayor o igual que (>=).

```
const igual = 5 == '5'; // true (compara valor sin importar el tipo)
const estrictamenteIgual = 5 === '5'; // false (compara valor y tipo)
const noIgual = 5 != '5'; // false (compara valor sin importar el tipo)
const estrictamenteNoIgual = 5 !== '5'; // true (compara valor y tipo)
const menorQue = 5 < 3; // false
const mayorQue = 5 > 3; // true
const menorOigualQue = 5 <= 5; // true
const mayorOigualQue = 5 >= 3; // true
```

## 4.3. Operadores Lógicos

- Los operadores lógicos se utilizan para combinar expresiones booleanas.
- Incluyen: AND lógico (&&), OR lógico (||), NOT lógico (!).

```
const verdad = true;
const mentira = false;
const andLogico = verdad && mentira; // false
const orLogico = verdad || mentira; // true
const notLogico = !verdad; // false
```

## 4.4. Operadores de Asignación

- Los operadores de asignación asignan un valor a una variable.
- Incluyen:
  - asignación simple (=),
  - suma y asignación (+=),
  - resta y asignación (-=),
  - multiplicación y asignación (*=),
  - división y asignación (/=),
  - módulo y asignación (%=),
  - exponente y asignación (**=)

```
let a = 5;
a += 3; // a = 8
a -= 3; // a = 5
a *= 3; // a = 15
a /= 3; // a = 5
a %= 2; // a = 1
a **= 3; // a = 1
```

## 4.5. Operadores de Cadena

- El operador de concatenación (+) se utiliza para unir dos cadenas de texto.
- También se puede utilizar la interpolación de variables en las template literals (`) para unir cadenas de texto y expresiones.

```
const cadena1 = 'Hola, ';
const cadena2 = 'mundo!';
const concatenacion = cadena1 + cadena2; // 'Hola, mundo!'
const variable = 'mundo';
const interpolacion = `Hola, ${variable}!`; // 'Hola, mundo!'
```

## 4.6. Operadores Ternarios

- El operador ternario (también conocido como operador condicional) es una forma corta de una estructura de control if-else.
- Utiliza el símbolo ? para separar la condición y la expresión que se evalúa si la condición es verdadera, y el símbolo : para separar las expresiones que se evalúan si la condición es verdadera o falsa.

```
const numero = 5;
```

```
const esPar = numero % 2 === 0 ? 'par' : 'impar'; // 'impar'
```

## 4.7. Operadores de Tipo

- Los operadores de tipo se utilizan para obtener información sobre el tipo de una variable o para cambiar su tipo.
- Incluyen: `typeof` (devuelve una cadena que indica el tipo de una variable) y instanceof (comprueba si un objeto es una instancia de un tipo en particular).

```
const texto = 'Hola';
const numero = 42;
const objeto = {};

console.log(typeof texto); // 'string'
console.log(typeof numero); // 'number'
console.log(typeof objeto); // 'object'

console.log(texto instanceof String); // false (texto es un primitivo, no un objeto
String)
console.log(objeto instanceof Object); // true
```

## 4.8. Operadores de Bits

- Los operadores de bits realizan operaciones a nivel de bits en números.
- Incluyen: AND (&), OR (|), XOR (^), NOT (~), desplazamiento a la izquierda (<<), desplazamiento a la derecha con signo (>>) y desplazamiento a la derecha sin signo (>>>).

```
const a = 5; // 0101 en binario
const b = 3; // 0011 en binario
const andBits = a & b; // 0001 en binario, 1 en decimal
const orBits = a | b; // 0111 en binario, 7 en decimal
const xorBits = a ^ b; // 0110 en binario, 6 en decimal
const notBits = ~a; // 1010 en binario, -6 en decimal
const shiftLeft = a << 1; // 1010 en binario, 10 en decimal
const shiftRight = a >> 1; // 0010 en binario, 2 en decimal
const shiftRightUnsigned = a >>> 1; // 0010 en binario, 2 en decimal
```