

- 1. Características Javascript**
- 2. Ejecución de JavaScript**
- 3. Generalidades de JavaScript**
- 4. Tipos de datos**
- 5. Operadores aritméticos**
- 6. Estructuras de control**
- 7. Entrada y salida en el navegador**
- 8. Variables**
- 9. ARRAYS**
- 10. Sets y WeakSets**
- 11. Maps y WeakMaps**

Características Javascript

JavaScript es un lenguaje de programación :

- **Ligero.** está diseñado para ocupar poco espacio en memoria, es fácil de implementar y cuenta con una sintaxis y una semántica simples, por lo que se puede aprender en poco tiempo.
- **Interpretado:** significa que se lee y se ejecuta cada línea de código de forma secuencial, a contrario que los lenguajes compilados cuyas líneas de código son convertidas en su conjunto a lenguaje máquina para ser ejecutados posteriormente.
- **Basado en prototipos:** trata de una variante de la programación orientada a objetos no se crean instanciando clases sino clonando otros objetos o creándolos directamente. Lo que debe recordarse de esta característica es que en Javascript todo es un objeto.
- **Case sensitive:** es sensible a mayúsculas y minúsculas, por lo que rojo, Rojo y ROJO serán tres identificadores distintos.

Características Javascript

- **Débilmente tipado:** el lenguaje no necesita conocer al detalle con qué tipo de dato se está trabajando en cada momento. Es lo suficientemente inteligente para deducirlo o realizar conversiones entre tipos de datos a lo largo de la ejecución del programa.
- **Multiparadigma:** a diferencia de otros lenguajes, permite programar aplicaciones completas desde cero aplicando muy distintos paradigmas de programación.
- **Monohilo:** un único hilo de ejecución se encarga de realizar el trabajo de interpretación del código, de forma que se dificultan los interbloqueos y se favorece el uso de las llamadas asíncronas.
- **Dinámico:** el lenguaje es capaz de cambiar importantes características del programa, como la estructura de un objeto o el tipo de una variable, mientras se está ejecutando.
- **Programación orientada a objetos:** un modelo de programación mucho más cercano al mundo real donde se modelan patrones a través de clases y se instancian en forma de objetos. Los objetos se relacionan entre sí para alcanzar los objetivos de las aplicaciones.

Características Javascript

- **Programación imperativa:** un modelo de programación consistente en una secuencia de instrucciones que determinan qué debe hacer la máquina en cada momento paso a paso. Se centran en el «cómo» de la solución.
- **Programación declarativa:** un modelo de programación que consiste en describir qué se quiere obtener al finalizar la ejecución del programa, en lugar de cómo se quiere obtener. Se centran en el «qué» de la solución.

Ejecución de JavaScript

JavaScript interno

Se encierra entre etiquetas `<script>` y se inserta en cualquier parte del documento, aunque suele ir entre etiquetas `<head>`

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html >
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<title>Ejemplo de código JavaScript en el propio documento</title>
<script>
console.log("Hola Mundo");
</script>
</head>
<body>
<p>Un párrafo de texto.</p>
</body>
</html>
```

- Impide la reutilización de código
- Genera archivos HTML con excesivo código
- Hace incomprensible la ejecución de javascript

Ejecución de JavaScript

JavaScript externo

Incluir en un archivo externo de tipo JavaScript que los documentos HTML enlazan mediante la etiqueta `<script>`. Se indica dónde está el archivo mediante la opción `src`. A partir de HTML 5 sólo es necesaria la propiedad `src`.

Archivo .html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html >
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<title>Ejemplo de código JavaScript en el propio documento</title>
<script src="js/codigo.js">
</script>
</head>
<body>
<p>Un párrafo de texto.</p>
</body>
</html>
```

Archivo .js

```
Console.log("Hola mundo");
```

- El código está mejor organizado
- Es reutilizable en varios archivos HTML
- El HTML es más fácil de leer sin grandes trozos de script en él

Ejecución de JavaScript

JavaScript externo: Ubicación de la etiqueta script

El navegador puede descargar un documento Javascript en cualquier momento de la carga de la página y necesitamos saber cuál es el más oportuno para nosotros.

- Si queremos que un documento Javascript actúe antes que se muestre la página, la opción de colocarlo en el <head> es la más adecuada.
- Si por el contrario, queremos que actúe una vez se haya terminado de cargar la página, la opción de colocarlo justo antes del </body> es la más adecuada. Esta opción equivale a usar el atributo defer en la etiqueta <script>, sin embargo, esta opción es además compatible con navegadores muy antiguos ([IE9 o anteriores](#)) que no soportan **defer**.

Ubicación	¿Cómo descarga el archivo Javascript?	Estado de la página
En <head>	ANTES de empezar a dibujar la página.	Página aún no dibujada.
En <body>	DURANTE el dibujado de la página.	Dibujada hasta donde está la etiqueta <script>.
Antes de </body>	DESPUÉS de dibujar la página.	Dibujada al 100%.

Ejecución de JavaScript

Controladores de JavaScript en línea

Incluir fragmentos de JavaScript dentro del código HTML de la página:

```
<button onclick="alert('Un mensaje de prueba') ">Pulsar</button>
```

Incluir fragmentos de JavaScript dentro del código HTML de la página, pero llamando a funciones de un fichero js:

Archivo .html

```
<button onclick="mostrar()">Pulsar</button>
```

Archivo .js

```
function crearParrafo() {  
    alert("Hola mundo");  
}
```

NO Incluir JavaScript en los elementos HTML

Ejecución de JavaScript

Los elementos `<script>` son unos de los recursos más comunes que **bloquean el análisis y renderizado** de un documento HTML. Cuando el navegador se encuentra con este tipo de recursos, detiene el análisis del documento, descarga el recurso, lo ejecuta y luego continua con el análisis del documento

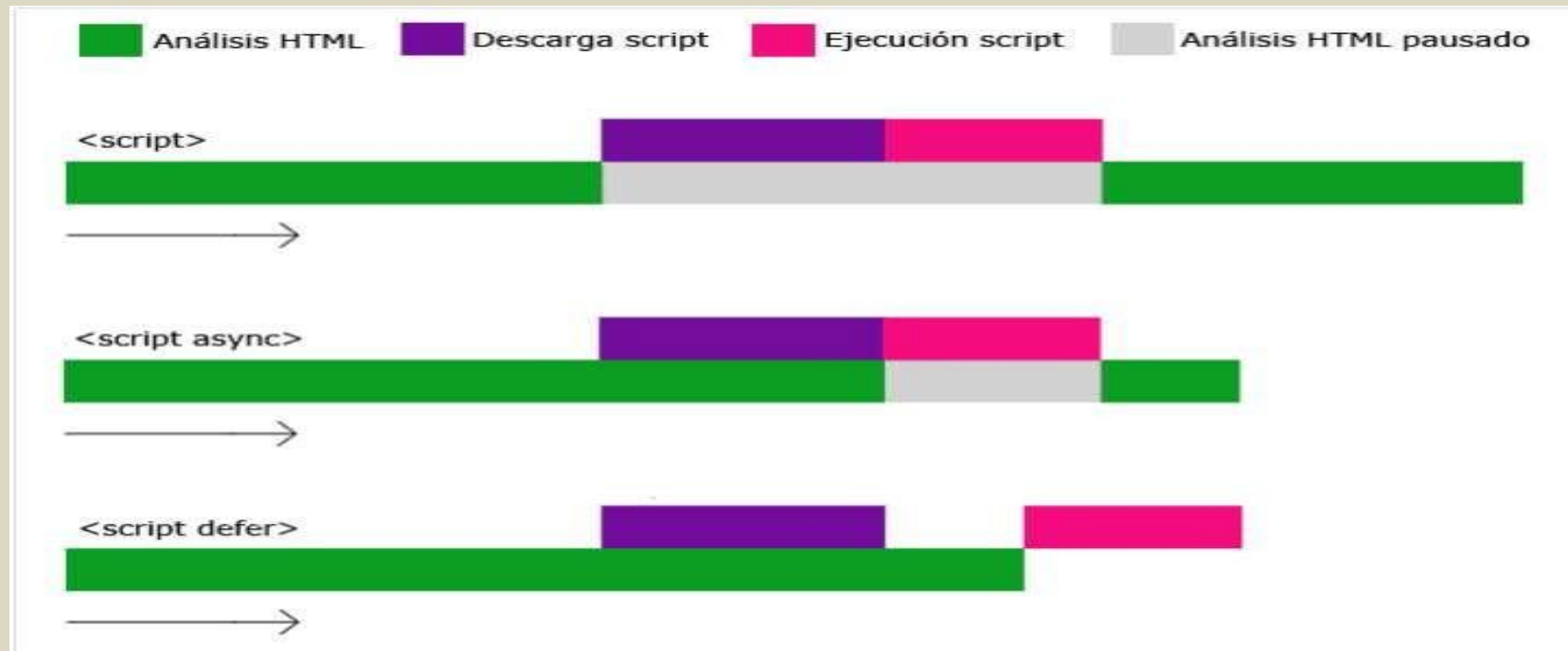
Para evitar este bloqueo podemos colocar los elementos `<script>` al final del HTML. Cuando el analizador se encuentra con los scripts, casi todo el documento ya se ha analizado y renderizado. Pero era una solución lejos de ser ideal por lo que está en desuso

HTML5 introduce dos atributos para cargar archivos .js

1. **`<script async>`**: el script se descarga de forma asíncrona, es decir, sin detener el análisis HTML, pero una vez descargado, sí se detiene para ejecutar el script. Tras la ejecución se reanuda el análisis HTML. Sigue existiendo un bloqueo en el renderizado pero menor que con el comportamiento normal. No se garantiza la ejecución de los scripts asíncronos en el mismo orden en el que aparecen en el documento
2. **`<script defer>`**: el script se descarga de forma asíncrona, en paralelo con el análisis HTML, y además su ejecución es diferida hasta que termine el análisis HTML. No hay bloqueo en el renderizado HTML. La ejecución de todos los scripts diferidos se realiza en el mismo orden en el que aparecen en el documento.

Ejecución de JavaScript

Descarga y análisis de archivos



async el script es completamente independiente. El navegador no se bloquea, se cargan en segundo plano y se ejecutan cuando están listos. El DOM y otros scripts no esperan por ellos, y ellos no esperan por nada

defer indica al navegador que no espere por el script, debe seguir procesando el HTML, construir el DOM. El script carga “en segundo plano” y se ejecuta cuando el DOM esta completo

Generalidades javaScript

Sangría y presentación

La sangría en programación, es una manera de estructurar el código para hacerlo más legible

Las instrucciones son priorizadas en varios niveles para desplazar a la derecha y crear una jerarquía

```
function interruptor(elemId) {  
    var elem = document.getElementById(elemId);  
    if (elem.style.display == 'block') {  
        elem.style.display = 'none';  
    } else {  
        elem.style.display = 'block';  
    }  
}
```

Los **comentarios** son anotaciones para explicar el funcionamiento de un script, una instrucción o incluso un grupo de instrucciones. Los comentarios no interfieren con la ejecución de un script

// para comentar una instrucción
comentario multilínea comienza con /* y termina con */

Tipos de datos

Dos grupos: **tipos de datos primitivos** (*básicos*) y **tipos de datos no primitivos** (*complejos*).
Caso especial: null. Representa un valor vacío o ausencia de información.

Tipo de dato

Descripción

Tipos de datos primitivos

Number	Valor numérico (enteros, decimales...)
BigInt	Valor numérico muy grande
String	Valor de texto (cadenas de texto, caracteres...)
Boolean	Valor booleano (valores verdadero o falso)
Symbol	Símbolo (valor único)
undefined	Valor sin definir (variable sin inicializar)

Tipos de datos no primitivos

Object	Objeto (estructura más compleja)
Function	Función (función guardada en una variable)

Tipos de datos

Tipos de datos primitivos

- Si es un valor numérico, será de tipo number.
- Si es un valor de texto, será de tipo string.
- Si es verdadero o falso, será de tipo boolean.

Ejemplo:

```
// Un texto, letra o carácter
const text = "Hola, me llamo Mario";
// Un número (entero o decimal):
const number = 42;
// Un número muy grande (se añade n al final)
const bignumber = 12345678901234567890n;
// Un valor de verdadero o falso
const boolean = true;
// Un valor único
const symbol = Symbol("unique");
```

Tipos de datos

El tipo de dato undefined

Es el valor que tienen las variables a las que no se les ha dado ningún valor específico (*es decir, que están sin definir*).

let notDefined; // Tiene el valor undefined

let sinDefinir = undefined; // Aunque no es lo habitual, también se puede asignar explícitamente

El valor especial null

El valor especial null indica la **ausencia intencional** de información. A diferencia de undefined, que indica que el valor **aún no se ha definido**, null indica que el valor ha sido definido explícitamente, pero representa una **ausencia de valor**.

Ejemplo:

// Caso A

let selectedUser;

// Caso B

let selectedUser = null;

En el caso A, selectedUser tiene el valor undefined indicando que posiblemente el usuario aún no ha sido seleccionado (*no se ha realizado la asignación de un huésped a esa habitación*). En el caso B, selectedUser tiene el valor null, indicando que el usuario ha sido seleccionado intencionalmente, pero no es ningún usuario (*la habitación está vacía, no tiene huésped*).

Tipos de datos

Tipos de datos no primitivos

En Javascript cualquier tipo de dato no primitivo se considera un object, lo que suele indicar que se trata de un tipo de dato más complejo.

Ejemplo de tipos de datos no primitivos:

```
// Tipo de dato: Objeto  
const user = { name: "Mario" };  
// Tipo de dato: Array  
const users = ["Mario", "Juan", "Pedro"];
```

En estos dos casos, los tipos de datos de cada constante se consideran object, más concretamente el primero es un Objeto y el segundo un Array.

Tipos de datos

A veces no resulta sencillo saber cuál es el tipo de dato de una variable o constante. Por ejemplo, cuando una función devuelve un valor que en principio desconocemos.

El operador typeof

La función `typeof` nos devuelve el tipo de dato primitivo de la variable que le pasemos por parámetro.

```
const text = "Hola Mundo!";  
typeof text; // Devuelve "String"  
const number = 42;  
typeof number; // Devuelve "Number"  
const boolean = true;  
typeof boolean; // Devuelve "Boolean"  
let notDefined;  
typeof notDefined; // Devuelve undefined
```

OJO: La función `typeof` no nos servirá para variables con **tipos de datos** más complejos, ya que siempre los mostrará como `object` y no sabremos exactamente el tipo de dato concreto:

```
const numbers = [1, 2, 3, 4];  
typeof numbers; // Devuelve "object"  
const user = { name: "Juan" };  
typeof user; // Devuelve "object"
```

Para esto puede ser mejor utilizar `constructor.name`.

Tipos de datos

La propiedad constructor.name

Con `constructor.name`, que es una parte de la Orientación a objetos, podemos obtener el tipo de constructor que se utiliza.

```
const text = "Hola Mundo!";  
text.constructor.name; // String  
const number = 42;  
number.constructor.name; // Number  
const boolean = true;  
boolean.constructor.name; // Boolean  
let notDefined;  
notDefined.constructor.name; // ERROR // constructor.name sólo funciona en variables definidas  
const numbers = [1, 2, 3, 4];  
numbers.constructor.name; // Array  
const user = { name: "Juan" };  
user.constructor.name; // Object
```

Tipos de datos

Cadena de prototipos

Javascript funciona de modo que cada elemento tiene una **cadena de prototipos**, esto es, cada elemento hereda de otros elementos.

Ejemplo: la función `getPrototypeOfChain()` devuelve una lista de prototipos de un elemento pasado por parámetro:

```
getPrototypeOfChain("Hola"); // ['String', 'Object', null]
getPrototypeOfChain(42); // ['Number', 'Object', null]
getPrototypeOfChain([1, 2, 3]); // ['Array', 'Object', null]
getPrototypeOfChain(/.+/); // ['RegExp', 'Object', null]
```

Los prototipos de "Hola" (*un String*) son String, Object y null, por lo que "Hola" hereda de los prototipos String, Object y null. Por otro lado, los prototipos de 42 son Number, Object y null.

Consideraciones:

- El primer prototipo es el tipo de dato de la variable. En el primer ejemplo: es String, por lo que puede usar todas las operaciones para String.
- El segundo prototipo es un Object. Por lo que también puede usar las operaciones de Object.
- Por último, null es el terminador de la cadena de prototipos. Se ha llegado al final.

Tipos de datos

El operador instanceof

Se puede utilizar instanceof para saber si una variable hereda de un tipo concreto.

Por ejemplo:

```
const numbers = [1, 2, 3];  
getPrototypeOf(numbers); // ['Array', 'Object', null]  
numbers instanceof Array // true  
numbers instanceof Object // true  
numbers instanceof Number // false  
numbers instanceof String // false
```

Pero mientras que typeof sólo sirve para tipos de datos primitivos, instanceof sólo sirve para tipos de datos no primitivos.

El hecho de que Javascript determine los **tipos de datos automáticamente** no quiere decir que no debemos preocuparnos por ello. A menudo, deberemos conocer el tipo de dato de una variable e incluso necesitaremos convertirla a otros tipos de datos.

Tipos de datos

En Javascript, tenemos un tipo de dato llamado **Boolean** que sólo puede tener dos valores: true (*verdadero*) o false (*falso*).

Conversiones de tipo de dato

En Javascript, podemos **forzar** que valores de un tipo de dato se conviertan a otro, simplemente escribiendo el tipo de dato como si fuera una función.

Por ejemplo: convertimos un número en un String:

```
const number = 42; // 42
number.constructor.name; // "Number"
const text = String(number); // "42"
text.constructor.name; // "String"
```

Se puede hacer ésto con múltiples tipos de datos como por ejemplo Number(), String(), Boolean(), RegExp(), Array() y muchos más, aunque lo más habitual es usarlo con los tres primeros.

Tipos de datos

Valores Falsy y Truthy

Cuando trabajamos con valores que no son booleanos, y los utilizamos como booleanos, es más complejo determinar que valor tienen.

Se le llama **Falsy** a un valor que aunque realmente no es un booleano false se comporta como tal. Por el lado opuesto, los valores **Truthy** son valores que aunque no sean booleanos, se comportan como un booleano true. Reglas:

- Se consideran false:
 - Cualquier valor false.
 - Cualquier valor que sea 0, incluyendo 0.0 o 0n (*bigint*).
 - Cualquier valor que sea una cadena vacía ("").
 - Los valores especiales null, undefined y NaN.
- El resto de valores, se consideran **truthy**.

En el siguiente ejemplo, convertimos números en booleanos:

```
Boolean(42); // true (número positivo)
```

```
Boolean(0); // false (cero)
```

```
Boolean(-42); // true (número negativo)
```

Tipos de datos

¿Qué es un Symbol?

Es una forma alternativa de crear **identificadores únicos** que ofrece algunas ventajas y garantías sobre usar nombres o números.

```
const id = Symbol("id"); // Identificador de "id"  
const unique = Symbol("unique"); // Identificador de "unique"  
Symbol("unique") === Symbol("unique"); // false
```

Declarando un Symbol() y pasándole por parámetro un String, se creará un símbolo (*identificador único*) para ese texto y lo devuelve como resultado. Es una forma rápida y simple de tener algo **realmente único**. Además, es inmutable, por lo que no podemos modificarlo intencional ni accidentalmente.

En la última línea, a pesar de crear dos símbolos con el mismo String, los dos objetos realmente son diferentes porque son **únicos**.

Si lo que queremos es comprobar si su parámetro es el mismo, podemos acceder a su descripción:

```
const u1 = Symbol("unique");  
const u2 = Symbol("unique");  
u1 === u2; // false (son símbolos diferentes)  
u1.description === u2.description; // true (se crearon con el mismo texto)
```

Tipos de datos: Number

Number

Dos formas de declararlos:

// Notación literal (preferida)

```
const number = 4;
```

```
const decimal = 15.8;
```

```
const legibleNumber = 5_000_000;
```

// Notación con objetos (evitar)

```
const number = new Number(4);
```

```
const decimal = new Number(15.8);
```

```
const letter = new Number("A");
```

Los números con decimales, en Javascript los separamos con un **punto** (.), mientras que de forma opcional, podemos utilizar el **guión bajo** (_) para separar visualmente y reconocer las magnitudes que usamos:

```
5_000_000 === 5000000; // true
```

Cualquier parámetro pasado al new Number() que no sea un número (*por ejemplo, la "A"*), dará como resultado un valor NaN (*Not A Number*).

Un número muy grande (*o muy pequeño*) puede tener un comportamiento extraño:

```
const incorrectNumber = 482598752875287528533; // 482598752875287540000 (Diferente)
```

```
const bigNumber = 482598752875287528533n; // 482598752875287528533n (El mismo, al añadir n)
```

Tipos de datos: Number

Rangos numéricos seguros

Cuando trabajamos con datos numéricos, es posible que ciertos números no se puedan representar exactamente, y no sean tan precisos como nos gustaría. Esto ocurre porque se guardan en un formato llamado coma flotante de doble precisión.

Observa la siguiente gráfica donde se puede ver claramente:



Existe una serie de **constantes** definidas en relación a este tema, que marcan los límites mínimo o máximo:

Constante	Valor en Javascript	Descripción
Number.MAX_VALUE	$\sim 2^{1024}$	Valor más grande
Number.MIN_VALUE	$\sim 5 \times 10^{-324}$	Valor más pequeño
Number.MAX_SAFE_INTEGER	$2^{53}-1$	Valor seguro más grande (para hacer cálculos)
Number.MIN_SAFE_INTEGER	$-(2^{53}-1)$	Valor seguro más pequeño (para hacer cálculos)
Number.EPSILON	2^{-52}	Número muy pequeño: ϵ

Si necesitamos realizar operaciones con muy alta precisión numérica en Javascript, se recomienda utilizar el tipo de dato BigInt.

Tipos de datos: Number

Constante Infinity

Se utiliza para representar un número muy alto, sin importar el valor en sí. Constante - Infinity se utiliza para representar valores muy bajos donde no importa el valor en sí:

Comprobaciones numéricas

// ¿Número finito?

Number.isFinite(42); // true

Number.isFinite(551.3); // true

Number.isFinite(Infinity); // false, es infinito

1e309 === Infinity; // true

Number.isSafeInteger(1e309); // false (valor en la franja roja)

// ¿Número seguro?

Number.isSafeInteger(1e15); //

true (valor en la franja verde)

Number.isSafeInteger(1e16); //

false (valor en la franja amarilla)

// ¿Número entero?

Number.isInteger(5); // true

Number.isInteger(4.6); // false

La notación 1e5 significa «5 ceros seguidos de un 1», es decir, 100000.

Representación numérica

En Javascript tenemos dos formas de representar los números: la **notación exponencial** (*basada en potencias de 10*) y la **notación de punto fijo** (*una cantidad de dígitos para la parte entera, y otra para la parte decimal*).

Tipos de datos: Number

Nan (Not A Number)

Se usa para representar **valores numéricos** que son indeterminados o imposibles de representar como número. Tipos:

- **Indeterminación matemática:** Por ejemplo, $0 / 0$.
- **Valores imposibles:** Por ejemplo, $4 - 'a'$, ya que es imposible restar una letra a un número.
- **Operaciones con NaN:** Por ejemplo, $\text{NaN} + 4$, ya que el primer operando es NaN.

También existe un `Number.NaN` como propiedad de `Number`, para tenerlo modularizado y encapsulado como propiedad estática en el objeto `Number`.

Función `Number.isNaN()`

`Number.isNaN(NaN); // true` (Forma correcta de comprobarlo)

`Number.isNaN(5); // false` (5 es un número, no es un NaN)

`Number.isNaN("A"); // false` ("A" es un string, no es un NaN)

Conversiones de valores NaN

Si intentamos convertir un valor NaN a otro tipo de dato primitivo

`Boolean(NaN); // false` (convertimos a booleano)

`String(NaN); // "NaN"` (convertimos a texto)

`Number(NaN); // NaN` (Ya estaba en contexto numérico)

Tipos de datos: Number

Para **convertir un string en number** se utilizan las funciones de parseo numérico:

.parseInt() funciona perfectamente para variables de texto que son números (*o que empiezan por números*). Esto es muy útil para eliminar unidades de variables de texto que se extraen de una página.

`Number.parseInt("42"); // 42`

`Number.parseInt("42€"); // 42` (descarta todo desde un carácter no numérico)

`Number.parseInt("Núm. 42"); // NaN` (empieza a descartar en Núm, descarta también 42)

`Number.parseInt("A"); // NaN` (No se puede representar como un número)

`Number.parseInt(""); // NaN` (No se puede representar como un número)

`Number.parseInt("42.5"); // 42` (descarta los decimales)

`Number.parseInt("88.99€"); // 88` (descarta decimales y resto de caracteres)

`Number.parseInt("Núm. 33.5"); // NaN` (empieza a descartar en Núm, descarta todo)

.parseFloat() funciona igual, pero con números decimales, en lugar de números enteros. Si utilizamos `.parseInt()` con un número decimal, nos quedaremos sólo con la parte entera, mientras que con `.parseFloat()` conservará también la parte decimal (*si la tiene*).

`Number.parseFloat("42.5"); // 42.5` (Conserva decimales)

`Number.parseFloat("42"); // 42` (El número es entero, convierte a entero)

`Number.parseFloat("88.99€"); // 88.99` (Conserva decimales)

`Number.parseFloat("42€"); // 42` (El número es entero, convierte a entero)

`Number.parseFloat("Núm. 33.5"); // NaN` (empieza a descartar en Núm, descarta todo)

Tipos de datos: Number

Para **convertir un number en string** se utilizan la función `toString()`.

Ejemplos:

```
(16).toString(); // "16" (lo convierte a string)
```

```
(42.5).toString(); // "42.5" (lo convierte a string)
```

Tanto `toString` como `parseInt` o `parseFloat` pueden tener un parámetro adicional que permite convertir entre bases:

```
Number.parseInt("11101", 2); // 11101 en binario es 29 en decimal
```

```
Number.parseInt("31", 8); // 31 en octal es 25 en decimal
```

```
Number.parseInt("FF", 16); // FF en hexadecimal es 255 en decimal
```

```
(26).toString(2); // "11010" (26 en decimal, es 11010 en binario)
```

```
(80).toString(8); // "120" (80 en decimal, es 120 en octal)
```

```
(245123).toString(16); // "3bd83" (245123 en decimal, es 3bd83 en hexadecimal)
```

Tipos de datos: Objeto Math

Tiene incorporadas ciertas constantes y métodos (*funciones*) para trabajar matemáticamente.

Constantes de Math

Constante	Descripción	Valor
Math.E	Número de Euler	2.718281828459045
Math.LN2	Equivalente a $\text{Math.log}(2)$	0.6931471805599453
Math.LN10	Equivalente a $\text{Math.log}(10)$	2.302585092994046
Math.LOG2E	Equivalente a $\text{Math.log}_2(\text{Math.E})$	1.4426950408889634
Math.LOG10E	Equivalente a $\text{Math.log}_{10}(\text{Math.E})$	0.4342944819032518
Math.PI	Número PI o π	3.141592653589793
Math.SQRT1_2	Equivalente a $\text{Math.sqrt}(1/2)$.	0.7071067811865476
Math.SQRT2	Equivalente a $\text{Math.sqrt}(2)$.	1.4142135623730951

Tipos de datos: Objeto Math

Métodos matemáticos

Método	Descripción	Ejemplo
Math.abs(x)	Devuelve el valor absoluto de x.	x
Math.sign(x)	Devuelve el signo del número: 1 positivo, -1 negativo	
Math.exp(x)	Exponenciación. Devuelve el número e elevado a x.	e^x
Math.expm1(x)	Equivalente a $\text{Math.exp}(x) - 1$.	$e^x - 1$
Math.max(a, b, c...)	Devuelve el número más grande de los indicados por parámetro.	
Math.min(a, b, c...)	Devuelve el número más pequeño de los indicados por parámetro.	
Math.pow(base, exp)	Potenciación. Devuelve el número base elevado a exp.	base^{exp}
Math.sqrt(x)	Devuelve la raíz cuadrada de x.	\sqrt{x}
Math.cbrt(x)	Devuelve la raíz cúbica de x.	$\sqrt[3]{x}$
Math.imul(a, b)	Equivalente a $a * b$, pero a nivel de bits.	
Math.clz32(x)	Devuelve el número de ceros a la izquierda de x en binario (32 bits).	

Tipos de datos: Objeto Math

Método Math.random()

Genera un número al azar entre los valores 0 y 1, con 16 decimales.

// Obtenemos un número al azar entre [0, 1) con 16 decimales

let x = Math.random(); //

Si queremos obtener un número al azar entre un rango, debemos multiplicar el número anterior por el valor máximo que buscamos

//Entre 0 y 5 (no incluido):

x = x * 5;

// Redondeamos inferior

x=Math.floor(x);

Tipos de datos: Objeto Math

Métodos de logaritmos

Método	Descripción	Ejemplo
Math.log(x)	Devuelve el logaritmo natural en base e de x.	$\log_e x$ o $\ln x$
Math.log10(x)	Devuelve el logaritmo decimal (en base 10) de x.	$\log_{10} x$ ó $\log x$
Math.log2(x)	Devuelve el logaritmo binario (en base 2) de x.	$\log_2 x$
Math.log1p(x)	Devuelve el logaritmo natural de (1+x).	$\log_e (1+x)$ o $\ln (1+x)$

Métodos de redondeo

Método	Descripción
Math.round(x)	Devuelve x con redondeo (<i>el entero más cercano</i>)
Math.ceil(x)	Devuelve x con redondeo superior (<i>el entero más alto</i>)
Math.floor(x)	Devuelve x con redondeo inferior (<i>el entero más bajo</i>)
Math.fround(x)	Devuelve x con redondeo (<i>flotante con precisión simple</i>)
Math.trunc(x)	Trunca el número x (<i>devuelve sólo la parte entera</i>)

Métodos trigonométricos

Nos permiten hacer operaciones como **seno**, **coseno**, **tangente**...

Otras librerías matemáticas

Si de forma nativa no encuentras una forma sencilla

Tipos de datos: String

Cuando una variable posee información de texto decimos que su tipo de dato es String. Para utilizarlos es más sencillo utilizar los literales que la notación que utiliza la palabra clave new. Para englobar los textos, se pueden utilizar tres tipos de comillas:

- **Comillas simples:** '
- **Comillas dobles:** "
- **Backticks:** ` (*más adelante, en Interpolación de variables*)

// Notación literal (preferida)

```
const text = "¡Hola a todos!";
```

```
const message = "Otro mensaje de texto";
```

// Notación mediante objeto

```
const text = new String("¡Hola a todos!");
```

```
const message = new String("Otro mensaje de texto");
```

Propiedades de un string

.length devuelve el tamaño total de la cadena de texto en cuestión, como se puede ver en los siguientes ejemplos:

```
"Hola".length; // 4
```

```
"Adiós".length; // 5
```

```
"".length; // 0
```

```
"¡Yeah!".length; // 6
```

Nota: se han utilizado directamente, sin necesidad de guardarlos en una variable.

Tipos de datos: String

Acceso a un carácter

Con el operador [] indicando la posición a la que queremos acceder:

```
const text = "Hola";
```

```
text[0]; // "H"
```

```
text[4]; // undefined
```

Si intentamos acceder a una **posición** (*índice*) que no existe, nos devolverá undefined.

Métodos de un string

.repeat() repite el texto en cuestión el número de veces que le indicamos por parámetro:

```
const text = "Javascript";
```

```
text.repeat(3); // "JavascriptJavascriptJavascript"
```

Interpolación de variables ó **Templates**

Antes, si queríamos concatenar (*unir*) varias cadenas de texto, teníamos que hacer algo parecido a esto:

```
const firstWord = "frase"; const secondWord = "concatenada";
```

```
"Una " + firstWord + " bien " + secondWord; // 'Una frase bien concatenada'
```

En **ECMAScript** se introduce una mejora las **backticks** (*comillas hacia atrás*), que nos permitirán **interpol**ar el valor de las variables sin tener que cerrar, concatenar y abrir la cadena de texto continuamente:

```
const firstWord = "frase"; const secondWord = "concatenada";
```

```
`Una ${firstWord} mejor ${secondWord}`; // 'Una frase mejor concatenada'
```

Tipos de datos: String

Obtener posición o índice

Existen varios métodos que permiten darnos información sobre la **posición** o **ubicación** que ocupa un determinado carácter o texto. Como ya hemos mencionado, esta posición también suele denominarse **índice**. Veamos detalladamente dichos métodos:

Método	Descripción
.charAt(pos)	Devuelve el carácter de la posición pos. Similar al operador [].
.indexOf(text)	Devuelve la primera posición del texto text.
.indexOf(text, from)	Idem al anterior, partiendo desde la posición from.
.lastIndexOf(text)	Devuelve la última posición del texto text.
.lastIndexOf(text, from)	Idem al anterior, partiendo desde from hacia el inicio.

Tipos de datos: String

Ejemplos:

```
const name = "Mario"; // Utilizando .charAt
name.charAt(); // 'M'
name.charAt(0); // 'M'
name.charAt(1); // 'a'
name.charAt(10); // "
// Utilizando operador []
name[]; // ERROR
name[0]; // 'M'
name[1]; // 'a'
name[10]; // undefined
```

```
const phrase = "LenguajeJS, página de Javascript";
phrase.indexOf("n"); // 2
phrase.indexOf("n", 3); // 16
phrase.indexOf("n", 17); // -1
```

Tipos de datos: String

Obtener fragmentos (substrings)

Método	Descripción
.repeat(num)	Devuelve el repetido num veces.
.substring(start, end)	Devuelve el substring desde la posición start hasta end.
.substr(start, size)	Devuelve el substring desde la posición start hasta start+size.
.slice(start, end)	Idem a .substr() con leves diferencias.

Ejemplo:

```
const text = "Submarino";  
text.substring(3); // 'marino' (desde el 3 en adelante)  
text.substring(3, 5); // 'ma' (desde el 3, hasta el 5)  
text.substr(3); // 'marino' (desde el 3 en adelante)  
text.substr(3, 5); // 'marin' (desde el 3, hasta el 3+5)  
text.substr(-3); // 'ino' (desde la posición 3 desde el final, en adelante)  
text.substr(-3, 2); // 'in' (desde la posición 3 desde el final, hasta 2 posiciones más)
```

El método .substr() con un valor negativo en su primer parámetro start, empieza a contar desde el final. Esto es algo que no ocurre con el método .substring().

Tipos de datos: String

Dividir un texto en partes (array)

Método	Descripción
<code>.split(text)</code>	Separa el texto en varias partes, usando text como separador.
<code>.split(text, limit)</code>	Idem, pero crea como máximo limit fragmentos.
<code>.split(regex)</code>	Separa el texto usando la regex como separador.
<code>.split(regex, limit)</code>	Idem, pero crea como máximo limit fragmentos.

Ejemplo:

`"88.12.44.123".split(".");` // ["88", "12", "44", "123"] (4 elementos)

`"1.2.3.4.5".split(".");` // ["1", "2", "3", "4", "5"] (5 elementos)

`"Hola a todos".split(" ");` // ["Hola", "a", "todos"] (3 elementos)

`"A,B,C,D,E".split(",", 3);` // ["A", "B", "C"] (limitado a los 3 primeros elementos)

`"Código".split("");` // ["C", "ó", "d", "í", "g", "o"] (6 elementos) . El separador es una **cadena vacía**, es decir, «separar por la unidad más pequeña posible». Al indicar esto, `.split()` realiza una división carácter por carácter

// Separa tanto por punto como por coma:

`"88.12,44.123".split(/[.,]/);` // ["88", "12", "44", "123"] (4 elementos)

Tipos de datos: String

Comprobación en textos

Método	Descripción
.startsWith(text, from)	Comprueba si el texto comienza por text.
.endsWith(text, to)	Comprueba si el texto termina por text.
.includes(text, from)	Comprueba si el texto contiene el subtexto text.

Cada método tienes un **segundo parámetro opcional**, donde se puede indicar desde donde quieres empezar a comprobar (*en el caso de from*), o hasta donde quieres comprobar (*en el caso de to*).

Búsqueda de cadenas de textos

Método	Descripción
.search(regex)	Busca un patrón que encaje con regex y devuelve la posición encontrada.
.match(regex)	Idem a la anterior, pero devuelve las coincidencias encontradas.
.matchAll(regex)	Idem a la anterior, pero devuelve un iterador para iterar por cada coincidencia.

Tipos de datos: String

Reemplazar cadenas de texto

Método	Descripción
<code>.replace(text, newText)</code>	Reemplaza la primera aparición del text por newText.
<code>.replace(regex, newText)</code>	Idem, pero busca a partir de una regexp en lugar de un string.
<code>.replaceAll(text, newText)</code>	Reemplaza todas las apariciones del texto text por newText.
<code>.replaceAll(regex, newText)</code>	Idem, pero busca a partir de una regexp en lugar de un string

text no cambia (*no muta*), es decir, el método `.replace()` devuelve un nuevo con el texto original reemplazado.

Tipos de datos: String

Modificar strings

Método	Descripción
<code>.toLowerCase()</code>	Devuelve el string transformado a minúsculas.
<code>.toUpperCase()</code>	Devuelve el string transformado a mayúsculas.
<code>.padStart(size, text)</code>	Devuelve el string rellenando el inicio con text hasta llegar al tamaño size.
<code>.padEnd(size, text)</code>	Devuelve el string rellenando el final con text hasta llegar al tamaño size.
<code>.trimStart()</code>	Devuelve el string eliminando espacios a la izquierda del texto.
<code>.trimEnd()</code>	Devuelve el string eliminando espacios a la derecha del texto.
<code>.trim()</code>	Devuelve el string eliminando espacios a la izquierda y derecha del texto.

Puede que en algunos lugares te encuentres los métodos `.trimLeft()` (*izquierda*) y `.trimRight()` (*derecha*) como alternativas a `.trimStart()` y `.trimEnd()`. En principio, son una implementación obsoleta que no debería usarse.

Concatenar strings

Método	Descripción
<code>.concat(text1, text2...)</code>	Devuelve el string unido (concatenado) a las variables text1, text2...

Operadores

Operadores aritméticos

+	Suma
-	Resta
*	Multiplicación
/	División
%	Resto de una división

Operadores condicionales

Igualdad y desigualdad	== y !=
Igualdad y desigualdad estricta (*)	=== y !==
Mayor y menor	> y <
Mayor o igual y menor o igual	>= y <=

Operadores lógicos

NO (NOT)	!
Y (AND)	&&
O (OR)	

(*) true si los operadores son iguales y del mismo tipo

Operadores de asignación

+=	Ejecuta una suma y asigna el valor al operando de la izquierda
-=	Ejecuta una resta y asigna el valor al operando de la izquierda
*=	Ejecuta una multiplicación y asigna el valor al operando de la izquierda
/=	Ejecuta una división y asigna el valor al operando de la izquierda
%=	Ejecuta el módulo y asigna el valor al operando de la izquierda

Operadores

Operador condicional (ternario)

El operador condicional es el único operador de JavaScript que toma tres operandos. El operador puede tener uno de dos valores según una condición. La sintaxis es:

`condition ? val1 : val2`

Ejemplo:

```
let status = age >= 18 ? "adult" : "minor";
```

Operadores

También se puede utilizar el operador ternario de forma anidada:

```
let nota = 7;  
console.log("He realizado mi examen.");  
let calificacion =  
    nota < 5 ? "Insuficiente" :  
    nota < 6 ? "Suficiente" :  
    nota < 8 ? "Bien" :  
    nota <= 9 ? "Notable" :  
    "Sobresaliente";  
console.log("He obtenido un", calificacion);
```

Estructuras de control

if –else	Ejecuta un bloque de instrucciones si la condición es verdadera y otro en caso de falso	if (condición1) sentencia1 else if (condición2) sentencia2 else if (condición3) sentencia3 //... else sentencia
do-while	Ejecuta una serie de instrucciones, al menos una vez, hasta que la condición que se evalúa toma valor falso	do expresión while (condición);
while	Ejecuta un bloque de instrucciones repetidamente mientras la condición sea verdadera	while (condición) expresión
for	Ejecuta un bloque de instrucciones recorriendo el rango de valores de forma secuencial hasta que la condición deja de cumplirse	for ([expresiónInicial]; [expresiónCondicional]; [expresiónDeActualización]) instrucción

Estructuras de control

for await	Crea un bucle que itera sobre objetos iterables asíncronos, así como en iterables sincronizados	
for ... in	Recorre las propiedades de un objeto	for (variable in objeto) instrucción
for ... of	Recorre los valores de un objeto	for (variable of objeto) expresión

Nota: Para ejecutar varias instrucciones, usa una declaración de bloque { ... } para agrupar esas declaraciones. Las instrucciones, como siempre, terminan en ;

Ejemplo for:

```
for (let i = 0; i < 9; i++)  
{ n += i;  
  Console.log(n);  
}
```

Ejemplo while:

```
n = 0; x = 0;  
while (n < 3)  
{ n++;  
  x += n;  
}
```

Estructuras de control

Switch

La estructura switch permite reemplazar a estructuras if then else if.

Consiste en evaluar una expresión, y dependiendo del valor, ejecuta un bloque de código u otro.

```
switch (expresión) {  
    case valor1:  
        // Bloque de código a ejecutar si la expresión es igual a valor1  
        break;  
    case valor2:  
        // Bloque de código a ejecutar si la expresión es igual a valor2  
        break;  
    default:  
        // Bloque de código a ejecutar si la expresión no coincide con ninguno de los  
valores  
}
```

Estructuras de control

Ejemplo:

```
let animal = prompt("Escribe un animal");
let salida="";
switch (animal) {
  case "perro":
    salida="Este animal es un perro";
    break;
  case "gato":
    salida="Este animal es un gato";
    break;
  case "leon":
    salida="Este animal es un leon";
    break;
  default:
    salida="Debes introducir otro animal";
    break;
}
console.log(salida);
```

Estructuras de control

Switch(true)

La estructura switch puede tener una variante con el switch(true). El caso es que la expresión a evaluar dentro de la sentencia **switch** no solo puede ser un valor sino también una expresión en sí. De manera que en caso de que dicha expresión sea true, se ejecuta el bloque de código correspondiente.

```
switch (true) {  
    case condicion1:  
        // Bloque de código a ejecutar si la condicion1 es true  
        break;  
    case condicion2:  
        // Bloque de código a ejecutar si la condicion2 es true  
        break;  
    default:  
        // Bloque de código a ejecutar si la expresión no coincide con ninguno de los  
valores  
}
```

Estructuras de control

Ejemplo:

```
let edad = prompt("Escribe tu edad");
let salida="";
numero = Number.parseFloat(edad);
switch (true) {
  case(numero >= 0 && numero <13):
    salida="Niño/a";
    break;
  case(numero >= 13 && numero <=16):
    salida="Adolescente";
    break;
  case(numero > 16 && numero < 30):
    salida="Joven";
    break;
  case(numero >= 30):
    salida="Edad adulta";
    break;
  default:
    alert("Tienes que introducir un número");
    break;
}
console.log(`La edad ${numero} se corresponde con ${salida}`);
```

Estructuras de control

Continue

Sentencia que al llegar a ella dentro de un bucle, el programa **salta** y abandona esa iteración, siguiendo por la siguiente iteración:

```
for (let i = 0; i < 11; i++) {  
    if (i == 5) {  
        continue;  
    }  
    console.log("Valor de i:", i);  
}
```

Se realizan las iteraciones desde 0 hasta 4, en la iteración del 5 se ejecuta el continue y salta a la siguiente iteración, por lo que continuará desde la iteración 6 hasta la 10.

break

Nos permite interrumpir el bucle y abandonarlo.

```
for (let i = 0; i < 11; i++) {  
    if (i == 5) {  
        break;  
    }  
    console.log("Valor de i:", i);  
}
```

Se van a realizar las iteraciones desde el 0 al 4, y cuando lleguemos a la iteración 5, se entrará en el if y como cumple su condición se abandonará el for, y continuará con el resto del programa

Entrada y Salida en el navegador

Mensajes en la consola (Visto anteriormente)

Con el objeto console se pueden mostrar mensajes con diferente estética.

Mensajes de confirmación

```
let respuesta=confirm("¿Estás seguro de querer eliminar?");  
Console.log(`Respuesta del cuadro de diálogo: ${respuesta}`);
```

¿Estás seguro de querer eliminar?

Aceptar

Cancelar

Entrada y Salida en el navegador

Mensajes de entrada

```
let respuesta=prompt("Para eliminar escribe ELIMINAR");  
Console.log(`El usuario escribió: ${respuesta}`);
```

Para eliminar escribe ELIMINAR

El prompt devuelve null cuando el usuario pulsa Cancelar

Mensajes de alerta

```
alert("Mensaje de alerta");
```

Mensaje de alerta

OJO!!! Los mensaje de alerta, cuadros de confirmación y mensajes de entrada son útiles para el aprendizaje, pero son cosas del pasado

Variables

Variables. Una variable de JavaScript puede contener un valor de cualquier tipo de datos.

Esta característica se denomina **escritura dinàmica**

En principio, existen dos ámbitos muy bien definidos:

- **Ámbito global:** Existe a lo largo de todo el programa o aplicación.
- **Ámbito local:** Existe sólo en una pequeña región del programa. Normalmente, esto se determina muy fácilmente observando las llaves que abren un nuevo ámbito o contexto.

Ámbitos de variables

```
let a = 1;
console.log(e); // Uncaught ReferenceError: e is not defined
if (a == 1)
{
    let e = 40;
    console.log(e); // 40, existe
}
console.log(e); // Uncaught ReferenceError: e is not defined
```

La variable a existe en un ámbito global, y la variable e sólo existe en el interior del if: es un ámbito local.

Variables

Ámbitos de variables: let y const

A partir de ECMAScript 2015 se introduce la palabra clave `let` en sustitución de la antigua `var` para declarar variables y `const` para declarar constantes. Tanto con `let` como con `const`, estaremos utilizando los ámbitos clásicos de programación: **ámbito global y ámbito local**.

```
console.log("Antes: ", p); // En este punto, p no está definida
for (let p = 0; p < 3; p++)
{
    console.log("Valor de p: ", p); // Aquí, p estará definida como 0, 1, 2
}
console.log("Después: ", p); // En este punto, vuelve a no estar definida
```

1. Utilizando `let` en el bucle `for`, la variable `p` sólo está definida dentro del bucle (*ámbito local*)
2. Tanto antes como después del bucle, `p` no existe.

Variables

Ámbitos de variables: var (legacy)

Declarar variables con var ya se considera legacy (*obsoleto*) y no debe usarse.

```
console.log("Antes:", p); // En este punto, p vale undefined
for (var p = 0; p < 3; p++) {
    console.log("Valor de p: ", p); // Aquí, p estará definida como 0, 1, 2
}
console.log("Después: ", p); // Después: 3
```

Si utilizamos var la variable p sigue existiendo fuera del bucle, aunque haya sido declarada en su interior. Esto ocurre porque en ese caso se usa un ámbito a **nivel de función**.

Variables

```
var a = 1;  
console.log(a); // Aquí accedemos a la "a" global, que vale 1  
function x() {  
    console.log(a); // En esta línea el valor de "a" es undefined  
    var a = 5; // Aquí creamos una variable "a" a nivel de función  
    console.log(a); // Aquí el valor de "a" es 5 (a nivel de función)  
    console.log(window.a); // Aquí el valor de "a" es 1 (ámbito global)  
}  
x(); // Aquí se ejecuta el código de la función x()  
console.log(a); // En esta línea el valor de "a" es 1
```

El valor de `a` dentro de una función no es el 1 inicial, sino que estamos en otro ámbito diferente donde la variable `a` anterior no existe: un **ámbito a nivel de función**. Mientras estemos dentro de una función, las variables inicializadas en ella estarán en el **ámbito** de la propia función.

Estamos usando el objeto especial `window` para acceder directamente al ámbito global independientemente de donde nos encontremos. Esto ocurre así porque las variables globales se almacenan dentro del objeto `window` (*la pestaña actual del navegador web*). Hoy en día, en lugar de utilizar `window` sería preferible utilizar `globalThis`.

Variables

Constantes

const crea una referencia de solo lectura a un valor

Se añade en ECMAScript 2015.

Las constantes son de ámbito de bloque, al igual que las variables definidas mediante la instrucción **let**

- Las constantes no pueden ser reasignadas a un valor
- Una constante no puede ser re-declarada
- Una constante requiere un inicializador. Esto significa que las constantes deben inicializarse durante su declaración
- El valor asignado a una variable **const** es inmutable

```
const name = "Pedro";  
console.log(name);  
name = "Paco"; // Uncaught TypeError: Assignment to constant variable.
```

La diferencia respecto al **let** es que **const** asigna un valor y no permite reasignarlo o redeclararlo posteriormente. Además, tampoco puedes crear una constante sin indicarle un valor concreto.

Algunas constantes (*objetos, arrays...*) si que pueden ser alteradas, aunque nunca reasignadas.

Variables

Constantes

Si le asignamos un objeto a una constante, éste no será totalmente inmutable ya que le podremos cambiar los valores de sus propiedades y métodos. Lo mismo sucederá con un *array*, al que le podremos 'mutar' el valor de cada uno de sus elementos

```
const OBJETO = {nombre:'Maria'};  
OBJETO = {nombre:Maria',apellido:'Garcia'};  
console.log(OBJETO.nombre);// No se puede  
alterar la estructura de un objeto. Dará error
```

```
const OBJETO = {  
  name: 'David'  
}  
OBJETO.name = 'Miguel';  
console.log(OBJETO.name);//Miguel
```

Las propiedades del objeto en sí se pueden modificar, añadir o borrar

```
const OBJETO = {nombre:'Rosa'};  
OBJETO.apellido = 'Garcia';  
console.log(OBJETO.nombre + ' '+OBJETO.apellido);
```

La constante es el propio objeto, no sus propiedades. El objeto en sí está con mayúsculas pero sus propiedades no, porque no son constantes.

Variables

Constantes

Otra excepción son los arrays, ya que podremos añadir y eliminar elementos a un array definido con `const`

```
const MISaLUMNOS = ['Antonio', 'Luis'];  
MISaLUMNOS.push ('Pedro');  
console.log (MISaLUMNOS);
```

Podemos **congelar** un objeto o array utilizando el método `freeze` que no nos permitirá cambiar ni propiedades ni valores

```
const OBJETO = {nombre: 'Rosa'};  
const NUEVOOBJETO = Object.freeze(OBJETO);  
NUEVOOBJETO.apellido = 'Garcia';  
console.log(NUEVOOBJETO.nombre) ;
```

Arrays

¿Qué es un array?

Un es una colección o agrupación de elementos en una misma variable, cada uno de ellos ubicado por la posición que ocupa en el array. Se pueden definir de varias formas:

// Forma tradicional (no se suele usar en Javascript)

```
const letters = new Array("a", "b", "c"); // Array con 3 elementos
```

```
const letters = new Array(3); // Array vacío de tamaño 3
```

// Mediante literales (notación preferida)

```
const letters = ["a", "b", "c"]; // Array con 3 elementos
```

```
const letters = []; // Array vacío (0 elementos)
```

```
const letters = ["a", 5, true]; // Array mixto (String, Number, Boolean)
```

Javascript permite que se puedan realizar arrays de **tipo mixto** (elementos de diferente tipo).

El operador []

Sirve para acceder a elementos del array.

La propiedad .length no dice el tamaño del array

```
const letters = ["a", "b", "c"];
```

```
letters.length; // 3
```

```
letters[0]; // 'a'
```

```
letters[5]; // undefined
```

Arrays

Para modificar elementos del array:

```
letters[1] = "Z"; // modifica letters a ["a", "Z", "c"]
```

Además del operador [], podemos utilizar el método .at(), añadido en ES2022 . Con él, se puede hacer exactamente lo mismo que con [pos], sólo que además permite valores negativos, mediante los cuales se puede obtener elementos en orden inverso, es decir, empezando a contar desde el último elemento:

```
const letters = ["a", "b", "c"];
```

```
letters.at(0); // "a"
```

```
letters.at(-1); // "c"
```

Añadir o eliminar elementos

- Los métodos .push() y .pop() actúan al **final del array**.
- Los métodos .unshift() y .shift() actúan al **inicio del array**.

Los métodos de inserción .push() o .unshift() insertan los elementos pasados por parámetro en el array y devuelve el tamaño actual del array después de la inserción. Los métodos de extracción, .pop() o .shift(), extraen y devuelven el elemento extraído.

```
const elements = ["a", "b", "c"]; // Array inicial
```

```
elements.push("d"); // Devuelve 4. Ahora elements = ['a', 'b', 'c', 'd']
```

```
elements.pop(); // Devuelve 'd'. Ahora elements = ['a', 'b', 'c']
```

```
elements.unshift("Z"); // Devuelve 4. Ahora elements = ['Z', 'a', 'b', 'c']
```

```
elements.shift(); // Devuelve 'Z'. Ahora elements = ['a', 'b', 'c']
```

Arrays

Aunque hemos visto las formas principales de crear un en Javascript, existen otras que permiten generar un partiendo de otras estructuras de datos.

Convertir a array

El método estático `Array.from()` se utiliza para convertir variables «parecidas» a los **arrays** (*pero que no son arrays*) en un Array. Este es el caso de variables como Strings o de la lista de elementos del DOM.

```
const texto = "12345";  
console.log(texto.constructor.name); // "String"  
const letras = Array.from(texto); // ["1", "2", "3", "4", "5"]  
console.log(letras);  
const letras2 = [...texto]; // ["1", "2", "3", "4", "5"]  
console.log(letras2);
```

```
String  
▶ Array(5) [ "1", "2", "3", "4", "5" ]  
1 2 3 4 5  
1 2 3 4 5  
▶ Array(5) [ "1", "2", "3", "4", "5" ]  
..
```

¿Qué es el operador ...?

Es el operador spread, y sirve para pasar un array o un strig a una lista de argumentos

```
const texto = "12345";  
console.log(...texto); //  
const letras = ["1", "2", "3", "4", "5"];  
console.log(...letras);
```

```
1 2 3 4 5  
1 2 3 4 5
```

Arrays

Concatenar arrays

El método `concat()`, nos permite unir los elementos pasados por parámetro en un array a la estructura que estamos manejando.

Se podría pensar que los métodos `.push()` y `concat()` funcionan de la misma forma, pero no es exactamente así:

```
const elements = [1, 2, 3];  
elements.push(4, 5, 6); // Devuelve 6. Ahora elements = [1, 2, 3, 4, 5, 6]  
elements.push([7, 8, 9]); // Devuelve 7. Ahora elements = [1, 2, 3, 4, 5, 6, [7, 8, 9]]
```

El mismo ejemplo con el método `.concat()`:

```
const firstPart = [1, 2, 3];  
const secondPart = [4, 5, 6];  
firstPart.concat(firstPart); // Devuelve [1, 2, 3, 1, 2, 3]  
firstPart.concat(secondPart); // Devuelve [1, 2, 3, 4, 5, 6] // Se pueden pasar elementos sueltos  
firstPart.concat(4, 5, 6); // Devuelve [1, 2, 3, 4, 5, 6] // Se pueden concatenar múltiples arrays e  
incluso mezclarlos con elementos sueltos  
firstPart.concat(firstPart, secondPart, 7); // Devuelve [1, 2, 3, 1, 2, 3, 4, 5, 6, 7]
```

El método `concat()`, a diferencia de `push()`, no modifica el array sobre el cuál trabajamos, sino que simplemente lo devuelve. Además, al pasar un array por parámetro, `push()` lo inserta como un array, mientras que `concat()` inserta cada uno de sus elementos.

Arrays

Separar y unir strings

El método `.split()` permite crear un Array a partir de un String. El método `.join()` es su contrapartida. Con `.join()` podemos crear un String con todos los elementos del array, separándolo por el texto que le pasemos por parámetro:

```
const letters = ["a", "b", "c"];  
letters.join("->"); // Devuelve 'a->b->c'  
letters.join("."); // Devuelve 'a.b.c'
```

```
"a.b.c".split("."); // Devuelve ['a', 'b', 'c']  
"5-4-3-2-1".split("-"); // Devuelve ['5', '4', '3', '2', '1']
```

`.join()` devolverá un String y `.split()` devolverá un Array.

```
"Hola a todos".split(""); // ['H', 'o', 'l', 'a', ' ', 'a', ' ', 't', 'o', 'd', 'o', 's']
```

En este caso, hemos pedido dividir sin indicar ningún separador, por lo que toma la unidad mínima como separador y nos devuelve un array con cada carácter del string original.

Arrays

Buscar elementos en un array

Método	Descripción
.includes(element)	Comprueba si element está incluido en el array.
.includes(element, from)	Idem, pero partiendo desde la posición from del array.
.indexOf(element)	Devuelve la posición de la primera aparición de element o -1 si no existe.
.indexOf(element, from)	Idem, pero partiendo desde la posición from del array.
.lastIndexOf(element)	Devuelve la posición de la última aparición de element. Devuelve -1 si no existe.
.lastIndexOf(element, from)	Idem, pero partiendo desde la posición from del array.

Ejemplos:

```
const numbers = [5, 10, 15, 20, 25, 10];  
numbers.includes(3); // false  
numbers.includes(15); // true  
numbers.includes(15, 4); // false  
numbers.indexOf(5); // 0  
numbers.indexOf(15, 4); // -1  
numbers.lastIndexOf(10); // 5  
numbers.lastIndexOf(10, 3); // 1
```

Arrays

Ordenacion de arrays

Método	Descripción
<code>.reverse()</code> ⚠	Invierte el orden de elementos del array.
<code>.toReversed()</code> ✓	Devuelve una copia del array, con el orden de los elementos invertido.
<code>.sort()</code> ⚠	Ordena los elementos del array bajo un criterio de ordenación alfabética .
<code>.sort(criterio)</code> ⚠	Idem, pero bajo un criterio de ordenación indicado por criterio.
<code>.toSorted()</code> ✓	Devuelve una copia del array, con los elementos ordenados.
<code>.toSorted(criterio)</code> ✓	Idem, pero ordenado por el criterio establecido por parámetro.

✓ El array original está seguro (*no muta*).

⚠ El array original cambia (*muta*).

Ejemplo:

```
const elements = ["A", "B", "C", "D", "E", "F"];  
const reversedElements = elements.reverse();  
reversedElements // ["F", "E", "D", "C", "B", "A"]  
elements // ["F", "E", "D", "C", "B", "A"]  
reversedElements === elements // true
```

El array original `elements` ha mutado, y el nuevo array resultante no es más que una referencia al original.

Arrays

Si queremos crear un nuevo array independiente del original, tendríamos que hacer lo siguiente:

```
const elements = ["A", "B", "C", "D", "E", "F"];  
const reversedElements = structuredClone(elements).reverse();  
reversedElements // ["F", "E", "D", "C", "B", "A"]  
elements // ['A', 'B', 'C', 'D', 'E', 'F']  
reversedElements === elements // false
```

Con `structuredClone()` creamos una copia de la estructura original `elements`, y luego se invierte. Ahora tenemos dos arrays independientes.

Otra solución es utilizar el nuevo método `.toReversed()`, ES2023, el cuál funciona exactamente igual que `.reverse()`, pero sin mutar el original:

```
const elements = ["A", "B", "C", "D", "E", "F"];  
const reversedElements = elements.toReversed();  
reversedElements // ["F", "E", "D", "C", "B", "A"]  
elements // ["A", "B", "C", "D", "E", "F"]  
reversedElements === elements // false
```

Arrays

```
const names = ["Alberto", "Zoe", "Ana", "Mauricio", "Bernardo"];  
const sortedNames = names.sort();  
sortedNames // ["Alberto", "Ana", "Bernardo", "Mauricio", "Zoe"]  
names // ["Alberto", "Ana", "Bernardo", "Mauricio", "Zoe"]  
sortedNames === names // true
```

Usamos `structuredClone()` para hacer una copia y que sea independiente:

```
const names = ["Alberto", "Zoe", "Ana", "Mauricio", "Bernardo"];  
const sortedNames = structuredClone(names).sort();  
sortedNames // ["Alberto", "Ana", "Bernardo", "Mauricio", "Zoe"]  
names // ["Alberto", "Zoe", "Ana", "Mauricio", "Bernardo"];  
sortedNames === names // false
```

Utilizando `.toSorted()` se mantiene el array original.

Si intentamos ordenar un array de elementos `Number`, falla la ordenación:

```
const numbers = [1, 8, 2, 32, 9, 7, 4];  
const sortedNumbers = numbers.toSorted();  
sortedNumbers // [1, 2, 32, 4, 7, 8, 9]  
numbers // [1, 8, 2, 32, 9, 7, 4]
```

Esto es porque `sort` trabaja con la ordenación alfabética, y ahora necesitamos ordenación natural. Se puede arreglar pasando una función de Comparación a `sort` (MÁS ADELANTE)

Arrays

Crear array bidimensional:

```
let tablaNotas=[[,],[,]]; //cada nivel de anidamiento de corchetes indica una dimensión del array, y el número de elementos separados por comas es el número de elementos de esa dimensión
```

Para almacenar valores:

```
tablaNotas[0][0]=1; //Fila 0-Columna 0  
tablaNotas[0][1]=2; //Fila 0-Columna 1  
tablaNotas[0][2]=3; //Fila 0-Columna 2  
tablaNotas[1][0]=4; //Fila 1-Columna 0  
tablaNotas[1][1]=5; //Fila 1-Columna 1  
tablaNotas[1][2]=6; //Fila 1-Columna 2
```

Otra forma:

```
let tablaNotas=new Array(2);  
tablaNotas[0]=new Array(3);  
tablaNotas[1]=new Array(3);
```

A partir de ahí, se rellenan los valores como antes.

Arrays

Otras formas de recorrer un array:

For in

```
let precios=[60,12,99,35,76];  
for (let i in precios){//no es necesario inicializar el contador ni incrementarlo  
    console.log(`El precio ${i} es:${precios[i]}`);  
}
```

For of: simplifica más el proceso, ya que ni siquiera se utiliza una variable para iterar por posición, ya que se realiza automáticamente. Desventaja: se desconocen los índices y que en la salida se muestran los elementos vacíos

```
let precios=[60,12,99,35,76];  
for (let precio of precios){  
    console.log(precio);  
}
```

forEach: Se verá más adelante con las funciones

Conjuntos

Los conjuntos o sets son estructuras de datos parecidas a los arrays, pero que no permiten valores duplicados. Previene esa duplicidad de forma automática.

Crear un conjunto: usando new, ya que se trata de un objeto

```
Let conjunto=new Set();
```

Para indicarle, desde su declaración, los elementos que lo componen inicialmente, hay que pasarle un objeto de tipo iterable (array, map, string, set...)

```
let conjunto1=new Set([34,1,"Girasol",25.9]);  
let conjunto2=new Set("cadena");
```

```
► Set(4) [ 34, 1, "Girasol", 25.9 ]
```

```
► Set(5) [ "c", "a", "d", "e", "n" ]
```

Automáticamente ha eliminado los elementos duplicados

Recorrido: con for of

```
For (let elemento of conjunto1){  
    console.log(elemento);  
}
```

Conjuntos

Adición de elementos: con add

```
let conjunto=new Set();  
conjunto.add(7);  
conjunto.add("Samuel").add(70).add("hola");  
//conjunto{7,"Samuel",70,"hola"}
```

Eliminación de elementos: delete, y devuelve true o false indicando el resultado de la operación.

```
conjunto.delete(70);  
//conjunto{7,"Samuel","hola"}
```

Para eliminar todos los elementos se usa clear

```
conjunto.clear();
```

Tamaño de un conjunto: size

```
let conjunto=new Set().add(1).add(2).add(3).add(2);  
console.log(conjunto.size);  
//3
```

Conjuntos

Búsqueda de un elemento: has. Devuelve true si se ha encontrado el elemento

```
let conjunto=new Set().add(1).add(2).add(3).add(2); If  
(conjunto.has(3))  
    console.log("Encontrado");
```

Conversiones: al principio, vimos cómo se usaba el operador spread, ahora lo utilizamos para convertir un array a un conjunto. Set, al ser una estructura iterable (*se puede recorrer*), es muy sencillo de utilizar con **desestructuración** y convertirlo a un array (*o viceversa*).

```
const conjunto = new Set([5, "A", [99, 10, 24]]);  
conjunto.size; // 3 (Contiene 3 elementos)  
conjunto.constructor.name; // "Set"  
const vector= [...conjunto];  
vector.constructor.name; // "Array"  
vector; // [5, "A", [99, 10, 24]]
```

Unión: con el operador spread:

```
let array1=[10,20,30,40,50]; let  
array2=[30,50,60,70,80]; let  
array3=[60,70,80,90,100];  
let conjunto=new Set([...array1,...array2,...array3]);  
//conjunto {10,20,30,40,50,60,70,80,90,100}
```

Conjuntos

Cuidado cuando se tengan conjuntos con tipos de datos más complejos con elementos anidados (*arrays, objetos, etc...*), ya que son referencias y modificar un elemento referenciado, modificará el original. Para evitar esto: `structuredClone()`:

```
const conjunto = new Set([5, "A", [99, 10, 24]]);
conjunto.size; // 3
const clonedArray = [...structuredClone(conjunto)];
const array = [...conjunto];
clonedArray[2][0] = "Modified";
[...conjunto][2][0]; // 99 (El original se mantiene intacto)
array[2][0] = "Modified";
[...conjunto][2][0]; // "Modified" (El original ha mutado)
```

Como ya se ha visto, se puede hacer la operación inversa para convertir un array en un set:

```
const array = [5, 4, 3, 3, 4];
const conjunto = new Set(array);
conjunto; // Set({ 5, 4, 3 })
```

Hay más operaciones con Conjuntos: Intersección, Diferencia y Exclusión. SE VERÁN MÁS ADELANTE.

WeakSet

Es una estructura de datos muy parecida a los Sets (no permiten datos duplicados), pero no permiten elementos primitivos.

```
// *** Set
const conjunto = new Set([1, "A", true]); // OK
const conjunto2 = new Set([{ name: "Mario" }, [2, 30]]); // OK
// *** WeakSet
const wconjunto = new WeakSet([1, "A", true]); // ERROR: Uncaught TypeError: Invalid value
                                                used in weak set
const wconjunto2 = new WeakSet([{ name: "Mario" }, [2, 30]]); // OK
```

Maps

Un mapa en Javascript es una estructura de datos nativa que permiten implementar una estructura de tipo **mapa**, es decir, una estructura que tiene **valores** guardados a través de una **clave** para identificarlos. Comúnmente, esto se denomina **pares clave-valor**.

Crear un mapa

```
let map = new Map(); // Mapa vacío
let map = new Map([[1, "uno"]]); // Map({ 1=>"uno" })
let map = new Map([[1, "uno"], [2, "dos"], [3, "tres"]]); // Map({ 1=>"uno", 2=>"dos", 3=>"tres" })
```

En el tercer caso se pasa un array de entradas, o array de array.

Recorrido de un mapa: for..of

```
let telefonos= new Map([
    [666666666,"Elena"],
    [655555555,"Mario"],
    [644444444,"Marta"]
])
for(let persona of telefonos)
    console.log(persona);
```



```
► Array [ 666666666, "Elena" ]
► Array [ 655555555, "Mario" ]
► Array [ 644444444, "Marta" ]
```

Esta forma no es muy útil para tratar datos por separado

Maps

Para obtener en diferentes variables las claves y los valores:

```
for (let [teléfono,persona] of teléfonos)  
    console.log(`El teléfono de ${persona} es ${telefono}.`);
```

```
El telefono de Elena es 666666666.  
El telefono de Mario es 655555555.  
El telefono de Marta es 644444444.
```

Además, si sólo se tiene que trabajar con las claves o con los valores, también se pueden usar los métodos: keys y values

```
for(let telefono of telefonos.keys())  
    console.log(telefono);
```

```
666666666  
655555555  
644444444
```

```
for(let persona of telefonos.values())  
    console.log(persona);
```

```
Elena  
Mario  
Marta
```

Maps

size: devuelve tamaño del map

```
const mapa = new Map();  
mapa.size; // 0  
const mapa2 = new Map([[1, "uno"], [2, "dos"]]);  
mapa2.size; // 2  
const mapa3 = new Map([[1, "uno"], [2, "dos"], [1, "tres"]]);  
mapa3.size; // 2 (El 1->"tres" sobrescribe al anterior)
```

Observa que si introducimos un nuevo par clave-valor que tiene la misma clave que otro (*tenemos dos que comparten la clave 1*), se sobrescribirá. No pueden existir dos pares clave-valor con la misma clave.

Establecer elementos (set) : fija un par clave-valor en el mapa.

- Si usamos .set() para una **clave** que no existe, se añade al mapa.
- Si usamos .set() para una **clave** que ya existe, la sobrescribe.

```
const mapa = new Map();  
mapa.set(5, "cinco");  
mapa.set("A", "letra A");  
mapa.set(5, "cinco sobrescrito"); // Sobrescribe el anterior map; // Map({ 5=>"cinco sobrescrito",  
"A"=>"letra A" })
```

Los Mapas pueden utilizar como clave cualquier tipo de dato.

Maps

Comprobar si existen (has): si la clave existe devuelve true, si no false

```
const mapa = new Map([[1, "uno"], [2, "dos"], [3, "tres"]]);  
mapa.has(2); // true  
mapa.has(34); // false  
mapa.set(34, "treinta y cuatro");  
mapa.has(34); // true
```

Si se está utilizando tipos de datos más complejos como objetos o arrays , se deberían tener almacenados en una variable, ya que si se crean al momento de pasarlos por parámetro, se pasa su referencia, y podrían no ser los mismos objetos aunque se escriban igual.

Borrar elementos (delete): Devuelve true si lo consigue eliminar, en caso contrario, false.

```
const mapa = new Map([[1, "uno"], [2, "dos"], [3, "tres"]]);  
mapa.delete(3); // true  
mapa.delete(39); // false  
mapa; // Map({ 1=>"uno", 2=>"dos" })
```

Vaciar conjunto (clear): borra todos los elementos del mapa, dejándolo vacío.

```
const mapa = new Map([[1, "uno"], [2, "dos"], [3, "tres"]]);  
mapa.clear();  
mapa.size; // 0
```

Obtener el valor a partir de una clave(get)

```
const mapa = new Map([[1, "uno"], [2, "dos"], [3, "tres"]]);  
mapa.get(1); // devuelve "uno"
```

Maps

Convertir a Arrays: Mediante la desestructuración, podemos convertir los Map en Array o en Objeto.

```
const mapa = new Map([[1, "uno"], [2, "dos"], [3, "tres"]]);  
mapa.size; // 3 (Contiene 3 elementos)  
mapa.constructor.name; // "Map"  
const entries = [...structuredClone(mapa)];  
entries.constructor.name; // "Array" entries; //  
[[1, "uno"], [2, "dos"], [3, "tres"]]
```

Recuerda utilizar `structuredClone()` para clonar la estructura si tiene elementos anidados, ya que sino sólo realizará una **copia superficial** y utilizará referencias para los elementos anidados.

Este array de entradas que nos da como resultado, lo podríamos utilizar para crear un nuevo map, o incluso un objeto:

```
const mapa2 = new Map(entries);  
mapa2; // Map({ 1=>"uno", 2=>"dos", 3=>"tres" })  
SE VERÁ MÁS ADELANTE, CON OBJETOS:  
const object = Object.fromEntries(entries);  
object; // { 1: "uno", 2: "dos", 3: "tres" }
```

WeakMaps

Se trata de una estructura derivada, muy similar a los Map, pero éstos no permiten utilizar tipos primitivos (, ,) como **clave**, mientras que el Map si lo permite:

```
// *** Map
const mapa = new Map([[1, "uno"]]); // OK
const mapa2 = new Map([[{ id: 1, type: "number" }, "uno"]]); // OK
// *** WeakMap
const wmapa = new WeakMap([[1, "uno"]]); // ERROR: Uncaught TypeError:
                                     Invalid value used in weak map key
const wmapa2 = new WeakMap([[{ id: 1, type: "number" }, "uno"]]); // OK
```