

- 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

En <head>

En <body>

Antes de </body>

¿Cómo descarga el archivo Javascript?

ANTES de empezar a dibujar la página.

DURANTE el dibujado de la página.

DESPUÉS de dibujar la página.

Estado de la página

Página aún no dibujada.

Dibujada hasta donde está la etiqueta <script>.

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

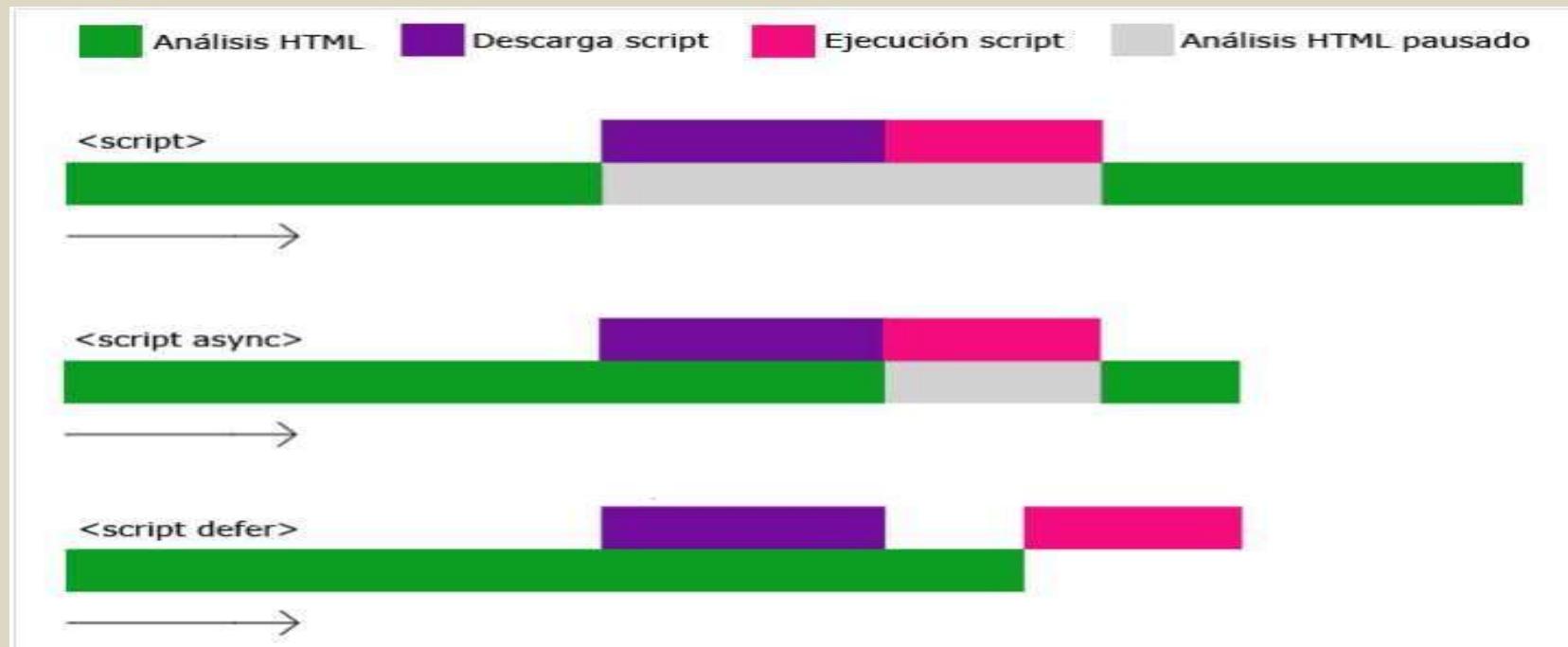
Para evitar este bloqueo podemos colocar los elementos <script> al final del HTML. Cuando el analizador se encuentra con los scripts, casi todo el documento ya se ha analizado y renderizado. Pero era una solución lejos de ser ideal por lo que está en desuso

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 debamos 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 **constants** 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 repesentar 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 Math.log(2)	0.6931471805599453
Math.LN10	Equivalente a Math.log(10)	2.302585092994046
Math.LOG2E	Equivalente a Math.log2(Math.E)	1.4426950408889634
Math.LOG10E	Equivalente a Math.log10(Math.E)	0.4342944819032518
Math.PI	Número PI o Π	3.141592653589793
Math.SQRT1_2	Equivalente a Math.sqrt(1/2).	0.7071067811865476
Math.SQRT2	Equivalente a 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 **interpolar** 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
.split(text)	Separa el texto en varias partes, usando text como separador.
.split(text, limit)	Idem, pero crea como máximo limit fragmentos.
.split(regexp)	Separa el texto usando la regexp como separador.
.split(regexp, limit)	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(regexp)	Busca un patrón que encaje con regexp y devuelve la posición encontrada.
.match(regexp)	Idem a la anterior, pero devuelve las coincidencias encontradas.
.matchAll(regexp)	Idem a la anterior, pero devuelve un iterador para iterar por cada coincidencia.

Tipos de datos: String

Reemplazar cadenas de texto

Método	Descripción
.replace(text, newText)	Reemplaza la primera aparición del text por newText.
.replace(regexp, newText)	Idem, pero busca a partir de una regexp en lugar de un string.
.replaceAll(text, newText)	Reemplaza todas las apariciones del texto text por newText.
.replaceAll(regexp, newText)	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
.toLowerCase()	Devuelve el string transformado a minúsculas.
.toUpperCase()	Devuelve el string transformado a mayúsculas.
.padStart(size, text)	Devuelve el string rellenando el inicio con text hasta llegar al tamaño size.
.padEnd(size, text)	Devuelve el string rellenando el final con text hasta llegar al tamaño size.
.trimStart()	Devuelve el string eliminando espacios a la izquierda del texto.
.trimEnd()	Devuelve el string eliminando espacios a la derecha del texto.
.trim()	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
.concat(text1, text2...)	Devuelve el string unido (concatenado) a las variables text1, text2...

Operadores

Operadores aritméticos

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

Operadores de asignación

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

Operadores condicionales

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

Operadores lógicos

NO (NOT)	!
Y (AND)	<code>&&</code>
O (OR)	<code> </code>

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

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:

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

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 (var 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

```
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

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}`);
```



Entrada y Salida en el navegador

Mensajes de entrada

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



El prompt devuelve null cuando el usuario pulsa Cancelar

Mensajes de alerta

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



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