JavaScript

Introducción

JavaScript® (a menudo abreviado como JS) es un lenguaje de pogramación de alto nivel, interpretado y compilado *('just-in-time')* y orientado a objetos con funciones de primera clase. Es más conocido por su uso para crear páginas web dinámicas e interactivas, pero también se utiliza en muchos entornos que no son de navegador, como por ejemplo *Node.js*.

Es un lenguaje de **tipado dinámico**, lo que significa que las variables pueden cambiar de tipo durante la ejecución del programa. También es **débilmente tipado**, lo que significa que el intérprete hace su mejor esfuerzo para realizar conversiones de tipo automáticamente.

JavaScript está basado en **prototipos**. Es un estilo de programación orientada a objetos en el que las clases no se definen explícitamente, sino que se derivan de agregar propiedades y métodos a una instancia de otra clase o, con menos frecuencia, agregarlos a un objeto vacío.

Por tanto JavaScript es multiparadigma ya que admite estilos de programación orientados a objetos, imperativos y funcionales.

En palabras simples: este tipo de estilo permite la creación de un objeto sin definir primero su clase.

La compilación en tiempo de ejecución (JIT o ***'Just-In-Time'***), también conocida como traducción dinámica, es una técnica para mejorar el rendimiento de sistemas de programación que compilan a bytecode, consistente en traducir el bytecode a código máquina nativo en tiempo de ejecución.

En un sistema que use interpretación de *bytecode* como por ejemplo Smalltalk, Perl, GNU CLISP o las primeras versiones de Java, el código fuente es traducido a un código intermedio llamado *bytecode*. El *bytecode* no es el código máquina de ninguna computadora en particular, y puede por tanto ser portable entre diferentes arquitecturas. El *bytecode* es entonces interpretado, o ejecutado por una máquina virtual.

Un entorno con **compilación dinámica** es aquel en el que el compilador puede ser usado durante la ejecución.

El proyecto Mozilla proporciona dos implementaciones de JavaScript. El primer JavaScript fue creado por Brendan Eich en Netscape, y a partir de entonces se ha actualizado para cumplir con ECMA-262 Edición 5 y versiones posteriores. Este motor, cuyo nombre en código es *SpiderMonkey*, está implementado en C/C++. El motor _Rhino*, creado principalmente por Norris Boyd (también en Netscape) es una implementación de JavaScript escrita en Java. Al igual que SpiderMonkey, Rhino también es compatible con ECMA-262 Edition 5.

Más allá de estas implementaciones, la más conocida es la *V8 de Google, que es utilizada por el navegador Google Chrome y por *Node.js*.

El estándar para JavaScript es **ECMAScript (ECMA-262)** y la especificación de la API para la Internacionalización de ECMAScript (ECMA-402).

La versión que garantiza una compatibilidad de prácticamente el 100% en navegadores desde 2012 es la **ECMAScript 5.1**, mientras que la mayoría de novedades de las versiones superiores obligan a utilizar un navegador más actualizado. Sin embargo es a partir del año 2015 donde se establece un antes y un después para JavaScript.

A partir de ese año 2015 se toma como regla nombrar las diferentes especificaciones por su año en vez de por la versión. Por tanto se recomienda utilizar **ECMAScript 2015** o **ES2015** en vez de ECMAScript 6.

Más información sobre sobre la especificación ECMAScript

Tabla de compatibilidades entre el estándar ECMAScript y las diferentes versiones de navegadores

JavaScript en el navegador

La herramienta **Consola** integrada en Firefox (y en otros navegadores) es útil para experimentar con JavaScript. Nos permite ver errores y advertencias que han surgido al ejecutar el código de página, pertenecientes tanto a JavaScript como CSS o HTML. Puede usarse en dos modos: modo de entrada unilínea y modo de entrada multilínea.

Por otro lado, tenemos el **depurador** de código en el navegador, que nos permite poner *breakpoints* y depurar el código JavaScript.

Integración con HTML

JavaScript puede ser añadido a un documento HTML de dos formas:

- **Scripts en línea**: el código JS se incluye en el HTML, dentro de la propia etiqueta
- **Scripts externos**: el código JS se incluye en un fichero JavaScript externo

Scripts en línea

El código JavaScript puede ser escrito en la cabecera del código (`<head>`) o en el cuerpo (`<body>`). Es preferible ponerlo justo antes del cierre de la etiqueta `<\body>` de manera que el código sea llamado cuando todo el DOM de documento HTML haya sido cargado.

```
<!DOCTYPE html>
<html lang="es">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>JavaScript interno</title>
  </head>
  <body>
    <script type="text/javascript">
      console.log("Hello World!!!");
    </script>
  </body>
</html>
```

Scripts externos

La recomendación es utilizar uno o varios archivos externos para contener el código JavaScript, sobretodo cuando el código comience a tener cierta cantidad de líneas. Los ficheros tendrán la extensión **.js** y por norma general suelen situarse en una carpeta dentro del proyecto.

El enlace del fichero externo o de los ficheros externos puede estar en la cabecera del código (`<head>`) o en el cuerpo (`<body>`). Al igual que antes, es preferible ponerlo justo antes del cierre de la etiqueta `<body>` de manera que el código sea llamado cuando todo el DOM de documento HTML haya sido cargado.

Si la etiqueta `<script>` se coloca en la etiqueta `<head>` la página aún no ha sido "dibujada", por lo que aún no tendremos acceso al DOM. Por lo tanto, si en el código JavaScript necesitamos acceder a ciertas partes de la página HTML, obtendremos referencias nulas a esas etiquetas.

Si la etiqueta `<script>` se coloca en alguna parte de la etiqueta `<body>` , el script se cargará durante el "dibujado" de la página HTML y únicamente tendremos disponible el DOM hasta el punto donde se encuentra la etiqueta `<script>` . Por lo tanto sólo dispondremos de acceso al DOM hasta ese punto.

En cambio, si la etiqueta `<script>` se coloca justo antes de la etiqueta de cierre `</body>`, la página HTML estará "dibujada" al 100% y tendremos acceso a todo el DOM y por tanto a todas las etiquetas.

⚠️ Históricamente, el atributo `type` se utilizaba para indicar el tipo de script que ibamos a utilizar, escribiendo generalmente el valor `text/javascript`. Aún se puede encontrar en páginas antiguas pero en la actualidad se omite. Utilizamos este atributo para cargar Javascript como módulo como por ejemplo `<script type="module" src="js/index.js"></script>`.

```
<!DOCTYPE html>
<html lang="es">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>avaScript en ficheros externos</title>
    <script type="module" src="js/externo1.js"></script>
  </head>
  <body>
    <p id="test2"></p>
    <p id="test3"></p>
    <script src="js/externo2.js"></script>
    <script src="js/externo3.js"></script>
  </body>
</html>
```

Carga de un fichero externo

El atributo `integrity` permite comprobar la integridad de un fichero fichero JavaScript externo y así detectar posibles manipulaciones.

En el atributo se indica el hash proporcionado por el desarrollador de la librería. Al descargar este fichero externo el navegador calcurará el hash y comprobará si el hash calculado y el hash indicado en el atributo coinciden:

```
<head>
  <script src="https://code.jquery.com/jquery-3.7.1.min.js"
    integrity="sha256-/JqT3SQfawRcv/BIHPThkBvs0OEvtFFmqPF/lYI/Cxo="
    crossorigin="anonymous"></script>
</head>
```

- Más información
- Más información sobre SRI (Subresource Integrity)

Modalidad de carga clásica

En el modo **clásico** y que es el modo de carga por defecto de los navegadores, cuando indicamos un script externo mediante el atributo ***'src'***, el proceso de carga del script por parte del navegador es el siguiente:

1. El navegador se encuentra parseando (leyendo) y renderizando (dibujando) el .html en la página.
2. Detiene temporalmente el dibujado en el HTML cuando encuentra un `<script src>`.
3. Descarga el script .js referenciado en el atributo ***'src'*** en el caché del navegador, es decir, le da **prioridad al JS**.
4. Ejecuta el código javascript una vez descargado.
5. Reanuda el proceso de parseo y renderizado del documento HTML por donde lo dejó.

Modalidad de carga en diferido

En la modalidad de carga **diferida**, el navegador le da **prioridad a la carga del documento HTML**. El proceso de carga del script por parte del navegador es el siguiente:

1. El navegador está renderizando el documento HTML y encuentra una etiqueta `<script defer>`.
2. Descarga el script de forma paralela sin detener ni bloquear el renderizado del documento HTML.
3. Continua la renderización del HTML. Si encuentra otro `<script defer>` repite los pasos.
4. Una vez termina de renderizar el documento HTML, ejecuta el script.

Este tipo de carga se realiza incluyendo el atributo ***'defer'*** en la etiqueta `<script>`.

```
<head>
  <script src="js/index.js" defer></script>
</head>
```

En muchas ocasiones las etiquetas `<script>` se colocan (o se aconseja hacerlo) justo antes del cierre de la etiqueta `</body>`. Esto ocurre así porque históricamente, el atributo ***'defer'*** no existía (o existía pero Internet Explorer no lo soportaba) y se necesitaba procesar el JavaScript una vez se hubiese terminado de cargar todo el HTML, para evitar acceder a una parte del documento HTML desde JavaScript y que aún no hubiera cargado.

Modalidad de carga asíncrona

En la modalidad de carga **asíncrona**, el navegador le da prioridad a la ejecución del Javascript. En esta modalidad de carga asíncrona, lo que ocurre es lo siguiente:

1. El navegador está renderizando el documento HTML y encuentra una etiqueta `<script async>`.
2. El navegador descarga el script sin detener ni bloquear la carga del documento HTML.
3. Una vez descargado, interrumpe el renderizado HTML temporalmente y ejecuta el script.
4. Una vez terminada la ejecución del código Javascript, continua con el renderizado HTML.

Este tipo de carga se realiza incluyendo el atributo ***'async'*** en la etiqueta `<script>`.

```
<head>
  <script src="js/index.js" async></script>
</head>
```

Este comportamiento puede interesar para cargar ciertas librerías que interesa que estén disponibles lo antes posible, pero que no van acceder directamente al HTML, por lo que no importa que el documento no esté renderizado por completo.

Navegador sin soporte JavaScript

A día de hoy es improbable que un usuario esté utilizando un navegador sin soporte JavaScript. Es más probable que tenga soporte para JavaScript pero lo tenga deshabilitado. En estos casos podemos utilizar la etiqueta `<noscript>`.

En el caso de que el usuario tenga capacidades de Javascript en su navegador, se ejecutará el código de la etiqueta `<script>` y se ignorará la etiqueta `<noscript>`. Sin embargo, si el navegador no posee Javascript o no lo tiene habilitado, se mostrará el contenido HTML proporcionado en la etiqueta `<noscript>`.

Si no es posible realizar una alternativa al código Javascript, lo ideal sería mostrar al usuario un mensaje donde se le avisa que la página actual sólo es capaz de funcionar con Javascript, y que parece que el navegador que está utilizando no es capaz de procesarlo.

```
<script>
  const usuario = prompt("¿Cuál es tu nombre?");
  alert("¡Hola, " + usuario + "!");
</script>

<noscript>
```

```
  <p>Su navegador no soporta JavaScript o lo tiene deshabilitado</p>
</noscript>
```

Ejecución de JavaScript fuera del navegador

Node.js es un entorno de ejecución para JavaScript del lado del servidor, construido sobre el motor V8 de Google Chrome y creado por Ryan Dahl. A diferencia de JavaScript tradicional, que se ejecuta en el navegador del cliente, Node.js permite ejecutar código JavaScript en el servidor.

Existen otras alternativas como **Deno**, también de Ryan Dahl o **Bun**.

```
# Ejecutar el contenido del fichero 'test.js' con Node.js
$ node test.js
```

Otra opción es invocar una consola interactiva, como en Python, y ejecutar el código directamente:

```
# Invoca la consola
$ node
> console.log("Hello World!!");
Hello World!!
```

Sintaxis del lenguaje

JavaScript está influenciado sobre todo por la sintaxis de Java, C y C++, pero también ha sido influenciado por Awk, Perl y Python.

JavaScript distingue entre mayúsculas y minúsculas (es ***case-sensitive***) y utiliza el conjunto de caracteres Unicode. Por ejemplo, la palabra «Früh» (que significa "temprano" en alemán) se podría usar como el nombre de una variable:

```
let Früh = "foobar";
```

En JavaScript, las instrucciones se denominan **declaraciones** y están separadas por punto y coma (;).

No es necesario un punto y coma después de una declaración si está escrita en su propia línea. Pero si se desea más de una declaración en una línea, entonces debes separarlas con punto y coma. Sin embargo, se considera una buena práctica escribir **siempre** un punto y coma después de una declaración, incluso cuando no sea estrictamente necesario.

Comentarios

La sintaxis de los comentarios es la misma que en C++ y en muchos otros lenguajes:

```
// un comentario de una línea

/* este es un comentario
 * más largo, de varias líneas
 */

/* Sin embargo, no puedes /* anidar comentarios */ SyntaxError */
```

Existe un tercer tipo de sintaxis de comentario al comienzo de algunos archivos JavaScript, que se parece a esto: `#!/usr/bin/env node`. Esto se denomina sintaxis de **comentario hashbang** y es un comentario especial que se utiliza para especificar la ruta a un motor JavaScript en particular que debe ejecutar el script.

Variables

JavaScript es un lenguaje débilmente tipado. Esto quiere decir que no se indica de qué tipo es cada variable que se declara.

Una variable es un espacio de memoria donde se almacena temporalmente un dato para ser utilizado posteriormente en el código. Los nombres de las variables, llamados **identificadores**, se ajustan a ciertas reglas.

Un identificador de JavaScript debe comenzar con una letra, un guión bajo (_) o un signo de dólar (`$`). Los siguientes caracteres, además de lo indicado anterioremente, pueden incluir dígitos (0-9). La única restricción es que no puede **empezar por un número**.

Dado que JavaScript distingue entre mayúsculas y minúsculas, las letras incluyen los caracteres "A" a "Z" (mayúsculas), así como "a" a "z" (minúsculas).

Se puede utilizar la mayoría de las letras ISO 8859-1 o Unicode como å y ü en los identificadores.

Hay que tener en cuenta, al igual que en el resto de lenguajes, que el nombre de las variables no pueden coincidir con el de una palabra reservada de JavaScript.

Declaración de variables

JavaScript tiene tres tipos de declaraciones de variables:

- **var**: Declara una variable tanto local como global, dependiendo del contexto de ejecución. Si se declara dentro de una función tiene ámbito de función. Si se declara fuera de una función tiene ámbito global.
- **let** (a partir de ES2015): Declara una variable local con ámbito de bloque.
- **const** (a partir de ES2015): Declara un nombre de constante de sólo lectura y ámbito de bloque.

Se puede asignar un valor directamente a una variable, como por ejemplo `x = 4`. Esto crea una variable global no declarada. A menudo provocan un comportamiento inesperado por lo que se **desaconseja su uso**.

Una variable declarada usando la instrucción **var** o **let** sin un valor asignado especificado tiene el valor de **undefined**.

```javascript {.numberLines}
var x; // Declaración sin asignación de valor. Tiene el valor de 'undefined'
var y = 10; // Declaración y asignación de valor

console.log( `El valor de c es ${c}` );
// Error de referencia no detectado: c no está definida
// ("Uncaught ReferenceError: c is not defined")

var input;
if (input === undefined) {
// Se puede usar 'undefined' para determinar si una variable tiene un valor.
doThis();
} else {
doThat();
}
```

// Declaración con 'let' dentro de bloque
var x = 5;
{
let x = 10;
console.log(`Declaración con 'let' dentro del bloque: ${x}`); // x = 10
}
console.log(`Declaración con 'var' fuera del bloque: ${x}`); // x = 5

function declaracionVariableEnFuncion() {
// Variable declarada dentro de una función
var y = 20;
}
console.log(y); // ERROR!! ReferenceError: y is not defined

```
#### Ámbito de las variables

Cuando se declara una variable fuera de cualquier función, se denomina **variable global**, porque está disponible para

Cuando se declara una variable dentro de una función tanto con `const` como `var` o `let`, se llama **variable local**,
```

js
// El ámbito de 'x' es el contexto global (o el de una función si este código estuviera
// dentro de una función). Por tanto 'x' no se limita al bloque if.
if (true) {
var x = 5;
}
console.log(x); // x es 5

```
A partir de ECMAScript 6 (2015) se presenta el área de validez a _nivel de bloque_. Tanto `let` como `const` tiene el á
```

js
if (true) {
let y = 5;
}
console.log(y); // ReferenceError: y no está definida

```
##### Elevación de variables (hoisting)

Otra cosa inusual acerca de las variables en JavaScript es que puedes hacer referencia a una variable declarada más tar

Este concepto se conoce como **elevación (hoisting)**. Las variables en JavaScript son, en cierto sentido, "elevadas" (
```

js
/* Ejemplo 1 */
console.log(x === undefined); // true
var x = 3;

/* Ejemplo 2 */
var myVar = "my value";

(function () {
console.log(myVar); // undefined

var myVar = "valor local";
})();

```
Siguiendo con los ejemplos anteriores, lo que hace JavaScript es "elevar" la declaración de las variables a la parte su
```

js
/* Ejemplo 1 */
var x;
console.log(x === undefined); // true
x = 3;

/* Ejemplo 2 */
var myVar = "my value";

(function () {
var myVar;
console.log(myVar); // undefined
myVar = "valor local";
})();

```
En ECMAScript 6 (2015), `let` y `const` se elevan pero **no se inician**. Hacer referencia a la variable en el bloque a
```

js
console.log(x); // ReferenceError
let x = 3;

```
##### Elevación de función

En el caso de las funciones, solo se incluyen declaraciones de función, pero no las expresiones de la función.
```

js
/* Declaración de función */

foo(); // "bar"

function foo() {
console.log("bar");
}

/* Expresión de función */

baz(); // TypeError: baz no es una función

var baz = function () {
console.log("bar2");
};

```
#### Variables globales

Las variables globales, de hecho, son propiedades del objeto global.

En las páginas web, el objeto global es `window`, por lo que se puede establecer y acceder a variables globales utiliza
```

```
En consecuencia, se puede acceder a las variables globales declaradas en una «ventana» o «marco» desde otra «ventana» o

### Constantes

Se puede crear una constante de solo lectura con nombre con la palabra clave `const`.

La sintaxis de un identificador de constante es la misma que la de cualquier identificador de variable: debe comenzar c
```

js
const PI = 3.14;

```
Una constante no puede cambiar el valor a través de la asignación o volver a declararla mientras se ejecuta el script.

Sin embago, las propiedades de los objetos asignados a constantes y el contenido de un array no están protegidos:
```

js
// Modificar la propiedad de un objeto asignado a una constante
const MY_OBJECT = { key: "value" };
MY_OBJECT.key = "otherValue";

// Añadir un nuevo elemento a un array asignado a una constante
const MY_ARRAY = ["HTML", "CSS"];
MY_ARRAY.push("JAVASCRIPT"); // El array ahora contiene "['HTML','CSS','JAVASCRIPT']"

```
Las reglas de ámbito para las constantes son las mismas que las de ámbito de bloque de las variables let. Si se omite l

No se puede declarar una constante con el mismo nombre que una función o una variable en el mismo ámbito

### Estructuras y tipos de datos

- **Tipos primitivos**:
  - _number_ - valor numérico como enteros o decimales
  - _string_ - una secuencia de caracteres que representan un valor de texto
  - _boolean_ - valores de `true` o `false`
  - _null_ - `null` es una palabra clave especial que denota un valor nulo
  - _undefined_ - `undefined` es una propiedad de alto nivel cuyo valor no está definido
  - _symbol_ (ECMAScript 6) - tipo de dato cuyas instancias son únicas e inmutables

- **Objetos**:
  - _Object_ - los objetos son como contenedores con nombre para los valores
  - _Array_ - un tipo especial de objeto para almacenar una secuencia ordenada de valores
  - _Function_ - objetos que contienen código ejecutable.

- **Estructuras de datos adicionales**:
  - _Set_ - colección de valores únicos.
  - _Map_ - colección de pares clave-valor.
  - _WeakSet_ y _WeakMap_ - versiones "débiles" de Set y Map que no impiden la eliminación de elementos por el recolect

- **Tipos especiales**:
  - _Bigint_ (ECMAScript 2020) - representa un número entero con precisión arbitraria
  - _Promise_ - utilizado para operaciones asincrónicas.
  - _Proxy_ - utilizado para la creación de objetos con comportamientos personalizados.
  - _RegExp_ - representa expresiones regulares.

#### Comprobación de tipos

Para comprobar el tipo de datos de un determinado valor, utilizamos el operador `typeof`:
```

js
console.log(typeof "Hello World!!"); // string

```
console.log(typeof 5); // number
console.log(typeof true); // boolean
console.log(typeof null); // object
console.log(typeof undefined); // undefined
```

```
Existe la función `isNaN()` para comprobar si una expresión es numérica o no lo es. Para JavaScript, el valor **NaN (_"
```

```
js
isNaN(NaN); // true
isNaN(1); // false: 1 is a number
isNaN(-2e-4); // false: -2e-4 is a number (-0.0002) in scientific notation
isNaN(Infinity); // false: Infinity is a number
isNaN(true); // false: converted to 1, which is a number
isNaN(false); // false: converted to 0, which is a number
isNaN(null); // false: converted to 0, which is a number
isNaN(""); // false: converted to 0, which is a number
isNaN(" "); // false: converted to 0, which is a number
isNaN("45.3"); // false: string representing a number, converted to 45.3
isNaN("1.2e3"); // false: string representing a number, converted to 1.2e3
isNaN("Infinity"); // false: string representing a number, converted to Infinity
isNaN(new Date); // false: Date object, converted to milliseconds since epoch
isNaN("10$"); // true : conversion fails, the dollar sign is not a digit
isNaN("hello"); // true : conversion fails, no digits at all
isNaN(undefined); // true : converted to NaN
isNaN(); // true : converted to NaN (implicitly undefined)
isNaN(function(){}); // true : conversion fails
isNaN({}); // true : conversion fails
isNaN([1, 2]); // true : converted to "1, 2", which can't be converted to a number
```

```
[Más información en MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/isNaN)

#### Conversión de tipos de datos

JavaScript es un lenguaje **_tipado dinámicamente_**. Esto significa que no hay que especificar el tipo de dato de una
```

```
js
// Esto es perfectamente válido y no genera ningún error
var answer = 42;

answer = "La variable ahora contiene un texto";
```

```
##### Convertir texto a números

En el caso que un valor representando un número está en memoria como texto, tenemos los métodos `parseInt()` y `parseFl

La función `parseInt()` devuelve un entero por lo que desechará la parte decimal si el número en formato texto que quer

Además, la función `parseInt()` permite indicar el sistema numérico a utilizar.
```

```
js
parseInt("3.456"); // Devuelve 3

parseInt("101", 2); // Devuelve 5 ya que 101 es la representación en binario de 5
```

```
#### Números

A diferencia de Java o C/C++, JavaScript utiliza el mismo tipo para todos los números, tanto números enteros como númer
```

```
js
// Número entero
let entero = 1980;
// Número con decimales
let decimales = 0.25;

// Notación científica
const AVOGRADOR = 6.022e+23;
// Número hexadecimal (prefijo 0x)
let hexa = 0xAB12;
// Número octal (prefijo 0o)
let octal = 0o27652;
// Número binario (prefijo 0b)
let bin = 0b100101;
```

```
JavaScript permite la utilización del número infinito (**_'Infinity'_**):
```

```
js
var x = 1/0;
console.log(z); // imprime 'Infinity'

var z1 = Infinity;
console.log(z1); // imprime 'Infinity'
```

```
El valor **undefined** se comporta como **NaN** en contextos numéricos mientras que **null** se comporta como un **0**.

[Más información en MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number)

#### Valores booleanos

Los valores booleanos sólo pueden tomar los valores **true** o **false**.

Sin embargo, en JavaScript todo valor o expresión se comporta como un valor booleano. Mediante la función `Boolean()` p
```

```
js {.numberLines}
// Se comportan como 'true'
console.log(Boolean(true));
console.log(Boolean(1));
console.log(Boolean("cadena no vacía"));
console.log(Boolean(Infinity));

// Se comportan como 'false'
console.log(Boolean(false));
console.log(Boolean(0));
console.log(Boolean(""));
console.log(Boolean(NaN));
console.log(Boolean(undefined));
console.log(Boolean(null));
```

```
Por tanto, sólo los **textos vacíos**, el valor **0**, el valor **false**, el valor **undefined** y **null** se comport

#### Strings

JavaScript permite delimitar los textos tanto con comillas simples como con comillas dobles. Desde la versión ES2016 se

El operador de suma `+` permite concatenar texto y expresiones en JavaScript aunque esta práctica se ya considera obsol
```

js
var x = "La respuesta es " + 42; // "La respuesta es 42"
console.log(typeof x); // string

```
Con todos los demás operadores, JavaScript no convierte valores numéricos en cadenas:
```

js
var y = "37" - 7; // 30
console.log(typeof y); // number

var z = "37" + 7; // "377"
console.log(typeof z); // string

```
A partir de ES2016 se deben utilizar las **_string templates_** utilizando `${}` dentro de una cadena con comillas inve
```

js
let x = 8;
console.log(`La variable 'x' vale ${x}`);

console.log(`Un día tiene ${24*60*60} segundos`);

```

```

js
// Secuencias de escape
let texto ="Una línea\nOtra línea";

```
- `\n` - salto de línea
- `\t` - tabulador
- `\r` - retorno de carro
- `\f` - salto de página
- `\v` - tabulador vertical
- `\"` - comillas dobles
- `\'` - comilla simple
- `\b` - retroceso
- `\\` - barra invertida

Para mostrar caracteres [Unicode](https://home.unicode.org/) se utiliza la secuencia de escape `\u{code}` o usando `Str
```

js
// Imprimir el carácter Unicode usando el formato de escape Unicode
console.log('\u{1F601}'); // Imprime el emoji 😁

```
// Imprimir el carácter Unicode utilizando 'String.fromCodePoint()'
console.log(String.fromCodePoint(0x1F601)); // Imprime el emoji 😁
```

```
En JavaScript los strings disponen de algunas funciones y propiedades como por ejemplo:
```

```
js
// La propiedad 'length' devuelve la longitud de la cadena
console.log("miCadena".length) // imprime '8'

// 'charAt(index)' devuelve el carácter en la posición especificada
console.log("miCadena".charAt(2)) // imprime 'C'

// 'charCodeAt(index)' devuelve el valor Unicode del carácter en la posición especificada
console.log("miCadena".charCodeAt(1)); // Imprime el valor Unicode de "i" = 105

// 'concat(str1, str2, …)' combina dos o más cadenas y devuelve una nueva cadena
console.log("Hello ".concat("World!!")); // Imprime 'Hello World!!"

// 'indexOf(subcadena[, inicio])' devuelve la primera posición de la subcadena en la cadena
// o -1 si no se encuentra
console.log("Hola, mundo!".indexOf("mundo")); // Imprime 7

// 'slice(inicio[, fin])' extrae una porción de la cadena desde inicio hasta fin (sin incluir fin).
console.log("Hola, mundo!".slice(0, 5)); // Imprime "Hola"

// 'toUpperCase()' y 'toLowerCase()'
console.log("Hola, mundo!".toUpperCase()); // Imprime "HOLA, MUNDO!"

// 'trim()' elimina los espacios en blanco al principio y al final de la cadena
console.log(" Hola, mundo! ".trim()); // Imprime "Hola, mundo!"

// 'split(separador)' divide la cadena en un array de subcadenas utilizando el separador proporcionado.
console.log("Hola, mundo!".split(", ")); // Imprime ["Hola", "mundo!"]
```

```
[Más información en MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String)

### Operadores

#### Operadores aritméticos
```

```
js
// Suma '+'
let suma = 2 + 2;

// Resta '-'
let resta = 10 - 2;

// Multiplicación '*'
let multiplicacion = 5 * 4;

// División '/'
let division = 4 / 2;
```

```
// Resto '%'
let resto = 8 % 3;

// Exponente '**'
let exponente = 4 ** 2;
```

```
#### Operadores relacionales o de comparación
```

```
js
// Mayor '>'
console.log(5 > 4); // imprime 'true'

// Menor '<'
console.log(4 < 5); // imprime 'true'

// Mayor o igual '>='
console.log(5 >= 4); // imprime 'true'

// Menor o igual '<='
console.log(4 <= 5); // imprime 'true'

// Igualdad (o igualdad débil) '=='
console.log(5 == "5"); // imprime 'true'

// Identidad (o igualdad fuerte) '==='
console.log(5 === "5"); // imprime 'false'

// Desigualdad débil '!='
console.log(5 != "5"); // imprime 'false'

// Desigualdad estricta '!=='
console.log(5 !== "5"); // imprime 'true'
```

```
#### Operadores lógicos
```

```
js
// Negación '!'
console.log(!true); // imprime 'false'
console.log(!false); // imprime 'true'

// AND '&&'
console.log(true && true); // imprime 'true'
console.log(true && false); // imprime 'false'
console.log(false && true); // imprime 'false'
console.log(false && false); // imprime 'false'

// OR '||'
console.log(true || true); // imprime 'true'
console.log(true || false); // imprime 'true'
console.log(false || true); // imprime 'true'
console.log(false || false); // imprime 'false'
```

```
#### Operadores de asignación

El operador de asignación más importante es `=`, que permite asignar un valor a una variable.
```

```
js
// Asignación '='
let miDato = 5;

// Preincremento '++var'
console.log(++miDato); // Realiza el incremento +1 y luego lee la variable e imprime '6'

// Postincremento 'var++'. Equivale a 'miDato = miDato + 1'
console.log(miDato++); // Lee e imprime el valor '5' y luego realiza el incremento +1

// Predecremento '--var'
console.log(--miDato); // Realiza el decremento -1 y luego lee la variable e imprime '4'

// Postdecremento 'var--'. Equivale a 'miDato = miDato - 1'
console.log(miDato++); // Lee e imprime el valor '5' y luego realiza el decremento -1

// Suma y asignación
x += 5; // Equivale a x = x + 5;

// Resta y asignación
x -= 5; // Equivale a x = x - 5;

// Multiplicación y asignación
x *= 5; // Equivale a x = x * 5;

// División y asignación
x /= 5; // Equivale a x = x / 5;

// Resto y asignación
x %= 5; // Equivale a x = x % 5;

// Exponente y asignación
x **= 5; // Equivale a x = x ** 5;
```

```
## Control de flujo
```

```
js
// Asignación condicinal "condicion ? valor_si_true : valor_si_false"
true ? 5 : 2; // Devuelve 5
false ? 5 : 2; // Devuelve 2
```

```

```

```
js
// Sentencia condicional simple
if (condicion) {
// bloque
}
```

```
// Sentencia condicional compuesta
if (condicion) {
// bloque
} else {
// bloque
}

// Anidación
if (condicion) {
// bloque
} else if (condicion_2) {
// bloque
} else if (condicion_3) {
// bloque
} else if (condicion_4) {
// bloque
} else {
// bloque
}
```

```
js
switch (expresion) {
case valor1:
// código a ejecutar si expresion === valor1
break;
case valor2:
// código a ejecutar si expresion === valor2
break;
// Puedes tener tantos casos como necesites
default:
// código a ejecutar si ninguno de los casos coincide con la expresion
}
```

```
js
// Bucles 'while'
while (condición) {
// Código a ejecutar en cada iteración. No se garantiza la ejecución
}

// Bucles 'do-while'
do {
// Código a ejecutar en cada iteración. Mínimo habrá una iteración
} while (condición);

// Bucles 'for'
for (inicialización; condición; actualización) {
// Código a ejecutar en cada iteración
}
```

```
// Bucle 'for..of' para iterar sobre elementos de objetos iteralbles como arrays
let array = [1, 2, 3, 4, 5];
for (let element of array) {
console.log(element);
}

// Bucle 'forEach' para iterar sobre los arrays
let array = [1, 2, 3, 4, 5];
array.forEach(function(element) {
console.log(element);
});

// Bucle 'for..in' para iterar sobre las propiedades enumerables de un objeto
let objeto = { a: 1, b: 2, c: 3 };
for (let key in objeto) {
console.log(key, objeto[key]);
}
```

```
## Colecciones

### Arrays

Un arreglo o _array_ es una estructura de datos. En JavaScript los arrays son **objetos**.

Los arrays son **dinámicos** y su tamaño se puede modificar después de ser creados.

Además, son **heterogéneos**, lo que significa que pueden almacenar a la vez distintos tipos.
```

```
js
// Crear un array vacío
let myList = [];
let otherList = new Array();

// Inicialización posterior
myList = [1, 2, 3, 4];

// Crear e inicializar un array
let fruits = ["banana", "apple", "orange"];
let mix = new Array('banana', 3, true, ['John', 'Doe'], {'firstName': 'John', 'lastName': 'Smith'});

// Array con elementos indefinidos
let itemsUndefined = ["a", "b", , , "d"];

// Formas de recorrer un array
// Este loop no evita las posiciones indefinidas
for (let i = 0; i < fruits.length; i++) {
console.log( `Fruits: ${fruits[i]}` );
}

// Este loop evita las posiciones indefinidas
fruits.forEach(function(fruit) {
console.log( `Fruits: ${fruit}` );
});

// Este loop evita las posiciones indefinidas
let fruits = ["banana", "apple", , , "orange"];
```

```
for (let index in fruits) {
console.log( `Fruits: ${fruits[index]}` );
}
// Fruits: banana
// Fruits: apple
// Fruits: orange

// Estandar ES2015 'for…of'
// Este loop no evita las posiciones indefinidas
let fruits = ["banana", "apple", , , "orange"];
for (let fruit of fruits) {
console.log( `Fruits: ${fruit}` );
}
// Fruits: banana
// Fruits: apple
// Fruits: undefined
// Fruits: undefined
// Fruits: orange
```

```
js
let dogs = ["Bulldog", "Beagle", "Labrador"];

dogs.toString(); // convert to string: results "Bulldog,Beagle,Labrador"
dogs.join(" * "); // join: "Bulldog * Beagle * Labrador"
dogs.pop(); // remove last element
dogs.push("Chihuahua"); // add new element to the end
dogs[dogs.length] = "Chihuahua"; // the same as push
dogs.shift(); // remove first element
dogs.unshift("Chihuahua"); // add new element to the beginning
delete dogs[0]; // change element to undefined (not recommended)
dogs.splice(2, 0, "Pug", "Boxer"); // add elements (where, how many to remove, element list)
let animals = dogs.concat(cats,birds); // join two arrays (dogs followed by cats and birds)
dogs.slice(1,4); // elements from [1] to [4-1]
dogs.sort(); // sort string alphabetically
dogs.reverse(); // sort string in descending order
x.sort(function(a, b){return a - b}); // numeric sort
x.sort(function(a, b){return b - a}); // numeric descending sort
highest = x[0]; // first item in sorted array is the lowest (or highest) value
x.sort(function(a, b){return 0.5 - Math.random()}); // random order sort

dogs.includes("Beagle"); // ES20015 - 'true' si existe el elemento
dogs.indexOf("Beagle"); // imprime la posición si encuentra el elemento o -1
dogs.lastIndexOf("Beagle") // imprime la posición si encuentra el elemento o -1
```

```
La versión estándar ES2015 incorporó el operador de **propagación o _spread_** `...` aplicable a iterables, como los ar
```

```
js
let [a, b, c] = ["banana", "apple", "orange"];
console.log(a); // imprime 'banana'
console.log(b); // imprime 'apple'
console.log(c); // imprime 'orange'
```

```
let fruits = ["banana", "apple", "orange"];
let [a, b, c] = […fruits]; // 'spread' operator
console.log(a); // imprime 'banana'
console.log(b); // imprime 'apple'
console.log(c); // imprime 'orange'

let [a, b, …array] = ["banana", "apple", "orange", "kiwi", "watermelon"];
console.log(a); // imprime 'banana'
console.log(b); // imprime 'apple'
console.log(array); // imprime [ 'orange', 'kiwi', 'watermelon' ]
```

```
[Más información en MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array)

### Set

Los conjuntos o _sets_ son una estructura de datos y se consideran como **objetos**. Aparecieron en ES2015.

A diferencia de los arrays, los sets no admiten valores duplicados.
```

```
js
// Declarar un conjunto vacío
let conjunto = new Set();
let conjunto2 = new Set(["a", "b", "c"]);

// Añadir elementos
conjunto.add("a");
conjunto.add("b").add("c").add("d");

const mySet1 = new Set();

mySet1.add(1); // Set(1) { 1 }
mySet1.add(5); // Set(2) { 1, 5 }
mySet1.add(5); // Set(2) { 1, 5 }
mySet1.add("some text"); // Set(3) { 1, 5, 'some text' }
const o = { a: 1, b: 2 };
mySet1.add(o);

mySet1.add({ a: 1, b: 2 }); // o is referencing a different object, so this is okay

mySet1.has(1); // true
mySet1.has(3); // false, since 3 has not been added to the set
mySet1.has(5); // true
mySet1.has(Math.sqrt(25)); // true
mySet1.has("Some Text".toLowerCase()); // true
mySet1.has(o); // true

mySet1.size; // 5

mySet1.delete(5); // removes 5 from the set
mySet1.has(5); // false, 5 has been removed

mySet1.size; // 4, since we just removed one value

mySet1.add(5); // Set(5) { 1, 'some text', {…}, {…}, 5 }

console.log(mySet1); // Set(5) { 1, "some text", {…}, {…}, 5 }
```

```
mySet1.clear(); // {}
```

```
js
// ITERAR SOBRE LOS SETS
for (const item of mySet1) {
console.log(item);
}
// 1, "some text", { "a": 1, "b": 2 }, { "a": 1, "b": 2 }, 5

for (const item of mySet1.keys()) {
console.log(item);
}
// 1, "some text", { "a": 1, "b": 2 }, { "a": 1, "b": 2 }, 5

for (const item of mySet1.values()) {
console.log(item);
}
// 1, "some text", { "a": 1, "b": 2 }, { "a": 1, "b": 2 }, 5

// key and value are the same here
for (const [key, value] of mySet1.entries()) {
console.log(key);
}
// 1, "some text", { "a": 1, "b": 2 }, { "a": 1, "b": 2 }, 5

// Convert Set object to an Array object, with Array.from
const myArr = Array.from(mySet1); // [1, "some text", {"a": 1, "b": 2}, {"a": 1, "b": 2}, 5]

// the following will also work if run in an HTML document
mySet1.add(document.body);
mySet1.has(document.querySelector("body")); // true

// converting between Set and Array
const mySet2 = new Set([1, 2, 3, 4]);
console.log(mySet2.size); // 4
console.log([…mySet2]); // [1, 2, 3, 4]

// intersect can be simulated via
const intersection = new Set([…mySet1].filter((x) => mySet2.has(x)));

// difference can be simulated via
const difference = new Set([…mySet1].filter((x) => !mySet2.has(x)));

// Iterate set entries with forEach()
mySet2.forEach((value) => {
console.log(value);
});
// 1
// 2
// 3
// 4
```

```
js
// UTILIZAR UN ARRAY EN UN SET
const myArray = ["value1", "value2", "value3"];

// Use the regular Set constructor to transform an Array into a Set
const mySet = new Set(myArray);

mySet.has("value1"); // returns true

// Use the spread syntax to transform a set into an Array.
console.log([…mySet]); // Will show you exactly the same Array as myArray

// Use to remove duplicate elements from an array
const numbers = [2, 13, 4, 4, 2, 13, 13, 4, 4, 5, 5, 6, 6, 7, 5, 32, 13, 4, 5];

console.log([…new Set(numbers)]); // [2, 13, 4, 5, 6, 7, 32]
```

```
[Más información en MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set)

### Map

Los mapas o _maps_ son estructuras de tipo **clave-valor**, en las cuales las claves no se pueden repetir y tienen asoc

Tanto las claves como los valores pueden ser de cualquier tipo. En un mismo mapa no pueden haber dos elementos con la m
```

```
js
const myMap = new Map();

const keyString = "a string";
const keyObj = {};
const keyFunc = function () {};

// setting the values
myMap.set(keyString, "value associated with 'a string'");
myMap.set(keyObj, "value associated with keyObj");
myMap.set(keyFunc, "value associated with keyFunc");

myMap.set(1, "a").set(2, "b").set(3, "c");

console.log(myMap.size); // 3

console.log(myMap.has(1)) // true
console.log(myMap.has(5)) // false

myMap.delete(2);

// getting the values
console.log(myMap.get(keyString)); // "value associated with 'a string'"
console.log(myMap.get(keyObj)); // "value associated with keyObj"
console.log(myMap.get(keyFunc)); // "value associated with keyFunc"

console.log(myMap.get("a string")); // "value associated with 'a string'", because keyString === 'a string'
console.log(myMap.get({})); // undefined, because keyObj !== {}
console.log(myMap.get(function () {})); // undefined, because keyFunc !== function () {}
```

```
```

```
js
// UTILIZAR UN ARRAY CON UN MAP
const kvArray = [
["key1", "value1"],
["key2", "value2"],
];

// Use the regular Map constructor to transform a 2D key-value Array into a map
const myMap = new Map(kvArray);

console.log(myMap.get("key1")); // "value1"

// Use Array.from() to transform a map into a 2D key-value Array
console.log(Array.from(myMap)); // Will show you exactly the same Array as kvArray

// A succinct way to do the same, using the spread syntax
console.log([…myMap]);

// Or use the keys() or values() iterators, and convert them to an array
console.log(Array.from(myMap.keys())); // ["key1", "key2"]
```

```
```

```
js
// ITERAR SOBRE UN MAP
const myMap = new Map();
myMap.set(0, "zero");
myMap.set(1, "one");

for (const [key, value] of myMap) {
console.log( `${key} = ${value}` );
}
// 0 = zero
// 1 = one

for (const key of myMap.keys()) {
console.log(key);
}
// 0
// 1

for (const value of myMap.values()) {
console.log(value);
}
// zero
// one

for (const [key, value] of myMap.entries()) {
console.log( `${key} = ${value}` );
}
// 0 = zero
// 1 = one
```

myMap.forEach((value, key) => {
console.log(`${key} = ${value}`);
});
// 0 = zero
// 1 = one

```
## Funciones

### Funciones de primera clase

Un lenguaje de programación se dice que tiene **funciones de primera clase** cuando las funciones en ese lenguaje son t

Como identificador/nombre de una función se aplican las mismas reglas que los identificadores de las variables. Debe co

#### Asignar función a una variable

La variable es una **referencia** a la función.
```

js
// Asignamos una función anónima a una variable
const foo = function () {
console.log("foobar");
};

// Invocación de la función usando la variable y añadiendo los paréntesis
foo();

```
#### Pasar la función como argumento
```

js
function diHola() {
return "Hola ";
}
function saludar(saludo, nombre) {
console.log(saludo() + nombre);
}

// Pasamos la función `diHola` como argumento de la función `saludar`
saludar(diHola, "JavaScript!");

```
Una función que se pasa como argumento a otra función se llama _**callback function**_.

#### Devolver una función
```

js
function diHola() {
return function () {
console.log("¡Hola!");
};
}

```
Podemos devolver una función porque JavaScript trata la función como un _value_. Una función que devuelve una función s
```

```
Siguiendo con el ejemplo, para invocar la función devuelta se pueden usar dos formas:

- Usando una variable
```

js
const diHola = function () {
return function () {
console.log("¡Hola!");
};
};
const miFuncion = diHola(); // Asignamos la función devuelta a la variable 'miFuncion'
miFuncion(); // Invocamos la función devuelta

```
- Usando paréntesis dobles
```

javascript
function diHola() {
return function () {
console.log("¡Hola!");
};
}
diHola()(); // Usamos paréntesis dobles para invocar también la función retornada

```
### Funciones flecha

Esta forma de declarar una función sólo sirve con funciones anónimas.
```

js
// Función anónima
function(x) {
return 3*x;
}

// Notación en forma de función flecha
const triple = x => 3*x;

```
Si hay dos o más parámetros, se utilizan los paréntesis:
```

js
// Función anónima
function(x, y) {
return x + y;
}

// Notación en forma de función flecha
const suma = (x, y) => x + y;

```
Si no hay parámetros, hay que colocar paréntesis vacíos:
```

js
// Función anónima
function() {
console.log("Hello World!!);
}

// Notación en forma de función flecha
const saludo = () => {console.log("Hello World!!");}

```
Si no hay `return` en el cuerpo de la función o requiere de varias líneas, es necesario el uso de llaves.

### Argumentos de la función

Los tipos básicos como booleanos, números o strings se pasan **por valor** en los argumentos de una función, es decir,

Los tipos complejos como arrays, conjuntos, mapas o cualquier otro objeto se pasan **por referencia**, por lo que al mo

En JavaScript los parámetos de las funciones pueden tener **valor por defecto**. Eso convierte a dicho parámetro como *
```

js
// Notación en forma de función flecha
const saludo = (test = "World") => {
console.log(`Hello ${test}!!!`);
}

saludo(); // imprime 'Hello World!!!'
saludo("friend"); // imprime "Hello friend!!!

```
Una función puede tener un **número variable** de parámetros. Para acceder a esos parámetros se utiliza el operador de

Si la función tiene parámetros definidos, se colocan primero y luego el operador de propagación:
```

js
function test(x, y, …otros) {
console.log(`${x}, ${y}, ${otros}`);
}

```
### Funciones callback

En JavaScript, los _callbacks_ son funciones que se pasan como argumentos a otras funciones y se ejecutan después de qu

En términos sencillos, una función de _callback_ es una función que se pasa como argumento a otra función y se invoca d
```

js
function operacionAsincrona(callback) {
// Simulación de una operación que toma tiempo
setTimeout(function() {
console.log("Operación completada");
callback(); // Llamada al callback después de completar la operación
}, 2000);
}

function miCallback() {
console.log("Callback ejecutado");

}

// Llamando a la función con el callback
operacionAsincrona(miCallback);

```
Es muy habitual usar funciones _callback_ usando funciones anónimas o funciones flecha:
```

js
function escribe(x, accion) {
console.log(accion(x));
}

function doble(y) {
return y * 2;
}

// Pasándole el nombre de la función definida
escribe(4, doble);

// Pasándole una función anónima
escribe(8, function(y) {
return y * 2;
});

// Pasándole una función flecha
escribe(12, y => y * 2);

```
### Expresión de función ejecutada inmediatamente o 'IIFE'

Las expresiones de función ejecutadas inmediatamente (IIFE por su sigla en inglés) son funciones que se ejecutan tan pr
```

js
(function () {
//statements;
})();

```
Es un patrón de diseño también conocido cómo **función autoejecutable** (_'Self-Executing Anonymous Function'_ en inglé

La segunda parte crea la expresión de función cuya ejecución es inmediata (), siendo interpretado directamente en el en
```

js
(function () {
var aName = "Barry";
})();
// Variable 'name' is not accessible from the outside scope
aName; // throws "Uncaught ReferenceError: aName is not defined"

```

```

js
// Asignar el IIFE a una variable almacena el valor de retorno, no la definición de la función

var result = (function () {
var name = "Barry";
return name;
})();
// Immediately creates the output:
result; // "Barry"

Programación orientada a objetos

Los objetos más sencillos que podemos crear en **JavaScript** son los **llamados** **literales** o ****instancias directas**:**

js
let punto = new Object();
// Propiedades
punto.x = 5;
punto.y = 10;
// Métodos
punto.mostrarCoordenadas = function() {
console.log(`x=${this.x}, y=${this.y}`);
}

punto.mostrarCoordenadas(); // imprime 'x=5, y=10'

Se puede utilizar **la forma** `let punto = new Object();` o **la forma** más directa `let punto = {};`. Las llaves indican **que**

Otra posibilidad es crear el objeto y directamente asignarle propiedades y **m**étodos:

js
let punto = {
x:15,
y:20,
mostrarCoordenadas: function() {
console.log(`x=${this.x}, y=${this.y}`);
}
}

punto.mostrarCoordenadas(); // imprime 'x=15, y=20'

Podemos recorrer las propiedades de un objeto **con el** bucle `**for**...in`:

js
for (let prop in punto) {
console.log(`${prop} - ${punto[prop]}`);
}

Para borrar **una** propiedad **se** puede utilizar **el** operador `**delete**`:

js
delete punto.x;

```
### Constructores

Una forma de crear objetos es usando un **constructor** mediante el operador `new`:
```

js
// Función constructora
function Punto(coordX, coordY) {
this.x = coordX;
this.y = coordY;

this.mostrarCoordenadas = () => {console.log(`x=${this.x}, y=${this.y}`)};
}

// Instanciar un objeto
let punto = new Punto(5, 10);

punto.mostrarCoordenadas(); // // imprime 'x=5, y=10'

```
## Resumen
```

javascript
// Los comentarios en JavaScript son los mismos como comentarios en C.

//Los comentarios de una sola línea comienzan con //,
/* y los comentarios multilínea comienzan
y terminan con */

// Cada sentencia puede ser terminada con punto y coma ;
hazAlgo();

// … aunque no es necesario, ya que el punto y coma se agrega automáticamente
// cada que se detecta una nueva línea, a excepción de algunos casos.
hazAlgo()

// Dado que esta práctica puede llevar a resultados inesperados, seguiremos agregando
// punto y coma en esta guía.

///////////////////////////////////
// 1. Números, Strings y Operadores

// JavaScript tiene un solo tipo de número (doble de 64-bit IEEE 754).
// Así como con Lua, no te espantes por la falta de enteros: los dobles tienen 52 bits
// de mantisa, lo cual es suficiente para guardar enteros de hasta 9×10^{15}.
3; // = 3
1.5; // = 1.5

// Toda la aritmética básica funciona como uno esperaría.
1 + 1; // = 2
0.1 + 0.2; // = 0.30000000000000004
8 - 1; // = 7
10 * 2; // = 20
35 / 5; // = 7
```
```

```
// Incluyendo divisiones con resultados no enteros.
5 / 2; // = 2.5

// Las operaciones con bits también funcionan; cuando ejecutas una operación con bits
// el número flotante se convierte a entero con signo hasta 32 bits.
1 << 2; // = 4

// La jerarquía de las operaciones se aplica con paréntesis.
(1 + 3) * 2; // = 8

// Hay tres casos especiales de valores con los números:
Infinity; // por ejemplo: 1/0
-Infinity; // por ejemplo: -1/0
NaN; // por ejemplo: 0/0

// También hay booleanos:
true;
false;

// Los Strings se pueden crear con ' o ".
'abc';
"Hola, mundo";

// La negación se aplica con la expresión !
!true; // = false
!false; // = true

// Para comprobar una igualdad se usa ===
1 === 1; // = true
2 === 1; // = false

// Para comprobar una desigualdad se usa !==
1 !== 1; // = false
2 !== 1; // = true

// Más comparaciones
1 < 10; // = true 1 > 10; // = false
2 <= 2; // = true 2 >= 2; // = true

// Los Strings se concatenan con +
"¡Hola " + "mundo!"; // = "¡Hola mundo!"

// y se comparan con < y con >
"a" < "b"; // = true

// Los tipos no importan con el operador ==…
"5" == 5; // = true
null == undefined; // = true

// …a menos que uses ===
"5" === 5; // = false
null === undefined; // false

// Los Strings funcionan como arreglos de caracteres
// Puedes acceder a cada caracter con la función charAt()
"Este es un String".charAt(0); // = 'E'
```

```
// …o puedes usar la función substring() para acceder a pedazos más grandes
"Hola Mundo".substring(0, 4); // = "Hola"

// length es una propiedad, así que no uses ()
"Hola".length; // = 4

// También hay null y undefined
null; // usado para indicar una falta de valor deliberada
undefined; // usado para indicar que un valor no está presente actualmente
// (aunque undefined es un valor en sí mismo)

// false, null, undefined, NaN, 0 y "" es false; todo lo demás es true.
// Note que 0 es false y "0" es true, a pesar de que 0 == "0".
// Aunque 0 === "0" sí es false.

///////////////////////////////////
// 2. Variables, Arrays y Objetos

// Las variables se declaran con la palabra var. JavaScript cuenta con tipado dinámico,
// así que no se necesitan aplicar tipos. La asignación se logra con el operador =.
var miPrimeraVariable = 5;

// si no escribes la palabra var no se marcará ningún error…
miSegundaVariable = 10;

// …pero tu variable se declarará en el ámbito global, no en el ámbito
// en el que se definió.

// Las variables que no están aún asignadas tienen el valor undefined.
var miTerceraVariable; // = undefined

// Existen atajos para realizar operaciones aritméticas:
miPrimeraVariable += 5; // equivalente a miPrimeraVariable = miPrimeraVariable + 5;
// miPrimeraVariable ahora es 10
miPrimeraVariable *= 10; // ahora miPrimeraVariable es 100

// Y atajos aún más cortos para sumar y restar 1
miPrimeraVariable++; // ahora miPrimeraVariable es 101
miPrimeraVariable--; // de vuelta a 100

// Los arreglos son listas ordenadas de valores, de cualquier tipo.
var miArreglo = ["Hola", 45, true];

// Los miembros de un arreglo pueden ser accesados con la sintaxis
// de indices dentro de corchetes [].
// Los índices empiezan en cero.
miArreglo[1]; // = 45

// Los arreglos son mutables y pueden cambiar de longitud.
miArreglo.push("Mundo");
miArreglo.length; // = 4

// Agregar/Modificar en un determinado índice
miArreglo[3] = "Hola";
```

```
// Los objetos en JavaScript son equivalentes a los 'diccionarios' o 'mapas' en otros
// lenguajes: una colección de pares llave/valor desordenada.
var miObjeto = {llave1: "Hola", llave2: "Mundo"};

// Las llaves son strings, pero no se necesitan las comillas si son un identificador
// válido de JavaScript. Los valores pueden ser de cualquier tipo.
var miObjeto = {miLlave: "miValor", "mi otra llave": 4};

// Los atributos de los objetos también pueden ser accesadas usando
// la sintaxis de corchetes,
miObjeto["mi otra llave"]; // = 4

// … o usando la sintaxis de punto, dado que la llave es un identificador válido.
miObjeto.miLlave; // = "miValor"

// Los objetos son mutables; los valores pueden ser cambiados y se pueden
// agregar nuevas llaves.
miObjeto.miTerceraLlave = true;

// Si intentas acceder con una llave que aún no está asignada tendrás undefined.
miObjeto.miCuartaLlave; // = undefined

////////////////////////////////////
// 3. Lógica y estructura de control

// La sintaxis de esta sección es casi idéntica a la de Java.

// La estructura if funciona de la misma forma.
var contador = 1;
if (contador == 3){
// evaluar si contador es igual a 3
} else if (contador == 4){
// evaluar si contador es igual a 4
} else {
// evaluar si contador no es igual a 3 ni a 4
}

// De la misma forma la estructura while.
while (true){
// ¡Loop infinito!
}

// La estructura Do-while es igual al while, excepto que se ejecuta al menos una vez.
var input
do {
input = conseguirInput();
} while (!esValido(input))

// la esctructura for es la misma que la de C y Java:
// inicialización; condición; iteración.
for (var i = 0; i < 5; i++){
// correrá cinco veces
}

// && es un "y" lógico, || es un "o" lógico
if (casa.tamano == "grande" && casa.color == "azul"){
```

```
casa.contiene = "oso";
}
if (color == "rojo" || color == "azul"){
// el color es rojo o azul
}

// && y || "corto circuito", lo cual es útil para establecer valores por defecto.
var nombre = otroNombre || "default";

// la estructura switch usa === para sus comparaciones
// usa 'break' para terminar cada caso
// o los casos después del caso correcto serán ejecutados también.
calificacion = 'B';
switch (calificacion) {
case 'A':
console.log("Excelente trabajo");
break;
case 'B':
console.log("Buen trabajo");
break;
case 'C':
console.log("Puedes hacerlo mejor");
break;
default:
console.log("Muy mal");
break;
}

////////////////////////////////////
// 4. Funciones, ámbitos y closures

// Las funciones en JavaScript son declaradas con la palabra clave "function".
function miFuncion(miArgumentoString){
return miArgumentoString.toUpperCase(); //la funcion toUpperCase() vuelve todo
// el String a mayúsculas
}
miFuncion("foo"); // = "FOO"

// Note que el valor a ser regresado debe estar en la misma línea que la
// palabra clave 'return', de otra forma la función siempre regresará 'undefined'
// debido a la inserción automática de punto y coma.
function miFuncion()
{
return // <- punto y coma insertado aquí automáticamente
{
estaEsUna: 'propiedad'
}
}
miFuncion(); // = undefined al mandar a llamar la función

// Las funciones en JavaScript son de primera clase, así que pueden ser asignadas
// a variables y pasadas a otras funciones como argumentos - por ejemplo:
function miFuncion(){
// este código será llamado cada cinco segundos
}
```

```
setTimeout(miFuncion, 5000);
// Note: setTimeout no es parte de JS, pero lo puedes obtener de los browsers
// y Node.js.

// Es posible declarar funciones sin nombre - se llaman funciones anónimas
// y se definen como argumentos de otras funciones.
setTimeout(function(){
// este código se ejecuta cada cinco segundos
}, 5000);

// JavaScript tiene ámbitos de funciones; las funciones tienen su propio ámbito pero
// otros bloques no.
if (true){
var i = 5;
}
i; // = 5 - en un lenguaje que da ámbitos por bloque esto sería undefined, pero no aquí.

// Este conlleva a un patrón de diseño común llamado "ejecutar funciones anónimas
//inmediatamente", que preveé variables temporales de fugarse al ámbito global
(function(){
var temporal = 5;
// Podemos acceder al ámbito global asignando al 'objeto global', el cual
// en un navegador siempre es 'window'. El objeto global puede tener
// un nombre diferente en ambientes distintos, por ejemplo Node.js .
window.permanente = 10;
})();
temporal; // da ReferenceError
permanente; // = 10

// Una de las características más útiles de JavaScript son los closures.
// Si una función es definida dentro de otra función, la función interna tiene acceso
// a todas las variables de la función externa, incluso aunque la función
// externa ya haya terminado.
function decirHolaCadaCincoSegundos(nombre){
var texto = "¡Hola, " + nombre + "!";
// Las funciones internas son puestas en el ámbito local por defecto
// como si fueran declaradas con 'var'.
function interna(){
alert(texto);
}
setTimeout(interna, 5000);
// setTimeout es asíncrono, así que la función decirHolaCadaCincoSegundos
// terminará inmediatamente, y setTimeout llamará a interna() a los cinco segundos
// Como interna está "cerrada dentro de" decirHolaCadaCindoSegundos, interna todavía tiene
// acceso a la variable 'texto' cuando es llamada.
}
decirHolaCadaCincoSegundos("Adam"); // mostrará una alerta con "¡Hola, Adam!" en 5s

////////////////////////////////////
// 5. Más sobre objetos; constructores y prototipos

// Los objetos pueden contener funciones.
var miObjeto = {
miFuncion: function(){
return "¡Hola Mundo!";
```

```
}
};
miObjeto.miFuncion(); // = "¡Hola Mundo!"

// Cuando las funciones de un objeto son llamadas, pueden acceder a las variables
// del objeto con la palabra clave 'this'.
miObjeto = {
miString: "¡Hola Mundo!",
miFuncion: function(){
return this.miString;
}
};
miObjeto.miFuncion(); // = "¡Hola Mundo!"

// Las funciones de un objeto deben ser llamadas dentro del contexto de ese objeto.
var miFuncion = myObj.miFuncion;
miFuncion(); // = undefined

// Una función puede ser asignada al objeto y ganar acceso a él gracias a esto,
// incluso si no estaba dentro del objeto cuando este se definió.
var miOtraFuncion = function(){
return this.miString.toUpperCase();
}
miObjeto.miOtraFuncion = myOtherFunc;
miObjeto.miOtraFuncion(); // = "¡HOLA MUNDO!"

// Podemos especificar el contexto en el que una función será llamada con los comandos
// 'call' o 'apply'.

var otraFuncion = function(otroString){
return this.miString + otroString;
}
otraFuncion.call(miObjeto, " y hola Luna!"); // = "¡Hola Mundo! y hola Luna!"

// 'apply' es casi idéntico, pero recibe un arreglo como argumento.

otraFuncion.apply(miObjeto, [" y hola Sol!"]); // = "¡Hola Mundo! y hola Sol!"

// Esto es útil cuando estás trabajando con una función que acepta una secuencia de
// argumentos y quieres pasar un arreglo.

Math.min(42, 6, 27); // = 6
Math.min([42, 6, 27]); // = NaN (uh-oh!)
Math.min.apply(Math, [42, 6, 27]); // = 6

// Pero 'call' y 'apply' sólo son temporales. Cuando queremos que se quede, usamos bind.

var funcionUnida = otraFuncion.bind(miObjeto);
funcionUnida(" y hola Saturno!"); // = "¡Hola Mundo! y hola Saturno!"

// Bind también puede ser usada para aplicar parcialmente (curry) una función.

var producto = function(a, b){ return a * b; }
var porDos = producto.bind(this, 2);
porDos(8); // = 16
```

```
// Cuando llamas a una función con la palabra clave 'new' un nuevo objeto es creado.
// Se hace disponible a la función. Las funciones diseñadas para ser usadas así se
// llaman constructores.

var MiConstructor = function(){
this.miNumero = 5;
}
miNuevoObjeto = new MiConstructor(); // = {miNumero: 5}
miNuevoObjeto.miNumero; // = 5

// Todos los objetos JavaScript tienen un 'prototipo'. Cuando vas a acceder a una
// propiedad en un objeto que no existe en el objeto el intérprete buscará en
// el prototipo.

// Algunas implementaciones de JavaScript te permiten acceder al prototipo de
// un objeto con la propiedad **proto**. Mientras que esto es útil para explicar
// prototipos, no es parte del estándar; veremos formas estándar de usar prototipos
// más adelante.

var miObjeto = {
miString: "¡Hola Mundo!"
};
var miPrototipo = {
sentidoDeLaVida: 42,
miFuncion: function(){
return this.miString.toLowerCase()
}
};

miObjeto.**proto** = miPrototipo;
miObjeto.sentidoDeLaVida; // = 42

// Esto funcionan también para funciones.
miObjeto.miFuncion(); // = "hello world!"

// Por supuesto, si la propiedad que buscas no está en el prototipo,
// se buscará en el prototipo del prototipo.
miPrototipo.**proto** = {
miBoolean: true
};
miObjeto.miBoolean; // = true

// Esto no involucra ningún copiado, cada objeto guarda una referencia a su
// prototipo. Esto significa que podemos alterar el prototipo y nuestros
// cambios serán reflejados en todos lados.
miPrototipo.sentidoDeLaVida = 43;
miObjeto.sentidoDeLaVida; // = 43

// Mencionabamos anteriormente que **proto** no está estandarizado, y que no
// existe una forma estándar de acceder al prototipo de un objeto. De todas formas.
// hay dos formas de crear un nuevo objeto con un prototipo dado.

// El primer método es Object.create, el cual es una adición reciente a JavaScript,
// y por lo tanto, no disponible para todas las implementaciones aún.
var miObjeto = Object.create(miPrototipo);
miObjeto.sentidoDeLaVida; // = 43
```

```
// El segundo método, el cual trabaja en todos lados, tiene que ver con los
// constructores. Los constructores tienen una propiedad llamada prototype.
// Este NO ES el prototipo de la función constructor; es el prototipo que
// se le da a los nuevos objetos cuando son creados con la palabra clave
// new.

MiConstructor.prototype = {
miNumero: 5,
getMiNumero: function(){
return this.miNumero;
}
};
var miNuevoObjeto2 = new MiConstructor();
miNuevoObjeto2.getMiNumero(); // = 5
miNuevoObjeto2.miNumero = 6
miNuevoObjeto2.getMiNumero(); // = 6

// Los tipos que vienen por defecto en JavaScript (como Strings y números)
// también tienen constructores que crean objetos equivalentes.
var miNumero = 12;
var miNumeroObjeto = new Number(12);
miNumero == miNumeroObjeto; // = true

// No son exactamente iguales.
typeof miNumero; // = 'number'
typeof miNumeroObjeto; // = 'object'
miNumero === miNumeroObjeyo; // = false
if (0){
// Este código no se ejecutará porque 0 es false.
}

// Aún así, los objetos que envuelven y los prototipos por defecto comparten
// un prototipo. así que puedes agregar funcionalidades a un string de la
// siguiente forma:
String.prototype.primerCaracter = function(){
return this.charAt(0);
}
"abc".primerCaracter(); // = "a"

// Este hecho se usa normalmente en "polyfilling", lo cual es implementar
// nuevas funciones a JavaScript en un JavaScript más viejo, así que pueda ser
// compatible con ambintes más viejos de JavaScript (por ejemplo, navegadores viejos).

// Por ejemplo, mencionabamos que Object.create no está aún disponible en todas
// las implementaciones, pero podemos hacerlo con polyfill:
if (Object.create === undefined){ // esta validación sirve para no sobreescribir
Object.create = function(proto){
// hace un constructor temporal con el prototipo correcto
var Constructor = function(){};
Constructor.prototype = proto;
// y luego lo usamos para hacer un objeto con el prototipo
// correcto.
return new Constructor();
}
```

}
```

## Enlaces de interés

- https://github.com/sorrycc/awesome-javascript
- https://github.com/ryanmcdermott/clean-code-javascript
- https://lenguajejs.com
- https://developer.mozilla.org/es/docs/Web/JavaScript
- https://developer.mozilla.org/es/docs/Web/JavaScript/Language_overview
- https://github.com/CodeKommissar/JavaScript-Elocuente/blob/master/00_intro.md
- https://overapi.com/javascript
- https://htmlcheatsheet.com/js/
- https://ilovecoding.org/blog/js-cheatsheet
- https://cheatsheets.shecodes.io/javascript
- https://roadmap.sh/javascript
- https://learnxinyminutes.com/docs/es-es/javascript-es/
- https://javascript.info/
- https://www.javascripttutorial.net/
- https://exploringjs.com/es6/
- https://www.theodinproject.com/paths/full-stack-javascript/courses/javascript
- https://jsfiddle.net/

## Licencia
```