JavaScript

Introducción

JavaScript® (a menudo abreviado como JS) es un lenguaje ligero, interpretado o 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 scripts que es dinámico, multiparadigma, basado en prototipos y admite estilos de programación orientados a objetos, imperativos y funcionales.

La **programación basada 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.

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

Modalidad 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 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 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 ***'defer'*** 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>
```

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

Una variable es un espacio de memoria donde almacenamos temporalmente un dato para ser utilizado posteriormente en nuestro 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 también pueden ser 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. Opcionalmente se le puede asignar un valor. Esta sintaxis se puede utilizar para declarar variables locales y globales, dependiendo del contexto de ejecución.
- **let** Declara una variable local con ámbito de bloque, opcionalmente la inicia a un valor.
- **const** 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();
}

```
El valor **undefined** y **null** se comportan como **false** en contextos booleanos.

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

js {.numberLines}
var a;
a + 2; // Evalúa a NaN

var input; // undefined
if (input) {
// Se comporta como 'false' en contextos booleanos.
doThis();
} else {
doThat(); // Se ejecutará este código
}

```
#### Ámbito de las variables

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

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 de datos

El último estándar ECMAScript define ocho tipos de datos:

- Tipos de **datos primitivos**:
  - Number -> valor numérico como enteros o decimales
  - Bigint -> Un número entero con precisión arbitraria
  - Booleano -> `true` o `false`
  - Nulo -> `null` es una palabra clave especial que denota un valor nulo
  - undefined -> `undefined` es una propiedad de alto nivel cuyo valor no está definido
  - String -> Una secuencia de caracteres que representan un valor de texto
  - Symbol -> (nuevo en ECMAScript 6) Un tipo de dato cuyas instancias son únicas e inmutables
- Tipos de **datos no primitivos**:
  - Object -> Los objetos son como contenedores con nombre para los valores

#### 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
```

```
#### Conversión de tipos de datos

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

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

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

```
#### Números y el operador de suma

En expresiones que involucran valores numéricos y de cadena con el operador `+`, JavaScript convierte los valores numér
```

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
```

```
Utilizar el operador de suma para concatenar cadenas se considera una práctica obsoleta. En su lugar es mejor utilizar

#### 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
```

```
## 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

#### Asignar función a una variable
```

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

```
### 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"
```

---

# Enlaces de interés

- https://lenguajejs.com
- https://developer.mozilla.org/es/docs/Web/JavaScript
- 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://jsfiddle.net/

Licencia