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 se iba a utilizar, escribiendo generalmente el valor `text/javascript`. Aún se puede encontrar en páginas antiguas pero en la actualidad se omite. Se utiliza 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 (minúscula o mayúscula), 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 u otros símbolos**.

Dado que JavaScript distingue entre mayúsculas y minúsculas (es ***"case-sensitive"***), 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 o ES6): Declara una variable local con ámbito de bloque.
- **const** (a partir de ES2015 o ES6): 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**.

```
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
```

Mientras que el uso de **var** permite la redefinición o sobreescritura de variables, las variables definidas con **let** no se permite que puedan ser definida de nuevo dentro del mismo contexto o bloque:

```
let miVariable = 10; // Fuera del bloque 'if'

if (true) {
    let miVariable = 20; // Dentro del bloque 'if'
    let miVariable = 30; // SyntaxError: Identifier 'miVariable' has already been declared
    console.log(miVariable); // Nunca se ejecutará debido al error anterior
}
```

Ámbito de las variables

Cuando se declara una variable **fuera** de cualquier función, se denomina **variable global**, porque está disponible para cualquier otro código en el documento actual.

Cuando se declara una variable **dentro** de una función con `const`, `var` o `let`, se llama **variable local**, porque solo está disponible dentro de esa función.

```
// 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 área de validez a nivel de bloque. Un bloque está delimitado por llaves (`{}`).

```
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 tarde, sin obtener una excepción.

Este concepto se conoce como **elevación (hoisting)**. Las variables en JavaScript son, en cierto sentido, "elevadas" (o "izadas") a la parte superior de la función o declaración. Sin embargo, las variables que se elevan devuelven un valor de *undefined*. Entonces, incluso si la declaras e inicias después de usarla o hacer referencia a esta variable, todavía devuelve undefined.

```
/* 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 superior de la función. Como hemos visto anteriormente, si no se asigna valor a una variable tiene el valor de *undefined*. Lo que hace JavaScript sería algo así:

```
/* 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 antes de la declaración de la variable da como resultado un `ReferenceError`:

```
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.

```
/* 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 utilizando la sintaxis `window.variable`.

En consecuencia, se puede acceder a las variables globales declaradas en una «ventana» o «marco» desde otra «ventana» o «marco» especificando el nombre de la window o el frame. Por ejemplo, si se declara una variable llamada `phoneNumber` en un documento, se puede hacer referencia a esta variable desde un iframe como `parent.phoneNumber`.

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 con una letra, un subrayado o un signo de dólar ($) y puede contener caracteres alfabéticos, numéricos o de subrayado.

```
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. Se debe iniciar a un valor.

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

```
// 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 la palabra clave `const`, se asume que el identificador representa una variable.

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

Tipos de datos

- **Tipos primitivos**:
  - *number* - valor númerico 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 recolector de basura.

- **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.

Números

A diferencia de Java o C/C++, JavaScript utiliza el mismo tipo para todos los números, tanto números enteros como números con decimales. Por tanto en JavaScript todos los números ocupan en memoria **64bits**.

```
// 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'***):

```
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**.

Los tipos numéricos disponen de la función `toLocaleString()` para formatear números en **decimal**, **moneda** o **tanto por ciento**. Según el tipo, acepta un objeto con las opciones para presentar el número:

```
(0.54).toLocaleString("es-ES", {style: "percent"}); // '54 %'


(0.21).toLocaleString("es-ES", {style: "percent", minimumFractionDigits: 2}); // '21,00 %'

var options = {
  style: "currency",
  currency: "EUR",
  currencyDisplay: "name"
}
(4.1).toLocaleString("es-ES", options); // '4,10 euros'

var options = {
  style: "currency",
  currency: "EUR",
  currencyDisplay: "symbol"
}
(4.1).toLocaleString("es-ES", options); // '4,10 €'
```

Para controlar el número de dígitos en la parte entera o decimal se utiliza en las opciones `minimumIntegerDigits`, `minimumFractionDigits`, `maximumFractionDigits`.

- Más información en MDN

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()` podemos imprimir el valor booleano de cualquier otro valor:

```
// 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 comportan como **false** en contextos booleanos.

Strings

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

```
let cadena1 = "Esto es una cadena";
let cadena2 = 'Esto es una cadena';
let cadena3= `Esto es una cadena`;
```

El operador de suma `+` permite concatenar texto y expresiones en JavaScript aunque esta práctica se considera **obsoleta**:

```
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:

```
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 invertidas:

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

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

```
// 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 se utiliza la secuencia de escape `\u{code}` o usando `String.fromCodePoint()`:

```
// 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 las cadenas se comparan con el operador `==`, a diferencia de Java:

```
if ("home" == "home") {
  console.log("Equals");
}
```

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

```
// 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

Comprobación y conversión de tipos

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

```
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 (*"Not a Number"*)** es un valor válido.

```
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

JavaScript es un lenguaje ***tipado dinámicamente***. Esto significa que no hay que especificar el tipo de dato de una variable cuando se declara. También significa que los tipos de datos se convierten automáticamente según sea necesario durante la ejecución del script.

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

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

JavaScript dispone de las funciones `parseInt()` y `parseFloat()` integradas en el lenguaje para realizar la conversión de tipos de **'string' a 'number'**.

```
let cadena = "123";
let numero = parseInt(cadena);

console.log(typeof(cadena)); // imprime 'string'
console.log(typeof(numero)); // imprime 'number'
```

La función `parseInt()` devuelve un entero por lo que desechará la parte decimal si el número en formato texto que queremos convertir tiene decimales.

Además, la función `parseInt()` permite indicar el sistema numérico a utilizar. Por defecto se asume en **base decimal**. Si el valor comienza por `0x` se asume que está en **base hexadecimal**.

Si se intenta hacer la conversión de una cadena que no es un número devuelve `NaN`.

```
parseInt("3.456"); // Devuelve 3
parseInt("101", 2); // Devuelve 5 ya que 101 es la representación en binario de 5

console.log(parseInt("42")); // Salida: 42
console.log(parseInt("-123")); // Salida: -123
console.log(parseInt("3.14")); // Salida: 3
console.log(parseInt("abc123")); // Salida: NaN (Not a Number)
console.log(parseInt("123abc")); // Salida: 123
console.log( parseInt("   456")); // Salida: 456
console.log(parseInt("0123", 8)); // Salida: 83 (en octal)
console.log(parseInt("0123", 10)); // Salida: 123 (en decimal)
console.log(parseInt("0123")); // Salida: 123 (en decimal por defecto)
console.log(parseInt("0x11")); // Salida: 17 (en hexadecimal)
```

Operadores

Operadores aritméticos

```
// 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

```
// 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

```
// 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.

```
// 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

Condicinal

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

```
// 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
}
```

```
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
}
```

Bucles

```
// 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 iterables 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.

```
// 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
```

```
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 arrays, un string o un objeto.

```
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

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.

```
// 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(); // {}
```

```
// 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
```

```
// 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

Map

Los mapas o *maps* son estructuras de tipo **clave-valor**, en las cuales las claves no se pueden repetir y tienen asociado un valor. Se consideran objetos y forman parte del estándar ES2015.

Tanto las claves como los valores pueden ser de cualquier tipo. En un mismo mapa no pueden haber dos elementos con la misma clave, pero sí pueden repetir valor.

```
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 () {}
```

```
// 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"]
```

```
// 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 tratadas como cualquier otra variable. Por ejemplo, en ese lenguaje, una función puede ser pasada como argumento a otras funciones, puede ser retornada por otra función y puede ser asignada a una variable.

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

Asignar función a una variable

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

```
// 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

```
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

```
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 se llama ***Higher-Order Function***.

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

- Usando una variable

```
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

```
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 y consiste en que no aparece la palabra `function` y que una flecha `=>` separa los parámetros del cuerpo. Esta forma de escribir funciones aparece en JavaScript a partir de ECMAScript 6.

Las funciones flechas tienen algunas ventajas a la hora de simplificar código bastante interesantes:

- Si el cuerpo de la función sólo tiene una línea, podemos omitir las llaves (`{}`).
- Además, en ese caso, automáticamente se hace un `return` de esa única línea, por lo que se puede omitir.
- En el caso de que la función no tenga parámetros, se indica como: `() =>` .
- En el caso de que la función tenga un solo parámetro, se puede indicar simplemente el nombre del mismo: `e =>` .
- En el caso de que la función tenga 2 ó más parámetros, se indican entre paréntesis: `(a, b) =>` .
- Si queremos devolver un objeto, que coincide con la sintaxis de las llaves, se puede englobar con paréntesis: (`{capture: true}`).

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

// Equivalente con notación en forma de función flecha
const triple = x => 3*x;
```

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

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

```
// Equivalente con notación en forma de función flecha
const suma = (x, y) => x + y;
```

Si no hay parámetros, hay que colocar paréntesis vacíos:

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

// Equivalente con 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, se envía una copia de los mismos.

Los tipos complejos como arrays, conjuntos, mapas o cualquier otro objeto se pasan **por referencia**, por lo que al modificar un objeto dentro de una función se modifica el original.

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

```
// 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 propagación o *spread*.

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

```
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 que cierta operación ha sido completada. Los *callbacks* son comúnmente utilizados en situaciones asíncronas, como en llamadas a API, manejo de eventos, o tareas que toman tiempo.

En términos sencillos, una función de *callback* es una función que se pasa como argumento a otra función y se invoca después de que la operación principal ha sido completada.

```
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:

```
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 pronto como se definen:

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

Es un patrón de diseño también conocido cómo **función autoejecutable** (*'Self-Executing Anonymous Function'* en inglés) y se compone por dos partes. La primera es la función anónima con alcance léxico encerrado por el *Operador de Agrupación ()*. Esto impide acceder a las variables fuera del IIFE, así cómo contaminar el alcance (scope) global.

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

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

```
// 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**:

```
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 es un objeto vacío.

Otra posibilidad es crear el objeto y directamente asignarle propiedades y métodos:

```
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`:

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

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

```
delete punto.x;
```

Constructores

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

```
// 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 con el operador 'new'
let punto = new Punto(5, 10);

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

Con el operador `instanceof` podemos saber el tipo de un objeto. Al igual que Java, los objetos heredan de la clase `Object`:

```
console.log(punto instanceof Punto); // imprime 'true'
console.log(punto instanceof Object); // imprime 'true'
```

Otra posibilidad introducida en el estándar ES2015 para comprobar el tipo de un objeto es una propiedad que tienen todos los objetos a la que se accede por `constructor.name`:

```
console.log(punto.constructor.name);
```

Prototipos

En JavaScript, los prototipos son una parte fundamental del sistema de **herencia** en el lenguaje.

Cada objeto en JavaScript tiene un prototipo, que es esencialmente un objeto del cual hereda propiedades y métodos. Cuando se intenta acceder a una propiedad o método de un objeto, JavaScript busca primero en el propio objeto y, si no lo encuentra, busca en su prototipo y así sucesivamente hasta llegar al objeto `Object.prototype`, que es la cima de la cadena de prototipos.

Todos los objetos procedentes de la misma función constructora tienen el mismo prototipo con el que enlazan. Es la parte común de los objetos del mismo tipo. Además, este enlace es dinámico por lo que cualquier cambio en el prototipo se ve reflejado en todos los objetos que lo enlazan.

El acceso al prototipo de un objeto se puede hacer con la propiedad `__proto__`, mediante una forma equivalente con el método `Object.getPrototypeOf()` o con la propiedad `prototype` de la clase en cuestión:

```
console.log(punto.__proto__);
console.log(Object.getPrototypeOf(punto));
console.log(Punto.prototype);
```

Para **modificar** un prototipo usamos la propiedad `prototype`. De esta forma podemos añadir propiedades y métodos al prototipo:

```
// Se añade una propiedad llamada 'z' con valor 10
Punto.prototype.z = 10;

// Se añade un método al prototipo
Punto.prototype.sumarXY = function() {
  return this.x + this.y;
}

// Instanciar un objeto con el operador 'new'
```

```
let punto = new Punto(5, 10);

console.log(punto.sumarXY()); // imprime '15'
```

Si una instancia de una objeto modifica el valor de una propiedad heredada, ese valor tiene **prioridad** sobre el valor del prototipo. Pasa lo mismo con los métodos, si un objeto redefine un método, éste tiene prioridad sobre el método del prototipo:

```
// Se modifica el valor de la propiedad 'z' para una instancia
punto.z = 20;

console.log(punto.z); // imprime '20'

// Podemos acceder al valor en el prototipo
console.log(punto.__proto__.z); // imprime '10'
```

Un detalle es que se puede modificar incluso el prototipo de los objetos estándar. En este ejemplo se modifica el prototipo del tipo `Array` añadiendo un nuevo método:

```
Array.prototype.obtenerPares = function() {
  return this.filter(x => x%2 == 0);
}

let a = [1, 2, 3, 4, 5, 6, 7, 8];
console.log(a.obtenerPares()); // imprime '[ 2, 4, 6, 8 ]'
```

Objetos globales predefinidos

- Más información

Math

Se trata de un objeto global que facilita la ejecución de algunas operaciones matemáticas.

Tiene disponibles algunas propiedades como `Math.E` o `Math.PI` y métodos como `Math.abs(x)` o `Math.random()`.

- Más información

Date

Este es otro objeto global que permite trabajar con fechas en JavaScript.

```
let hoy = new Date();
console.log(hoy); // Imprime '2024-01-24T18:39:49.207Z'
```

El objeto `Date` también tiene algunos métodos estáticos:

```
// Devuelve el número de milisegundos transcurridos desde el 1 de Enero de 1970, 00:00:00 UTC
console.log(Date.now()); // Imprime '1706121708552'
```

- Más información

JSON

JSON es el acrónimo de ***JavaScript Objects Notation_*** que se creó en 2001 por parte de Douglas Crockford.

La notación JSON es un formato de intercambio de datos ligero y legible por humanos. Las características clave de JSON:

- **Formato de texto**: JSON utiliza un formato de texto legible por humanos, lo que facilita su lectura y escritura. Está compuesto por pares clave-valor.
- **Datos estructurados**: Permite representar datos estructurados utilizando objetos y arreglos. Los objetos se definen entre llaves `{}` y contienen pares clave-valor separados por comas. Los arreglos se definen entre corchetes `[]` y contienen valores separados por comas:

```
{
  "nombre": "Juan",
  "edad": 30,
  "ciudad": "Ejemploville"
}

["manzana", "banana", "uva"]
```

- **Tipos de datos compatibles**: JSON admite varios tipos de datos, incluyendo cadenas de texto, números, booleanos, objetos, arreglos y el valor nulo (null).
- **Anidamiento**: Se pueden anidar objetos y arreglos dentro de otros objetos y arreglos para representar estructuras más complejas:

```
{
  "persona": {
    "nombre": "Ana",
    "edad": 25
  },
  "hobbies": ["leer", "viajar"]
}
```

- **Extensibilidad**: Aunque originalmente diseñado para JavaScript, JSON es un formato independiente de lenguaje y se utiliza en muchos entornos de programación. Es comúnmente utilizado en combinación con HTTP para intercambiar datos entre el cliente y el servidor.
- **Sintaxis estricta**: JSON tiene una sintaxis simple y estricta. Las claves y cadenas de texto deben ir entre comillas dobles, y la coma separa los elementos en objetos y arreglos.

La notación JSON es ampliamente utilizada en la transmisión y almacenamiento de datos, y su popularidad ha crecido en gran medida debido a su simplicidad y versatilidad.

JavaScript aporta un objeto global llamado `JSON` que permite manipular datos en este formato. Este objeto tiene dos métodos:

- `JSON.stringify()` : convierte un objeto o valor de JavaScript en una cadena de texto JSON, opcionalmente reemplaza valores si se indica una función de reemplazo, o si se especifican las propiedades mediante un array de reemplazo.
- `JSON.parse()` : analiza una cadena de texto en formato JSON, transformando opcionalmente el valor producido por el análisis al objeto JavaScript. Si no es correcto devuelve una excepción de tipo `SyntaxError` .

Manipulación del DOM

El **DOM** ***(Document Object Model)*** es una parte de lo que se conoce como **BOM** ***(Browser Objects Model)***.

El **BOM** no forma parte del estándar de JavaScript pero la mayoría de navegadores han igualado la forma de trabajar con estos objetos por lo que se puede hablar de un uso estandarizado de los objetos en un navegador.

El objeto raíz es el objeto **'window'**. Este objeto es tan importante que puede obviarse, es decir, se usa `alert()` en vez de `window.alert()`.

La jerarquía del **BOM** en JavaScript sigue una estructura que involucra varios objetos:

- **window (ventana)**: El objeto raíz en la jerarquía del BOM. Representa la ventana del navegador y proporciona acceso a otros objetos del BOM. Casi todas las variables y funciones globales en JavaScript son propiedades y métodos del objeto `window`.

```
// Ejemplo de acceso a la ventana actual a través de window
window.alert("Hola, mundo!");
```

- **document (documento)**: Aunque el modelo de objeto del documento (DOM) se ocupa principalmente de la estructura y contenido de la página HTML, el objeto `document` también se considera parte del BOM. Proporciona métodos para acceder y manipular el contenido HTML del documento.

```
// Ejemplo de acceso al documento a través de window
window.document.getElementById("miElemento");
```

- **navigator (navegador)**: Representa información sobre el navegador del usuario, como el nombre, la versión y las características. Este objeto posee numerosas propiedades de sólo lectura:

```
// Ejemplo de acceso al objeto navigator a través de window
window.navigator.userAgent;
```

- **location (ubicación)**: Proporciona información sobre la URL actual del documento y permite la manipulación de la ubicación del navegador.

```
// Ejemplo de acceso al objeto location a través de window
window.location.href = "https://www.ejemplo.com";
```

- **history (historial)**: Permite la manipulación del historial de navegación del usuario.

```
// Ejemplo de acceso al objeto history a través de window
window.history.back();
```

- **screen (pantalla)**: Proporciona información sobre las propiedades de la pantalla del dispositivo del usuario.

```
// Ejemplo de acceso al objeto screen a través de window
window.screen.width;
```

DOM

El Modelo de Objetos del Documento (**DOM**, por sus siglas en inglés) es una API para manipular **árboles DOM** de documentos HTML y XML (entre otros documentos en forma de árbol).

Un árbol DOM es una **estructura en forma de árbol** cuyos nodos representan el contenido de un documento HTML o XML. Cada documento HTML o XML tiene una representación de árbol DOM.

Cuando un navegador web analiza un documento HTML, crea un árbol DOM y luego lo usa para mostrar el documento.

Claramente JavaScript es el lenguaje que más se utiliza para manipular el DOM pero es una estructura de objetos independiente de todo lenguaje.

Tipos de nodos

Cada nodo que forma parte de un árbol DOM de un documento se corresponde con un tipo de nodo.

La propiedad `Node.nodeType` devuelve un *unsigned short* que representa el tipo de nodo:

```
console.log(document.nodeType); // Imprime '9'
console.log(document.head.nodeType); // Imprime '1'
```

- **Node.ELEMENT_NODE = 1**
- **Node.ATTRIBUTE_NODE = 2**
- **Node.TEXT_NODE = 3**
- **Node.CDATA_SECTION_NODE = 4**
- **Node.PROCESSING_INSTRUCTION_NODE = 7**
- **Node.COMMENT_NODE = 8**
- **Node.DOCUMENT_NODE = 9**
- **Node.DOCUMENT_TYPE_NODE = 10**
- **Node.DOCUMENT_FRAGMENT_NODE = 11**

La propiedad `Node.nodeName` devuelve una cadena que contiene el nombre del nodo

```
console.log(document.nodeName);  // Imprime '#document'
console.log(document.head.nodeName);  // Imprime 'HEAD'
```

- **Document = '#document'**
- **Comment = '#comment'**
- **Text = '#text'**
- **Element = Nombre de la etiqueta del elemento**
- **Attr = Nombre del atributo**
- Más información

Selección de elementos del DOM

El objeto `document` proporciona numerosos métodos para seleccionar elementos:

- `document.getElementById(id)` : selecciona un elemento por su identificador o *id*. Si no se encuentra el elemento retorna `null` .

- `document.getElementsByName(name)` : retorna un `NodeList` con todos los elementos que comparten el mismo atributo *name*.
- `document.getElementsByTagName(name)` : retorna un `NodeList` con todos los elementos que tengan ese nombre de etiqueta.
- `document.getElementsByClassName(name)` : retorna un `NodeList` con todos los elementos que tienen asignada una clase determinada de CSS.
- `document.querySelector(selector)` : retorna el primer elemento del documento que coincida con el grupo especificado de selectores.
- `document.querySelectorAll(selector)` : retorna un `NodeList` que representa una lista de elementos del documento que coincida con el grupo especificado de selectores.

Manipulación de atributos

- `Element.getAttribute(name)` : devuelve el valor del atributo especificado. Si no existe puede devolver o `null` o una cadena vacía `""`.

```
let list = document.querySelectorAll("li");
for (let li of lis) {
  console.log(li.getAtribute("class"));
}
```

- `Element.setAttribute(name, value)` : establece el valor de un atributo en el elemento indicado.
- `Element.removeAttribute(name)` : elimina un atributo del elemento especificado.
- `Element.toggleAttribute(name)` : permite añadir un atributo al elemento si no está añadido ya o quitar el atributo con ese nombre si ya está añadido.
- `Element.hasAttribute(name)` : devuelve un valor booleano indicando si el elemento tiene el atributo especificado o no.
- `Element.attributes` : esta propiedad retorna los atributos de un elemento en forma de `NamedNodeList`.

```
let listaAtributos = document.getElementById("id").attributes;
for (let atributo of listaAtributos) {
  console.log(`Atributo: ${atributo.name} - Valor: ${atributo.value}`);
}
```

Más información

Manipulación del contenido

La propiedad `textContent` de un elemento permite obtener y modificar el texto que contiene ese elemento y sus descendientes.

```
// Dado el siguiente fragmento HTML:
//   <div id="divA">Esto <span>es</span>un texto</div>

// Lee el contenido textual:
var text = document.getElementById("divA").textContent;
// |text| contiene la cadena "Esto es un texto".

// Escribe el contenido textual:
document.getElementById("divA").textContent = "Esto es un nuevo texto";
```

```
// El HTML "divA" ahora contiene una nueva cadena:
//   <div id="divA">Esto es un nuevo texto</div>
```

La propiedad `innerHTML` realiza la misma función que `textContent` pero interpretando las etiquetas HTML. Por temas de rendimiento es mejor utilizar `textContent`.

```
document.getElementById("p1").innerHTML = "New text!";
```

Además, al interpretar las etiquetas HTML existe un riesgo de seguridad. Aunque HTMl5 especifica que no se deben ejecutar etiquetas `<script>` insertadas a través de `innerHTML` hay maneras de ejecutar JavaScript sin necesidad de utilizar el elemento `<script>`:

```
//ejemplo 1
var name = "Juan";
// asumiendo que 'el' es un elemento de HTML DOM
el.innerHTML = name; // sin peligro

// ...

//ejemplo 2
name = "<script>alert('Soy Juan con una alerta molesta!')</script>";
el.innerHTML = name; // fíjese que el texto es molesto y no es realmente lo que se esperaba.

const name = "<img src='x' onerror='alert(1)'>";
el.innerHTML = name; // con peligro, la alerta ahora si es mostrada
```

Modificar CSS

Es posible modificar el CSS de los elementos a través de los atributos `style` o `class` mediante el método `setAttribute()`.

Sin embargo, al ser un característica tan importante, JavaScript proporciona métodos especiales para manipular las clases CSS de un elemento.

Los elementos poseen la propiedad `style` que permite acceder directamente a las propiedades CSS de un elemento.

Las propiedades se escriben en formato **CamelCase** como por ejemplo la propiedad CSS `background-color` se indica como `backgroundColor`:

```
let element = document.getElementById("id");
element.style.backgroundColor = "green";
```

La mayoría de navegadores actuales acepta también el formato idéntico a CSS a través de corchetes:

```
let element = document.getElementById("id");
element.style["background-color"] = "green";
```

Mediante el método `window.getComputedStyle(elemento)` se pueden consultar las propiedades CSS que se están aplicando a un elemento concreto:

```
let element = document.getElementById("id");

console.log(window.getComputedStyle(element)); // Imprime todas las propiedades
```

```
console.log(window.getComputedStyle(element).fontFamily); // Imprime la propiedad 'fontFamily'
console.log(window.getComputedStyle(element).margin); // Imprime una propiedad 'margin'
```

Los elementos disponen de la propiedad `className` que permite asignar una clase CSS a un elemento o consultar las clases aplicadas:

```
// <p id="intro" class="remarcado">Finding HTML Elements by Id</p>

let intro = document.getElementById("intro");
console.log(intro.className); // imprime 'remarcado'

intro.className = "fondo";  // sustituye a la clase anterior
console.log(intro.className); // imprime 'fondo'
```

Cuando un elemento tiene más de una clase CSS o se quiere añadir más clases y no sobreescribir la clases anteriores, JavaScript incorpora la propiedad `classList` a los elementos del DOM.

```
for (let clase of element.classList) {
  console.log(clase);
}
```

La propiedad `classList` dispone de una serie de métodos para facilitar el trabajo:

- `add(nombreClase1 [,nombreClaseN])`
- `remove(nombreClase1 [,nombreClaseN])`
- `toggle(nombreClase1 [,forzar])`
- `contains(nombreClase)`
- `replace(nombreViejo, nombreNuevo)`

```
const div = document.createElement("div");
div.className = "foo";

// our starting state: <div class="foo"></div>
console.log(div.outerHTML);

// use the classList API to remove and add classes
div.classList.remove("foo");
div.classList.add("anotherclass");

// <div class="anotherclass"></div>
console.log(div.outerHTML);

// if visible is set remove it, otherwise add it
div.classList.toggle("visible");

// add/remove visible, depending on test conditional, i less than 10
div.classList.toggle("visible", i < 10);

// false
console.log(div.classList.contains("foo"));

// add or remove multiple classes
div.classList.add("foo", "bar", "baz");
div.classList.remove("foo", "bar", "baz");

// add or remove multiple classes using spread syntax
const cls = ["foo", "bar"];
div.classList.add(...cls);
div.classList.remove(...cls);
```

```
// replace class "foo" with class "bar"
div.classList.replace("foo", "bar");
```

Atributos DATA

En los documentos HTML existe la posibilidad de crear atributos **data**, que son atributos de datos personalizables y modificables.

```
// <p id="libro" data-tipo="novela" data-autor="Miguel de Cervantes"></p>

let libro = document.getElementById("libro");
console.log(libro.dataset.tipo);
console.log(libro.dataset.autor);
```

Navegar por el DOM

JavaScript propone una serie de métodos para navegar por el DOM:

- `Node.childNodes` : propiedad de sólo lectura que devuelve una colección de nodos. Este método también devuelve texto y comentarios como nodos.
- `Node.children` : propiedad de sólo lectura que devuelve una colección viva de nodos hijos. Únicamente devuelve los elementos.
- `Node.firstChild`
- `Node.lastChild`
- `Node.nextSibling`
- `Node.previousSibling`
- `Node.parentNode`
- `Node.firstElementChild`
- `Node.lastElementChild`
- `Node.nextElementSibling`
- `Node.previousElementSibling`

Las propiedades tienen dos versiones. Las que incluyen `Element` sólo incluyen los nodos que son elementos como `<p>` , `<div>` , `<h1>` , etcétera…

```
<div id="miDiv" class="miClase">
  <p>Este es un párrafo dentro del div</p>
  <ul>
      <li>Elemento 1</li>
      <li>Elemento 2</li>
      <li>Elemento 3</li>
  </ul>
</div>

<script>
  // Obtener un elemento por su ID
  var miDiv = document.getElementById("miDiv");
```

```
  console.log("Elemento por ID:", miDiv);

  // Obtener elementos por su clase
  var elementosPorClase = document.getElementsByClassName("miClase");
  console.log("Elementos por clase:", elementosPorClase);

  // Obtener el primer párrafo dentro del div
  var primerParrafo = miDiv.querySelector("p");
  console.log("Primer párrafo:", primerParrafo);

  // Obtener todos los elementos de la lista dentro del div
  var elementosLista = miDiv.querySelectorAll("ul li");
  console.log("Elementos de la lista:", elementosLista);

  // Acceder al padre del div
  var padreDelDiv = miDiv.parentNode;
  console.log("Padre del div:", padreDelDiv);

  // Acceder al primer hijo del div
  var primerHijoDelDiv = miDiv.firstChild;
  console.log("Primer hijo del div:", primerHijoDelDiv);

  // Acceder al siguiente hermano del div
  var siguienteHermanoDelDiv = miDiv.nextSibling;
  console.log("Siguiente hermano del div:", siguienteHermanoDelDiv);

  var lista = miDiv.querySelector("ul");
  console.log(lista.childNodes.length); // imprime '7' ya que incluye el texto
  console.log(lista.children.length); // imprime '3', sólo cuenta los '<li>'

</script>
```

Modificar el DOM

```
<div id="miDiv" class="miClase">
  <p>Este es un párrafo dentro del div</p>
  <ul>
      <li>Elemento 1</li>
      <li>Elemento 2</li>
      <li>Elemento 3</li>
  </ul>
</div>

<script>
  // Permite crear elementos pero que aún no se ha añadido en el árbol DOM
  let nuevoDiv = document.createElement("div");

  // Se puede usar la referencia para acceder a las propiedades
  nuevoDiv.innerHTML = "<p>Esta es una nueva capa</p>";
  nuevoDiv.setAttribute("id", "nuevaCapa");

  // El nuevo 'div' Se añade al final del 'body'
  document.body.appendChild(nuevoDiv);

  // Se crea un nodo de tipo 'texto'
  let nodoTexto = document.createTextNode("Texto");
  document.body.appendChild(nodoTexto);

  let ul = document.querySelector("ul");
  let li = document.createElement("li");
  li.textContent = "Elemento 4";

  // Inserta el nuevo 'li' como nodo
  ul.appendChild(li);
</script>
```

Colección 'viva'

En JavaScript, una "colección viva" (*live collection*) se refiere a un tipo especial de colección de nodos del DOM que se actualiza automáticamente cuando cambia el documento.

Esto significa que si se realizan cambios en el documento después de que la colección se haya creado, la colección se actualizará automáticamente para reflejar esos cambios, sin necesidad de volver a seleccionar los elementos.

Ejemplos comunes de colecciones vivas son aquellas obtenidas mediante métodos como `getElementsByClassName` o `getElementsByTagName`, `attributes`, `classList` o `children`.

Sin embargo, el método `querySelectorAll()` devuelve una **colección estática** por lo que no se actualizan automáticamente si se añaden o eliminan nodos.

```
<ul>
  <li class="item">Elemento 1</li>
  <li class="item">Elemento 2</li>
  <li class="item">Elemento 3</li>
</ul>

<script>
// Obtener una colección viva de elementos con la clase "item"
var elementos = document.getElementsByClassName("item");

// Imprimir la longitud de la colección
console.log("Longitud inicial:", elementos.length);

// Agregar un nuevo elemento a la lista
var nuevoElemento = document.createElement("li");
nuevoElemento.textContent = "Elemento 4";
document.querySelector("ul").appendChild(nuevoElemento);

// La colección se actualiza automáticamente
console.log("Longitud después de agregar un elemento:", elementos.length);
</script>
```

Temporizadores

En JavaScript, los temporizadores son funciones que permiten ejecutar código después de un cierto período de tiempo o en intervalos regulares.

Aquí hay un resumen de los temporizadores más comunes en JavaScript:

- **`setTimeout(función, tiempo)`**: ejecuta la función después de un cierto período de tiempo (en milisegundos). Retorna un identificador de temporizador que puede ser utilizado para cancelar la ejecución planificada.

```
setTimeout(function() {
    console.log("¡Hola después de 2000 ms!");
}, 2000);
```

- **`clearTimeout(identificador)`**: cancela la ejecución de una función programada con `setTimeout()` antes de que ocurra.

```
var timeoutID = setTimeout(function() {
    console.log("Esta ejecución se cancelará");
}, 2000);
```

```
clearTimeout(timeoutID); // Cancela la ejecución programada
```

- `setInterval(función, intervalo)`: ejecuta la función repetidamente con un intervalo de tiempo entre cada ejecución (en milisegundos). Retorna un identificador de intervalo que puede ser utilizado para detener la ejecución repetida.

```
var intervaloID = setInterval(function() {
    console.log("¡Hola cada 1000 ms!");
}, 1000);
```

- `clearInterval(identificador)`: detiene la ejecución repetida de una función programada con setInterval.

```
var intervaloID = setInterval(function() {
    console.log("¡Hola cada 1000 ms!");
}, 1000);

clearInterval(intervaloID); // Detiene la ejecución repetida
```

Estos temporizadores son útiles para tareas como animaciones, actualizaciones periódicas de contenido, y otras situaciones en las que es necesario controlar el tiempo. Sin embargo, es importante usarlos con precaución para evitar problemas de rendimiento y asegurarse de cancelarlos cuando ya no son necesarios para evitar posibles fugas de memoria.

Gestión de eventos

Los eventos son el mecanismo fundamental para comunicar la aplicación con el usuario.

Un evento no es más que un suceso, algo que ha ocurrido como resultado de un acto del usuario o por otras razones. Para que se considere realmente un evento, la aplicación tien que ser capaz de detectarlo y tiene que ser capaz de ejecutar el código asociado a dicho evento.

Los eventos se asocian a un elemento del DOM.

La clave de los eventos en JavaScript es su capacidad asíncrona.

- Más información

Mecanismos obsoletos

El primer método para el registro de manejadores de eventos en la Web, utilizaba atributos HTML para manejar eventos (o manejadores de eventos en línea).

Sin embargo no se recomienda su uso ya que es una mala práctica. Mezclar código JavaScript con HTML dificulta la lectura y el mantenimiento del código:

```
<!-- Mecanismo en línea -->
<button onclick="alert('¡Hola, este es un manejador de eventos anticuado!');">
  Haz click
</button>

<!-- Otro mecanismo en línea con función en JavaScript -->
<button onclick="bgChange()">Haz clic</button>

<script>
function bgChange() {
```

```
  const rndCol = `rgb(${random(255)}, ${random(255)}, ${random(255)})`;
  document.body.style.backgroundColor = rndCol;
}
</script>
```

Los elementos tienen propiedades que permiten mejorar la legibilidad del código separando el código JavaScript del HTML:

```
<p id="parrafo1">Párrafo clickable</p>

<script>
let parrafo = document.getElementById("parrafo1");
parrafo.onclick = () => {alert('click')};
</script>
```

Mecanismo recomendado

Sin embargo, el método aconsejable es utilizar el método `addEventListener` que está disponible en todos los elementos.

A este método hay que indicarle el nombre del evento y la función *callback* a ejecutar.

La ventaja frente a los métodos obsoletos es que se puede asignar más de una función al mismo evento. También es posible eliminar la función.

Esta función se le denomina **manejador de eventos**. Cuando estas funciones se configuran para ejecutarse en respuesta a un evento, se dice que se está **registrando un manejador de eventos**:

```
<button>Cambiar el color</button>

<script>
const btn = document.querySelector("button");

function random(number) {
  return Math.floor(Math.random() * (number + 1));
}

btn.addEventListener("click", () => {
  const rndCol = `rgb(${random(255)}, ${random(255)}, ${random(255)})`;
  document.body.style.backgroundColor = rndCol;
});
</script>
```

Para eliminar un manejador de eventos, se utiliza la función `removeEventListener()`. Es necesario que la función tenga **nombre** por lo que no podemos eliminar manejadores con funciones anónimas:

```
btn.removeEventListener("click", changeBackground);
```

Propagación de los eventos

Una cuestión fundamental en la captura de eventos es cómo se propagan los eventos sobre estructuras con contenedores y elementos anidados, como por ejemplo una estructura tal que así:

```
<body>
  <div id="outer">
    <p>
      <button>Haz click</button>
```

```
    </p>
  </div>
</body>
```

La propagación de eventos se divide en dos fases:

- **Fase de captura** que va desde el elemento contenedor padre hacia los elementos anidados.
- **Fase de burbuja** que va ascendiendo desde el elemento anidado hacia el contenedor padre.

Por defecto, la función asociada al evento se lanza en la **fase de burbuja**:

```
let div = document.querySelector("#outer");
div.addEventListener("click", () => {console.log("CLICK DIV")});

let button = document.querySelector("button");
button.addEventListener( "click", () => {console.log("CLICK BUTTON")});

// En consola se imprimirá:
// 'CLICK BUTTON'
// 'CLICK DIV'
```

Para evitar que la propagación "ascienda" de los elementos anidados hasta los elementos contendores, se puede utilizar la función `stopPropagation()` del objeto `Event`:

```
let div = document.querySelector("#outer");
div.addEventListener("click", message);

let button = document.querySelector("button");
button.addEventListener( "click", message);

function message(event) {
  event.stopPropagation();
  console.log(`CLICK ${event.currentTarget.tagName}`);
}

// En consola solo se imprimirá:
// 'CLICK BUTTON'
```

Es posible modificar el comportamiento de la propagación de eventos de forma que los manejadores de eventos se ejecuten en la **fase de captura** añadiendo un tercer parámetro `{capture: true}` a la función `addEventListener`:

```
let div = document.querySelector("#outer");
div.addEventListener("click", message, {capture: true});

let button = document.querySelector("button");
button.addEventListener( "click", message, {capture: true});

function message(event) {
  console.log(`CLICK ${event.currentTarget.tagName}`);
}

// En consola se imprimirá:
// 'CLICK DIV'
// CLICK BUTTON'
```

Objeto 'Event'

Los gestores de eventos pueden estar atados a varios elementos en el DOM. Cuando un evento ocurre, un objeto de evento es dinámicamente creado y pasado secuencialmente a las "escuchas" (*listeners*) autorizadas para la gestión del evento.

La interfaz `Event` del DOM es entonces accesible por la función de manejo, vía el objeto de evento puesto como el primer (y único) argumento:

```
let button = document.querySelector("button");
button.addEventListener( "click", message);

function message(event) {
  // objeto 'Event'
  console.log(`CLICK ${event.currentTarget.tagName}`);
}
```

Algunas de las propiedades de este objeto `Event` :

- `event.bubbles` : Devuelve un valor que indica si el evento se propaga hacia arriba a través del DOM o no.
- `event.cancelable` : Devuelve un valor que indica si el evento se puede cancelar.
- `event.currentTarget` : Devuelve una referencia al objetivo actual registrado para el evento.
- `event.target` : Devuelve una referencia al objetivo en la cual el evento fue originalmente enviado.
- `event.type` : Devuelve el nombre del evento (distingue mayúsculas y minúsculas).

Además de las propiedades, los objetos de evento poseen métodos. Uno de los más importantes es el método sin parámetros `preventDefault()` .

Se utiliza este método para evitar o cancelar el comportamiento por defecto del elemento que recibe el evento.

Algunos de los comportamientos por defecto que aplican los navegadores según el elemento HTML:

- `<form>` : se envía los datos del formulario al servidor y recarga la página.
- `<input type="submit">` o `<button type="submit">` : se envía el formulario
- `<input type="reset">` o `<button type="reset">` : se restablece los campos del formulario a sus valores iniciales.
- `<input type="button">` o `<button type="button">` : no tiene una acción predeterminada
- `<a>` (como un botón dentro de un formulario): puede redirigir a otra página.

Tipos de eventos

Existen multitud de eventos clasificados en MDN como por ejemplo el evento `click` utilizado en los ejemplos:

```
let button = document.querySelector("button");
button.addEventListener( "click", message);
```

Estos eventos se clasifican en **categorías**, como por ejemplo:

- **MouseEvent**: eventos como por ejemplo `click` , `dblclick` , `mousenter` , `mouseleave` …
- **KeyboardEvent**: eventos como por ejemplo `keypress` , `keydown` , `keyup` …

Gestión de excepciones

Las excepciones en JavaScript son generalmente objetos globales de tipo `Error` o alguno de sus subtipos:

- **EvalError**: instancia que representa un error que ocurre con respecto a la función global `eval()`.
- **InternalError**: instancia que representa un error que ocurre cuando se produce un error interno en el motor de JavaScript.
- **RangeError**: instancia que representa un error que ocurre cuando una variable numérica o parámetro está fuera de su rango válido.
- **ReferenceError**: instancia que representa un error que ocurre cuando se quita la referencia a una referencia no válida.
- **SyntaxError**: instancia que representa un error de sintaxis.
- **TypeError**: instancia que representa un error que ocurre cuando una variable o parámetro no es de un tipo válido.
- **URIError**: instancia que representa un error que ocurre cuando `encodeURI()` o `decodeURI()` pasan parámetros no válidos.

Se puede usar el objeto `Error` como objeto base para errores definidos por el usuario o usar uno de los subtipos. El objeto `Error` tiene las propiedades `name` y `message`:

```
try {
  throw new Error("¡Ups!");
} catch (e) {
  console.error(`${e.name}: ${e.message}`);
}
```

Para capturar una excepción se utiliza un bloque `try...catch` o `try...catch...finally` según las necesidades. Se puede manejar errores específicos con `instanceof`:

```
try {
  foo.bar();
} catch (e) {
  if (e instanceof EvalError) {
    console.error(`${e.name}: ${e.message}`);
  } else if (e instanceof RangeError) {
    console.error(`${e.name}: ${e.message}`);
  }
  // ... code
}

try {
  foo.bar();
} catch (e) {
  console.error(`${e.name}: ${e.message}`);
} finally {
  // ... code
}
```

- Más información

Modo estricto

En JavaScript, `'use strict'` es una declaración que se utiliza para activar el modo estricto en un script o función. Introducido a partir del estándar **ES6**, este modo introduce un conjunto más riguroso de reglas y restricciones en el código, ayudando a detectar y prevenir errores comunes, así como a mejorar la calidad y la consistencia del código:

- **Prevención de errores**: este modo ayuda a prevenir errores comunes y silenciosos al lanzar excepciones para prácticas consideradas ambiguas o propensas a errores.
- **Eliminación de ambigüedades**: en el modo estricto, ciertas acciones que eran permitidas en el modo no estricto ahora lanzarán errores, eliminando ambigüedades y mejorando la claridad del código.
- **Mejoras en la seguridad**: algunas acciones que podrían ser peligrosas o inseguras se vuelven errores en el modo estricto, mejorando así la seguridad del código.
- **Mejoras en el rendimiento**: algunas optimizaciones de rendimiento están habilitadas en el modo estricto, lo que puede llevar a un código más eficiente en términos de ejecución.

Mientras que sin el modo estricto determinados problemas únicamente se muestran con un **warning**, cuando se declara el modo estricto se muestra una **excepción**.

Este modo se puede activar a nivel de **script o fichero** y a nivel de **función**:

```
'use strict';

// En el modo estricto se muestra una excepción :
// 'Uncaught ReferenceError: assignment to undeclared variable x'
x=9;  //
console.log(x);

// Declaración estricta dentro de una función
function f(param) {
  'use strict';
  // ....
}
```

Hay que tener en cuenta que si se utilizan módulos ECMAScript 6 (ES6), el modo estricto se activa **automáticamente** en cada módulo.

Módulos

Con la introducción de ECMAScript 2015 (también conocido como ES6), JavaScript incorporó **soporte nativo para módulos**, proporcionando una forma más organizada y modular de estructurar y gestionar el código en aplicaciones web.

Los módulos en JavaScript permiten encapsular piezas específicas de código en archivos separados, lo que facilita la organización y mantenimiento del código fuente. Al utilizar las palabras clave `import` y `export`, los desarrolladores pueden definir qué **funciones**, **variables** o **clases** están disponibles para ser utilizadas en otros módulos.

Un archivo de módulo puede exportar múltiples valores y otro módulo puede importar solo aquellos que necesita, lo que reduce la complejidad y las dependencias innecesarias. Además, los módulos tienen su propio ámbito (*scope*), evitando la contaminación del espacio global y permitiendo una mayor encapsulación.

La sintaxis básica de los módulos en JavaScript ES6 se ve así:

```
// modulo.js
export const miVariable = 42;

export function miFuncion() {
  console.log("Hola desde miFuncion");
}
```

Una forma alternativa de exportar los elementos es usar una sola declaración de exportación al final del archivo:

```
// modulo.js
const miVariable = 42;

function miFuncion() {
  console.log("Hola desde miFuncion");
}

// Export
export {miVariable, miFuncion};
```

Se utiliza la declaración `import` para importar los elementos de un módulo:

```
// main.js
import { miVariable, miFuncion } from './modulo.js';

console.log(miVariable); // Imprime 42
miFuncion(); // Imprime "Hola desde miFuncion"
```

Los elementos importados se pueden renombrar, ya sea para facilitar la legibilidad o para evitar problemas de nombres mediante la palabra clave `as`:

```
// main.js
import { miVariable as myVar, miFuncion as myFunction } from './modulo.js';
```

También se pueden importar todos los elementos de un módulo, pero en ese caso se tiene que asignar un nombre que se utilizará como **espacio de nombres**:

```
// main.js
import  * as myModule from './modulo.js';

console.log(myModule.miVariable); // Imprime 42
myModule.miFuncion(); // Imprime "Hola desde miFuncion"
```

Si se importa el módulo en HTML, se tiene que indicar en la etiqueta `<script>`:

```
<script type="module">
import * as myModule from './js/modulo.mjs';

console.log(myModule.miVariable); // Imprime 42
myModule.miFuncion(); // Imprime "Hola desde miFuncion"
</script>
```

- Más información

Programación asíncrona

La **asincronía** en JavaScript es un concepto fundamental que se refiere a la capacidad del lenguaje para realizar operaciones sin bloquear la ejecución del código. En lugar de esperar que una tarea se complete antes de pasar a la siguiente, JavaScript permite la ejecución de múltiples operaciones de manera concurrente mediante el uso de **callbacks**, **promesas** y ***async/await***.

Por tanto JavaScript se considera un **lenguaje no bloqueante o asíncrono**.

Esta característica es esencial para manejar operaciones que pueden llevar tiempo, como la lectura de archivos, solicitudes de red o procesos intensivos en recursos. La asincronía permite que el programa continúe ejecutándose mientras espera la finalización de otras tareas, mejorando así la eficiencia y la capacidad de respuesta de las aplicaciones.

El manejo adecuado de la asincronía es crucial para evitar bloqueos en la interfaz de usuario y mejorar la experiencia del usuario en aplicaciones web, donde las operaciones asíncronas son comunes.

En lenguajes síncronos la única manera de conseguir asíncronía es crear varios hilos independientes en los que cada uno realiza una tarea. Así funciona por ejemplo Java. Sin embargo, JavaScript es un lenguaje de un sólo hilo, no se pueden programar dos hilos a la vez. JavaScript es asíncrono por naturaleza, por lo que no es necesario trabajar con múltiples hilos.

Callbacks

Probablemente, la forma más **clásica** de gestionar la asincronía en Javascript. Es una práctica poco recomendable ya que cuando las funciones *callback* se van anidando dificulta la legibilidad del código y es más propenso a errores. Esto se denomina ***callback hell***:

```
// Ejemplo de callback hell
function iniciarProceso() {
  realizarPrimeraAccion(function(resultado1) {
    console.log(resultado1);

    realizarSegundaAccion(function(resultado2) {
      console.log(resultado2);

      realizarTerceraAccion(function(resultado3) {
        console.log(resultado3);

        // Y así sucesivamente...
      });
    });
  });
}

function realizarPrimeraAccion(callback) {
  setTimeout(function() {
    callback("Primera acción completada");
  }, 1000);
}

function realizarSegundaAccion(callback) {
  setTimeout(function() {
    callback("Segunda acción completada");
  }, 1000);
}

function realizarTerceraAccion(callback) {
  setTimeout(function() {
    callback("Tercera acción completada");
  }, 1000);
}

// Llamamos a la función que inicia el proceso
iniciarProceso();
```

De todas formas, las funciones *callback* no son más que un tipo de funciones que se pasan por parámetro a otras funciones. Además, los parámetros de dichas funciones toman un valor especial en el contexto del interior de la función.

Promesas

La solución al problema del *callback hell* es una estructura que permite controlar de forma más organizada las tareas asíncronas. Esta nueva estructura es lo que se conoce como **promesa**.

La norma ES2015 incorporó las promesas al estándar de JavaScript.

Las **promesas** en JavaScript se representan a través de un objeto de tipo `Promise`.

Cada promesa puede estar en un estado concreto: **pendiente (*pending*)**, **aceptada (*resolved o fullfilled*)** o **rechazada (*rejected*)**.

Estos objetos de tipo `Promise` son los que hacen la labor de relacionar la tarea asíncrona con las acciones a tomar en caso de éxito o error. Para ello proporcionan varios métodos:

- `.then(resolve)`: ejecuta la función *callback* `resolve` con un parámetro cuando la promesa se cumple.
- `.catch(reject)`: ejecuta la función *callback* `reject` con un parámetro cuando la promesa se rechaza.
- `.then(resolve, reject)`: método equivalente a las dos anteriores en el mismo `.then()`.
- `.finally(end)`: ejecuta la función *callback* `end` sin parámetros tanto si se cumple como si se rechaza.

La forma general de consumir una promesa es utilizando el `.then()` con un sólo parámetro, puesto que muchas veces lo único que nos interesa es realizar una acción cuando la promesa se cumpla:

```
fetch("https://google.es").then(function(response) {
  /* Código a realizar cuando se cumpla la promesa */
});
```

El método `.catch()` permite gestionar el rechazo o error en la promesa:

```
fetch("https://google.es")
  .then(function(response) {
    /* Código a realizar cuando se cumpla la promesa */
  })
  .catch(function(error) {
    /* Código a realizar cuando se rechaza la promesa */
  });
```

Se pueden encadenar varios `.then()` si se siguen generando promesas y se devuelven con un `return`. Cada una de ellas se ejecutada en el orden en el que son insertadas pero nunca antes de que la anterior termine con **éxito**, ya que en caso contrario se ejecuta el `.catch()`:

```
fetch("https://google.es")
  .then(response => {
    return response.text(); // Devuelve una promesa
  })
  .then(data => {
    console.log(data);
  })
  .catch(error => { /* Código a realizar cuando se rechaza la promesa */ })
```

De hecho, usando **funciones flecha** se puede mejorar aún más la legibilidad del código, recordando que cuando sólo tenemos una sentencia en el cuerpo de la función flecha hay un `return` implícito:

```
fetch("https://google.es")
  .then(response => response.text())
  .then(data => console.log(data))
```

```
  .finally(() => console.log("Terminado"))
  .catch(error => console.error(data));
```

Se puede añadir un método `.finally()` para añadir una función *callback* que se ejecutará tanto si la promesa **se cumple o se rechaza**.

Una `Promise` puede ser creada desde cero usando su constructor:

```
// Ejemplo de creación de una Promise
function ejecutarOperacionAsincrona() {
  return new Promise((resolve, reject) => {
    // Simulamos una operación asincrónica, por ejemplo, una solicitud a una API
    setTimeout(() => {
      const exito = true; // Cambia a false para simular un error

      if (exito) {
        resolve("La operación fue exitosa");
      } else {
        reject("Hubo un error en la operación");
      }
    }, 2000); // Simulamos un retardo de 2 segundos
  });
}

// Utilizamos la Promise
ejecutarOperacionAsincrona()
  .then(resultado => {
    console.log(resultado);
  })
  .catch(error => {
    console.error(error);
  });
```

En este ejemplo, `ejecutarOperacionAsincrona` es una función que devuelve una `Promise`. Dentro del constructor de `Promise`, hay una función que toma dos parámetros: `resolve` y `reject`. Si la operación es exitosa, sed llama a `resolve` con el resultado deseado mientras que si hay un error, se llama a `reject` con un mensaje de error.

El objeto `Promise` aporta el método estático `Promise.resolve` que crea una nueva promesa y enviando el objeto que se envíe como parámetro:

```
let promesa = Promise.resolve("Todo OK");
promesa.then(respuesta => console.log(respuesta)); // imprime 'Todo OK'
```

El método estático contrario es `Promise.rejected`:

```
let promesa = Promise.reject(Error("Todo KO"));

promesa
  .then(respuesta => console.log(respuesta))
  .catch(error => console.log(error.message)); // imprime 'Todo KO'
```

Por último, `Promise.all` es útil cuando hay varias promesas y hay que esperar a que todas se resuelvan antes de realizar alguna acción. Es una forma de sincronizar acciones asíncronas:

```
// Ejemplo de uso de Promise.all
function obtenerDatosDeServicios() {
  // Simulamos dos solicitudes a servicios diferentes
```

```
  const servicio1 = obtenerDatosDeServicio1();
  const servicio2 = obtenerDatosDeServicio2();

  // Utilizamos Promise.all para esperar a que ambas promesas se resuelvan
  return Promise.all([servicio1, servicio2])
    .then(resultados => {
      // resultados es un array con los resultados de ambas promesas
      const resultadoServicio1 = resultados[0];
      const resultadoServicio2 = resultados[1];

      console.log("Datos del Servicio 1:", resultadoServicio1);
      console.log("Datos del Servicio 2:", resultadoServicio2);

      return "Todas las operaciones fueron exitosas";
    })
    .catch(error => {
      // Si alguna de las promesas falla, se maneja aquí
      console.error("Hubo un error en al menos una operación:", error);
    });
}

// Funciones simuladas que devuelven promesas
function obtenerDatosDeServicio1() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve("Datos del Servicio 1");
    }, 1500);
  });
}

function obtenerDatosDeServicio2() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve("Datos del Servicio 2");
    }, 1000);
  });
}

// Llamamos a la función que utiliza 'Promise.all'
obtenerDatosDeServicios()
  .then(resultado => {
    console.log(resultado);
  });
```

Es importante resaltar que, si alguna promesa se rechazara o provocara un error, instantáneamente se generaría el rechazo sin esperar al resto de promesas.

- Más información

Async/await

`async/await` es una característica introducida en ECMAScript 2017 (ES8) que simplifica y mejora la legibilidad del código asíncrono en JavaScript. Esta combinación de palabras clave proporciona una forma más concisa y fácil de trabajar con **Promesas**, haciéndolas parecer más similares a código síncrono.

- `async`: se utiliza para declarar una función como asíncrona. Una función asíncrona siempre devuelve una `Promise`, y cualquier valor devuelto por la función es automáticamente envuelto en una `Promise` resuelta.
- `await`: se utiliza dentro de funciones declaradas con `async`. Permite pausar la ejecución de la función hasta que la `Promise` dada se resuelva, y luego reanudar la ejecución con el resultado de la `Promise`.

```
// Ejemplo de 'async/await'
function obtenerDatosUsuario(id) {
  return new Promise(resolve => {
```

```
    setTimeout(() => {
      // Simulamos obtener datos de un servicio
      const datos = {
        id: id,
        nombre: "Usuario Ejemplo",
        correo: "usuario@example.com"
      };
      resolve(datos);
    }, 1000);
  });
}

async function mostrarDatosUsuario() {
  try {
    // Esperamos a que se resuelva la 'Promise' antes de continuar
    const datosUsuario = await obtenerDatosUsuario(123);

    // Una vez que la 'Promise' se resuelve, continuamos con la ejecución
    console.log("Datos del usuario:", datosUsuario);
  } catch (error) {
    // Capturamos cualquier error que ocurra durante la ejecución de la 'Promise'
    console.error("Error al obtener datos del usuario:", error);
  }
}

// Llamamos a la función que utiliza 'async/await'
mostrarDatosUsuario();
```

Comunicación cliente/servidor

Un navegador, durante la carga de una página, suele realizar múltiples peticiones HTTP a un servidor para solicitar los archivos que necesita renderizar en la página, como la hoja de estilos CSS, scripts JS, imágenes, etcétera…

Por lo tanto, cuando cargamos una página (primera petición), en realidad estamos realizando múltiples peticiones posteriormente.

Una petición HTTP es como suele denominarse a la acción por parte del navegador de solicitar a un servidor web un documento o archivo, almacenarlo en un caché temporal de archivos del navegador y, finalmente, mostrarlo en la página actual que lo ha solicitado.

Políticas CORS

CORS *(Cross-Origin Resource Sharing)* es un conjunto de políticas de seguridad implementadas en navegadores web para gestionar solicitudes de recursos entre dominios. Estas restricciones son cruciales para prevenir solicitudes no autorizadas desde scripts en un dominio hacia recursos en otro dominio, mitigando posibles riesgos de seguridad.

Cuando se realiza una solicitud desde un dominio A a un dominio B a través de JavaScript en el navegador, CORS impone restricciones.

El servidor en el dominio B debe incluir encabezados CORS específicos en sus respuestas para indicar qué dominios (*origins*) tienen permiso para acceder a sus recursos.

El encabezado `Access-Control-Allow-Origin` indica qué dominios tienen permiso para acceder a los recursos. Si el dominio de origen no está permitido, se bloquea la solicitud.

Por otra parte el encabezado `Access-Control-Allow-Methods` especifica los métodos HTTP permitidos para la solicitud (GET, POST, etc.).

CORS es esencial para prevenir ataques de solicitudes entre sitios (CSRF) y garantizar que las interacciones entre dominios sean seguras y autorizadas.

- Más información

AJAX

Asynchronous JavaScript and XML es el acrónimo que se esconde bajo el término de AJAX. Es una técnica de desarrollo web que permite la comunicación asincrónica entre el cliente y el servidor, sin necesidad de recargar la página completa.

El hecho de que haga referencia a XML es porque cuando se ideó esta tecnología, XML era el formato documental predominante y los datos se enviaban en este formato. Actualmente el formato predominante es **JSON**, aunque por razones fonéticas evidentes se sigue manteniendo el término AJAX en lugar del horrible correspondiente "AJAJ".

AJAX ha sido un pilar en el desarrollo web, permitiendo la actualización dinámica de contenido sin recargar por completo las páginas. Sin embargo, con la evolución del lenguaje y las tecnologías web, la API `fetch` ha surgido como una alternativa más moderna y flexible para realizar solicitudes asíncronas en JavaScript.

El método tradicional de realizar solicitudes asíncronas implicaba el uso del objeto `XMLHttpRequest`. Este objeto proporcionaba una interfaz para realizar solicitudes HTTP y manejar las respuestas de manera asíncrona. Aunque efectivo, su sintaxis y manejo de eventos podían ser verbosos:

```
var xhttp = new XMLHttpRequest();
xhttp.open("GET", "ejemplo.txt", true);
xhttp.onreadystatechange = function() {
  if (this.readyState == 4 && this.status == 200) {
    // Manejar la respuesta del servidor
    console.log(this.responseText);
  }
};
xhttp.send();
```

La API `fetch` ofrece una sintaxis más limpia y una interfaz de promesas, simplificando el código y haciéndolo más legible y fácil de mantener. Además, `fetch` está diseñado para trabajar bien con las características modernas de JavaScript, como las promesas:

```
fetch("ejemplo.txt")
  .then(response => {
    if (!response.ok) {
      throw new Error("Error en la solicitud HTTP, código " + response.status);
    }
    return response.text();
  })
  .then(data => {
    // Manejar la respuesta del servidor
    console.log(data);
  })
  .catch(error => {
    console.error("Error en la solicitud: ", error);
  });
```

- Más información
- Especificaciones de Fetch

Peticiones con `fetch`

La API `fetch` proporciona un método global que se llama `fetch(url)`. Este método toma un argumento obligatorio, la ruta de acceso al recurso que desea recuperar.

El resultado es una `Promise` que se considera resuelta cuando se reciben resultados sin error del destino:

```
// URL de la API de ejemplo
const apiUrl = 'https://jsonplaceholder.typicode.com/todos/1';

// Utilizando fetch para realizar una solicitud GET
fetch(apiUrl)
  .then(response => {
    // Verificar si la solicitud fue exitosa (código de estado 200-299)
    if (!response.ok) {
      throw new Error(`Error de red - Código: ${response.status}`);
    }
    // Convertir la respuesta a formato JSON
    return response.json();
  })
  .then(data => {
    // Manejar los datos obtenidos de la respuesta
    console.log('Datos recibidos:', data);
  })
  .catch(error => {
    // Capturar y manejar cualquier error durante el proceso
    console.error('Error durante la solicitud:', error.message);
  });
```

Manipular el objeto `Response`

Una vez que el objeto `Response` es recuperado, dispone de varias propiedades con información de la respuesta recibida:

- **headers**: obtiene un objeto que contiene las cabeceras http del paquete de respuesta.
- **body**: cuerpo de la respuesta http.
- **status**: devuelve el código de respuesta de la petición http, que se corresponden a los estándares http (100, 200, 300, 400, 500, etcétera…)
- **statusText**: devuelve un texto descriptivo del código de respuesta
- **ok**: con valor **true** indica que la petición se resolvió sin problemas, es decir, el código de respuesta está entre 200-299.
- **redirected**: con valor **true** indica que la respuesta es el resultado de una redirección.
- **url**: devuelve la URL de la que procede la respuesta.
- **type**: indica el tipo de respuesta de la petición cuyos valores pueden ser *basic*, *cors*, *error* o *opaque*.

Además, el objeto de la respuesta tiene varios métodos:

- **redirect(url)**: método que redirige la respuesta a otro URL.
- **clone()**: clona la respuesta en otro objeto.
- **error()**: clona la respuesta generando un error de red.
- **text()**: obtiene un flujo de texto de la respuesta.
- **json()**: obtiene un flujo de texto de la respuesta pero en formato JSON.
- **blob()**: obtiene un objeto binario de la respuesta, como por ejemplo ficheros o imágenes.

Personalizar el objeto `Request`

Cuando se realizan peticiones mediante el método `fetch()`, además de la URL, podemos pasar como parámetro un objeto de tipo `Request`.

Este objeto permite modificar el paquete http que se envía con la petición para que se ajuste a las necesidades del cliente.

Algunas de las propiedades disponibles en el objeto `Request`:

- **url**: propiedad obligatoria que contiene la URL destino de la petición
- **method**: metodo http que usará la petición como por ejemplo **GET**, **POST**, **PUT**, **DELETE**, etcétera… Por defecto se utiliza **GET**.
- **headers**: permite indicar un objeto de tipo `Headers` que permite modificar la cabecera de la petición.
- **mode**: indica si se usa CORS en la petición.
- **cache**: indica el modo de caché de la respuesta.
- **redirect**: indica cómo se deben procesar las respuestas en caso de que procedan de redirecciones.
- **credentials**: permite especificar si se admiten cookies en la petición.
- **integrity**: almacena una cadena de integridad creada con un algoritmo hash para validar la integridad de la petición.

```
// URL de la API ficticia
const apiUrl = 'https://api.example.com/data';

// Opciones para la solicitud
const requestOptions = {
  method: 'GET',
  headers: new Headers({
    'Content-Type': 'application/json',
    'Authorization': 'Bearer YOUR_ACCESS_TOKEN', // Token de autorización si es necesario
    // Puedes agregar más encabezados según sea necesario
  }),
  // Agregar otras opciones según sea necesario
};

// Crear un objeto Request con la URL y las opciones
const request = new Request(apiUrl, requestOptions);

// Realizar la solicitud utilizando fetch()
fetch(request)
  .then(response => {
    // Verificar si la respuesta es exitosa (código de estado 200-299)
    if (!response.ok) {
      throw new Error(`Error en la solicitud: ${response.statusText}`);
    }
    // Parsear la respuesta JSON
    return response.json();
  })
  .then(data => {
    // Hacer algo con los datos recibidos
    console.log('Datos recibidos:', data);
  })
  .catch(error => {
    console.error('Error en la solicitud:', error);
  });
```

Uso de `async/await` con `fetch()`

Puesto que la API `fetch` es compatible con las **promesas**, es posible manejarla mediante la notación `async/await`:

```
// Función asincrónica para realizar la solicitud
async function fetchData() {
  // URL de la API ficticia
  const apiUrl = 'https://api.example.com/data';

  try {
    // Realizar la solicitud utilizando fetch() con await
    const response = await fetch(apiUrl);

    // Verificar si la respuesta es exitosa (código de estado 200-299)
    if (!response.ok) {
      throw new Error(`Error en la solicitud: ${response.statusText}`);
    }

    // Parsear la respuesta JSON con await
    const data = await response.json();

    // Hacer algo con los datos recibidos
    console.log('Datos recibidos:', data);
  } catch (error) {
    // Capturar y manejar cualquier error que pueda ocurrir durante la solicitud
    console.error('Error en la solicitud:', error.message);
  }
}

// Llamar a la función para iniciar la solicitud
fetchData();
```

Envío de datos con la petición

En muchas ocasiones, los servidores de Internet que otorgan servicios, requieren o permiten enviar datos para determinar de una forma más ajustada los datos que debe devolver.

De forma clásica esos datos son pares nombre/valor pero actualmente se admite enviar datos incluso en formato de texto JSON. Por tanto hay que conocer la API del servicio final para saber qué método usar, cómo enviar los datos y cómo debermos recibirlos.

Actualmente los comandos http se asocian a verbos que requieren un tipo concreto de acción:

- **GET** se asocia a una petición de datos.
- **POST** se asocia a un envío de datos a un servidor.
- **PUT** se asocia a una modificación de datos.
- **DELETE** se asocia a un borrado de datos.
- **PATCH** se asocia a una modificación de datos parcial.
- **HEAD** se asocia a una petición de cabecera.

Envío de datos usando `GET`

Cuando se realiza una solicitud HTTP con el método **GET**, los datos se envían en la URL como parámetros de consulta (*query parameters*). Estos parámetros son pares clave-valor que se concatenan a la URL después del signo de interrogación ('?') y se separan entre sí por el símbolo de ampersand ('&'):

```
https://api.example.com/data?param1=valor1&param2=valor2
```

Es importante señalar que como los parámetros de consulta se envían directamente en la URL, son visibles en la barra de direcciones del navegador y pueden quedar registrados en los registros del servidor.

Envío de datos usando `POST`

En este caso los datos no se envían en la URL sino que se envían en el cuerpo del propio paquete http.

En el caso clásico los datos proceden de datos de un formulario e incluso aunque no procedan, se debe marcar el paquete para indicar que los datos se envían al estilo de los formularios. Esto es, indicar el `Content-Type': 'application/x-www-form-urlencoded'`.

Esta forma usa un formato similar a **GET**. En las peticiones AJAX se debe asociar los valores que se envían a la propiedad `body`:

```
// URL de la API ficticia
const apiUrl = 'https://api.example.com/data';

// Datos a enviar en el cuerpo de la solicitud en formato x-www-form-urlencoded
const formData = new URLSearchParams();
formData.append('param1', 'valor1');
formData.append('param2', 'valor2');

// Opciones para la solicitud
const requestOptions = {
  method: 'POST',
  headers: new Headers({
    'Content-Type': 'application/x-www-form-urlencoded',
    'Authorization': 'Bearer YOUR_ACCESS_TOKEN' // Token de autorización si es necesario
    // Puedes agregar más encabezados según sea necesario
  }),
  // Convertir los datos a formato x-www-form-urlencoded y enviar en el cuerpo
  body: formData.toString()
  // Agregar otras opciones según sea necesario
};

// Realizar la solicitud utilizando fetch()
fetch(apiUrl, requestOptions)
  .then(response => {
    // Verificar si la respuesta es exitosa (código de estado 200-299)
    if (!response.ok) {
      throw new Error(`Error en la solicitud: ${response.statusText}`);
    }
    // Parsear la respuesta JSON
    return response.json();
  })
  .then(data => {
    // Hacer algo con los datos recibidos
    console.log('Datos recibidos:', data);
  })
  .catch(error => {
    console.error('Error en la solicitud:', error);
  });
```

Hay otra forma de enviar datos de formulario que es el que se asocia al tipo MIME: `application/multipart/form-data`. Es un estilo pensado para enviar datos de gran tamaño o de tipo binario. Si hay que enviar un archivo, este sería el formato apropiado:

```
// URL de la API ficticia
const apiUrl = 'https://api.example.com/upload';
```

```
// Crear un formulario HTML
const formData = new FormData();
formData.append('param1', 'valor1');
formData.append('param2', 'valor2');

// Puedes agregar archivos al formulario, si es necesario
// formData.append('file', fileInput.files[0]);

// Opciones para la solicitud
const requestOptions = {
  method: 'POST',
  headers: new Headers({
    'Authorization': 'Bearer YOUR_ACCESS_TOKEN' // Token de autorización si es necesario
    // No es necesario establecer Content-Type aquí, ya que FormData lo manejará automáticamente
    // Puedes agregar más encabezados según sea necesario
  }),
  body: formData
  // Agregar otras opciones según sea necesario
};

// Realizar la solicitud utilizando fetch()
fetch(apiUrl, requestOptions)
  .then(response => {
    // Verificar si la respuesta es exitosa (código de estado 200-299)
    if (!response.ok) {
      throw new Error(`Error en la solicitud: ${response.statusText}`);
    }
    // Parsear la respuesta JSON
    return response.json();
  })
  .then(data => {
    // Hacer algo con los datos recibidos
    console.log('Datos recibidos:', data);
  })
  .catch(error => {
    console.error('Error en la solicitud:', error);
  });
```

`FormData` se utiliza para construir un objeto que contiene pares clave-valor para cada campo del formulario, incluidos los campos de archivo si es necesario. `fetch()` manejará automáticamente el encabezado `Content-Type` para el tipo de contenido `multipart/form-data`.

Envío de datos JSON

No siempre el servicio destino de la petición usa los datos en el formato clásico. Hay servicios que aceptan los datos en formato JSON.

Para esas situaciones, se preparan los datos en un objeto y después se convierten a formato JSON con el método `JSON.stringfy(data)`:

```
// URL de la API ficticia
const apiUrl = 'https://api.example.com/data';

// Datos a enviar en el cuerpo de la solicitud en formato JSON
const jsonData = {
  param1: 'valor1',
  param2: 'valor2'
};

// Opciones para la solicitud
const requestOptions = {
  method: 'POST',
  headers: new Headers({
    'Content-Type': 'application/json',
    'Authorization': 'Bearer YOUR_ACCESS_TOKEN' // Token de autorización si es necesario
    // Puedes agregar más encabezados según sea necesario
```

```
  }),
  body: JSON.stringify(jsonData) // Convertir los datos a formato JSON y enviar en el cuerpo
  // Agregar otras opciones según sea necesario
};

// Realizar la solicitud utilizando fetch()
fetch(apiUrl, requestOptions)
  .then(response => {
    // Verificar si la respuesta es exitosa (código de estado 200-299)
    if (!response.ok) {
      throw new Error(`Error en la solicitud: ${response.statusText}`);
    }
    // Parsear la respuesta JSON
    return response.json();
  })
  .then(data => {
    // Hacer algo con los datos recibidos
    console.log('Datos recibidos:', data);
  })
  .catch(error => {
    console.error('Error en la solicitud:', error);
  });
```

Resumen

```
// 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✕10¹⁵.
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();
    }
}
```

Referencias

- https://developer.mozilla.org/es/docs/Web/JavaScript
- https://github.com/sorrycc/awesome-javascript
- https://cheatsheets.zip/javascript
- https://exercism.org/tracks/javascript
- https://jsfiddle.net/

Licencia

(cc) BY-SA Esta obra está bajo una licencia de Creative Commons Reconocimiento-Compartir Igual 4.0 Internacional.