

Tema 2. Javascript ES6+. Volumen 1.

Fundamentos y Nuevas Características en JavaScript ES6.
Volumen 1.

Luis José Molina Garzón

Agosto 2024



"JavaScript es un lenguaje con un gran poder. Pero con un gran poder viene una gran responsabilidad."

– **John Resig (creador de jQuery)**

© 2024 [Luis Molina Garzón](#). Todos los derechos reservados.

Estos apuntes no puede ser reproducido, distribuido ni transmitido de ninguna forma, ni por ningún medio, sin el permiso previo por escrito del autor, excepto en el caso de breves citas incorporadas en reseñas o artículos críticos.

Todos los nombres de productos, marcas comerciales y compañías mencionadas en estos apuntes son propiedad de sus respectivos dueños. Las marcas y nombres comerciales utilizados en estos apuntes son únicamente con fines de identificación y referencia, y no implican aprobación de los productos o servicios de esas marcas por parte del autor.

Estos apuntes han sido realizados con la asistencia de [gpt-4o](#) y [Claude 3.5 Sonnet](#).

IES Virgen del Carmen - Jaén.

Desarrollo de Aplicaciones Multiplataforma.

Desarrollo de interfaces.

lmolgar288@g.educaand.es

Índice

1 Introducción a ECMAScript y JavaScript	6
1.1 Soporte de Navegadores y Compatibilidad	6
1.2 Transpilación de Código	7
1.3 Ejecución de JavaScript	8
1.3.1 En el Navegador Web	8
1.3.2 Depuración en el navegador	9
1.3.3 Depurar Código Asíncrono	14
1.3.4 Node.js	15
1.3.5 Recursos Adicionales	16
1.4 Introducción a Babel	17
1.4.1 Uso de Babel	17
1.4.2 @babel/standalone	17
1.5 NPM (Node Package Manager)	18
1.5.1 Ejemplo de un servidor web con Nodejs y Express:	19
1.5.2 Ejemplo de una aplicación react	21
1.6 Empaquetado de Proyectos	24
1.6.1 Parcel: Un Empaquetador sin Configuración	24
1.7 Linter	26
1.7.1 ESLint	26
1.7.2 Proyecto con ESLint	26
2 Comentarios, literales, identificadores	28
2.1 Comentarios en JavaScript	28
2.1.1 Comentario de una línea	29
2.1.2 Comentario de múltiples líneas	29
2.2 Literales en JavaScript	29
2.3 Identificadores en JavaScript	30
3 Palabras reservadas, unicode, punto y coma	31
3.1 Palabras reservadas	31
3.2 Unicode en JavaScript	32
3.3 Uso del Punto y Coma en JavaScript	32
4 Declaración de variables	33
4.1 Declaración de Variables con var	33
4.2 Hoisting	34

4.3 Declaración de Variables con let y const (ES6)	34
4.4 Declaración con const	35
4.5 Tipado Dinámico en JavaScript	35
5 Tipos de datos en Javascript	36
5.1 Tipos de Datos Primitivos	36
5.1.1 Métodos y Propiedades en Tipos Primitivos	36
5.1.2 Undefined	37
5.1.3 Null	37
5.1.4 Boolean	38
5.1.5 Number	39
5.1.6 String	40
5.1.7 Symbol	50
5.2 Tipos de Datos de Objeto	52
5.2.1 Objetos (Object)	52
5.2.2 Arrays (Array)	52
5.2.3 Funciones (Function)	52
5.2.4 Mapas (Map)	53
5.2.5 Sets (Set)	53
5.3 Tipos de Datos Especiales	53
5.3.1 BigInt	53
5.3.2 TypedArray	54
5.4 El objeto global en Javascript	54
5.4.1 En entornos de navegador:	54
5.4.2 En Node.js:	54
5.4.3 Acceso a objetos globales:	54
5.4.4 Propiedades y métodos comunes:	55
5.4.5 Módulos y el objeto global:	56
5.4.6 El uso de “globalThis”:	56
5.5 Coerción	56
5.5.1 Peligros y Consideraciones	58
5.5.2 Ejemplos problemáticos	58
6 Estructuras condicionales: if, switch, ternario	58
6.1 Estructura if, else if, else	59
6.2 Evaluación implícita de valores a booleanos	59
6.3 Comprobación de inicialización de variables	60

6.4 Uso de operadores lógicos	60
6.4.1 Operador lógico AND (&&)	60
6.4.2 Operador lógico OR ()	60
6.5 Estructura switch	61
6.6 Operador ternario	61
7 Bucles	62
7.1 for	62
7.2 while	62
7.3 do...while	62
7.4 for...in	63
7.5 for...of	63
7.6 forEach	63
8 Objetos Javascript	63
8.1 Herencia y Prototipos	64
8.2 Creación de Objetos	65
8.3 Propiedades y Configuración	66
8.3.1 Atributos de las Propiedades	66
8.3.2 Atributos get y set	67
8.3.3 Métodos Object.defineProperties y Object.getOwnPropertyDescriptor	67
8.4 Acceso y Modificación de Propiedades	68
8.4.1 Ejemplos Avanzados con Herencia de Prototipos:	68
8.4.2 Modificación de Propiedades:	69
8.4.3 Acceso a Propiedades Inexistentes:	69
8.4.4 Operador Opcional Encadenado (? .):	70
8.4.5 Eliminación de Propiedades:	70
8.4.6 Comprobación de la Existencia de Propiedades:	70
8.5 Enumeración de Propiedades	71
8.5.1 Propiedades No Enumerable	71
8.5.2 Otras Opciones para Enumerar Propiedades	72
8.6 Extensión y Clonación de Objetos	72
8.7 Propiedades Abreviadas, Computadas y Símbolos	74
8.8 Operador spread en objetos	75
8.9 Métodos en Objetos	76
9 Arrays	77
9.1 Creación de Arrays	78
9.2 Operador Spread (...)	79

9.3 Leer y escribir elementos	79
9.4 Arrays Multidimensionales: Matriz Dispersa de 10x10	82
9.5 Métodos iteradores de Array	82
9.6 Otros Métodos Útiles de Arrays	84
10 Desestructuración	85
10.1 Desestructuración de Arrays	85
10.1.1 Características clave:	85
10.2 Desestructuración de Objetos	86
10.2.1 Características clave:	86
10.3 Aplicaciones Prácticas	87
11 Funciones	88
11.1 Hoisting en Funciones	88
11.2 Funciones como Expresiones	88
11.3 Funciones Anónimas y Recursividad	89
11.4 Funciones Arrow (Funciones Flecha)	89
11.4.1 Consideraciones sobre this en Funciones Arrow	90
11.4.2 Parámetros en Funciones	90
11.5 Objeto arguments	91
11.6 Operador Spread	91
11.7 Desestructuración de Argumentos	92
12 Ejercicios I de strings	92
13 Ejercicios II sobre desestructuración	93
14 Ejercicios III sobre arrays	94
15 Ejercicios IV sobre lógica de programación	95
16 Ejercicios V de Javascript	96

1 Introducción a ECMAScript y JavaScript

El estándar que define el lenguaje **JavaScript** se denomina **ECMAScript**. La primera versión de este estándar fue lanzada en 1997, marcando el inicio de un lenguaje que ha evolucionado significativamente desde entonces.

La versión 6 de ECMAScript, conocida también como **ECMAScript 2015 (ES2015)**, supuso una mejora significativa en el lenguaje. Entre las novedades más destacadas se incluyen:

- **Clases:** Introducción de la sintaxis para la creación de clases con `class`.
- **Módulos:** Se introdujeron los módulos ES6 con `import` y `export`.
- **Bucles for ... of:** Nueva forma de iterar sobre elementos de un iterable.
- **Funciones Arrow:** Sintaxis más concisa para definir funciones con `() => {}`.
- **Promesas:** Manejo asíncrono de código a través de promesas (`Promise`).
- Otras mejoras como `let`, `const`, `destructuring`, `spread`, etc.

La última especificación oficial, la **versión 13 de ECMAScript**, fue desarrollada en junio de 2022. Esta versión continuó con la evolución del lenguaje, incorporando nuevas funcionalidades y mejoras de rendimiento.

1.1 Soporte de Navegadores y Compatibilidad

Históricamente, los navegadores web han implementado las versiones de JavaScript con pequeñas diferencias, lo que solía generar problemas para desarrollar scripts compatibles. Los desarrolladores debían detectar el tipo de navegador y programar variantes específicas para cada uno. Afortunadamente, este problema ha disminuido gracias a la estandarización y la evolución de los motores de ejecución de JavaScript.

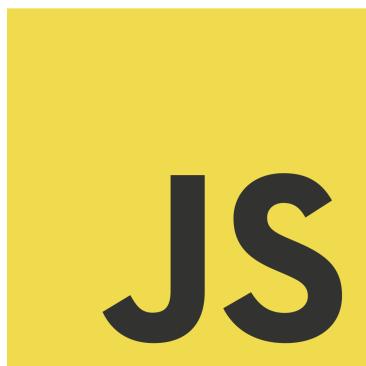


Figura 1: Logotipo de JavaScript

Para verificar el soporte que ofrecen los distintos navegadores a las versiones de ECMAScript (ES5, ES6, etc.), puedes consultar las siguientes páginas:

- [Can I use](#)
- [Compatibilidad ES6](#)

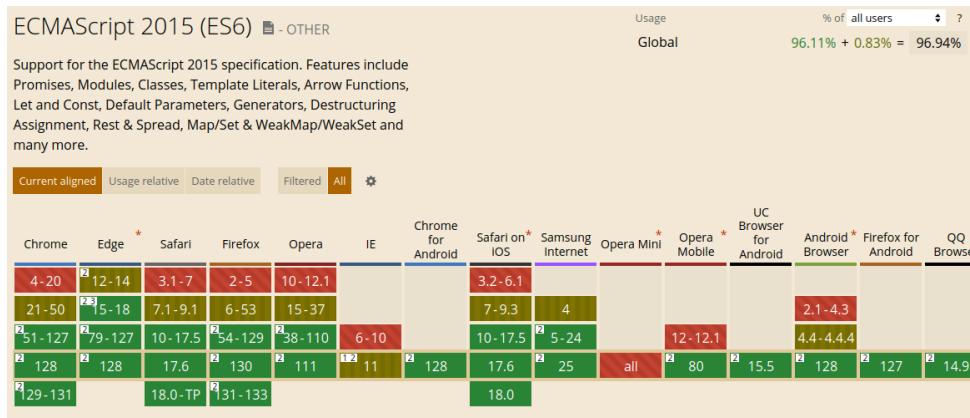


Figura 2: Soporte ES6

1.2 Transpilación de Código

Aunque el soporte de ES6 está muy avanzado, la **transpilación** sigue siendo una práctica común. La **transpilación** consiste en la traducción de código escrito en un lenguaje de alto nivel a otro del mismo nivel de abstracción. Por ejemplo:

- **De JavaScript ES6 a ES5:** Para asegurar una mayor compatibilidad con todos los navegadores.
- **De Java a Kotlin:** Para aprovechar las ventajas de Kotlin sobre Java.

Por otro lado, compilar se refiere a la traducción de un lenguaje de alto nivel a un nivel más bajo, como:

- **De código Java a Bytecode (.class)**
- **De código C a Código máquina**

Hasta hace poco, el soporte de ES6 en los navegadores web no era completo, lo que hacía necesario transpilar el código ES6 a ES5 al desarrollar aplicaciones con frameworks como React o Angular. Esto se hacía para garantizar la compatibilidad con una gama más amplia de navegadores, especialmente aquellos más antiguos como Internet Explorer, que no soportaban las nuevas características de ECMAScript 2015 (ES6). Sin embargo, hoy en día, la mayoría de los navegadores modernos han adoptado completamente ES6 y versiones posteriores, lo que reduce la necesidad de transpilar el código a ES5.

1.3 Ejecución de JavaScript

1.3.1 En el Navegador Web

El código JavaScript se puede ejecutar directamente en un navegador web. Existen varias formas de incluir código JavaScript en una página:

- **Código interno:** Usando la etiqueta `<script>` dentro del documento HTML. Crea un documento `index.html` con el siguiente contenido. Usa las herramientas para desarrolladores, tecla **F12**, y accede a la pestaña **Consola**.

`index.html`

```
<html>
  <head>
    <title>Ejecutando Javascript en el navegador</title>
  </head>
  <body>
    <script type="text/javascript">
      console.log("Hola Mundo.");
      // Código JavaScript aquí
    </script>
  </body>
</html>
```

Una página web se puede visualizar en el navegador de muchas formas:

- Como archivo HTML en el ordenador. En el navegador usa la URL `file:///home/usuario/proyecto/index.html` o `file:///C:/Users/usuario/proyecto/index.html`.
- Como archivo HTML en un servidor web. Puedes usar una extensión (plugin) útil para visual studio code como **Live Server** para visualizar la página web en el navegador. Instala la extensión **Live Server** desde el marketplace de vscode y pulsa **F1** y selecciona **Live Server: Open with Live Server**.
- **Código externo:** Referenciando un archivo JavaScript externo.

`index.html`

```
<html>
  <head>
    <title>Ejecutando Javascript en el navegador</title>
  </head>
  <body>
    <script src="js/programa.js" type="text/javascript"></script>
  </body>
</html>
```

`js/programa.js`

```
console.log("Hola mundo desde un fichero externo");
```

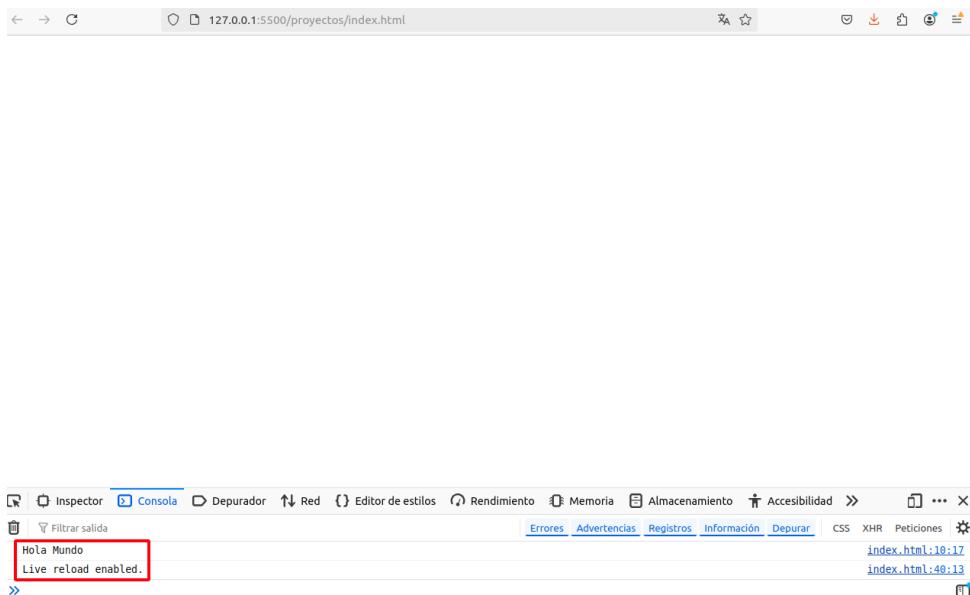


Figura 3: Ejecutando Javascript en el navegador

Generalmente, los scripts se colocan al final de la etiqueta <body>, aunque si se trata de librerías que no se ejecutan hasta que ocurre un evento, pueden situarse dentro del <head>. El motivo de situar los scripts al final del <body> es que el navegador debe cargar primero el contenido del <body> y luego el de los <script> para que el código JavaScript pueda acceder a los elementos cargados en la página web (DOM).

La **consola del navegador** es otra herramienta útil que permite ejecutar código JavaScript directamente en el navegador.

1.3.2 Depuración en el navegador

Los navegadores modernos, como Google Chrome, Firefox, Edge y Safari, incluyen **herramientas de desarrollo (Developer Tools)** que facilitan la **depuración** de código JavaScript. A continuación, se explica cómo depurar en el navegador, acompañando la explicación con ejemplos prácticos.

1.3.2.1 Acceder a las Herramientas de Desarrollo En la mayoría de los navegadores, puedes acceder a las herramientas de desarrollo pulsando **F12** o **Ctrl+Shift+I** (Windows/Linux) o **Cmd+Option+I** (Mac).

1.3.2.2 Usar la Consola (Console) La consola es una herramienta poderosa para visualizar mensajes, errores y para interactuar directamente con el entorno de JavaScript.

Ejemplo con console.log:

```
<!DOCTYPE html>
<html>
<head>
    <title>Ejemplo de Depuración</title>
    <script>
        function saludar(nombre) {
            console.log("Función saludar iniciada");
            let mensaje = "Hola, " + nombre;
            console.log("Mensaje creado:", mensaje);
            return mensaje;
        }
        saludar("Mundo");
    </script>
</head>
<body>
    <h1>Revisa la consola para ver los mensajes</h1>
</body>
</html>
```

Pasos:

1. Abre el archivo HTML en tu navegador.
2. Abre las herramientas de desarrollo (Ctrl + Shift + I).
3. Navega a la pestaña Console.
4. Verás los mensajes:

```
Función saludar iniciada
Mensaje creado: Hola, Mundo
```

1.3.2.3 Establecer Puntos de Interrupción (Breakpoints) Los breakpoints permiten pausar la ejecución del código en una línea específica para inspeccionar el estado de la aplicación.

Ejemplo:

```
<!DOCTYPE html>
<html>
<head>
    <title>Depuración con Breakpoints</title>
    <script>
        function calcularSuma(a, b) {
            let suma = a + b;
            return suma;
        }

        let resultado = calcularSuma(5, 10);
        console.log("El resultado es:", resultado);
    </script>
</head>
<body>
    <h1>Depura el código JavaScript</h1>
</body>
</html>
```

Pasos:

1. Abre el archivo HTML en tu navegador.
2. Abre las herramientas de desarrollo.
3. Ve a la pestaña Sources (Google Chrome) o Depurador (Firefox).
4. Navega al archivo JavaScript (en este caso, está embebido en el HTML).
5. Haz clic en el número de línea donde deseas establecer el breakpoint (por ejemplo, en la línea `let suma = a + b;`).
6. Recarga la página. La ejecución se pausará en el breakpoint.
7. Ahora puedes inspeccionar variables, el call stack y más.

1.3.2.4 Usar la Declaración debugger La palabra clave debugger detiene la ejecución del código en el punto donde se inserta, siempre que las herramientas de desarrollo estén abiertas. Ejemplo:

```
<!DOCTYPE html>
<html>
<head>
    <title>Usar Debugger</title>
    <script>
        function multiplicar(a, b) {
            let producto = a * b;
            debugger; // La ejecución se pausará aquí
            return producto;
        }

        let resultado = multiplicar(4, 5);
        console.log("El producto es:", resultado);
    </script>
</head>
<body>
    <h1>Depuración con la declaración debugger</h1>
</body>
</html>
```

Pasos:

1. Abre el archivo HTML en tu navegador.
2. Abre las herramientas de desarrollo.
3. Recarga la página.
4. La ejecución se pausará en la línea con `debugger;`.
5. Puedes inspeccionar las variables `a`, `b` y `producto`, y avanzar paso a paso.

1.3.2.5 Inspeccionar Variables y el Call Stack Cuando la ejecución está pausada (ya sea por un breakpoint o debugger), puedes inspeccionar:

- **Variables Locales y Globales:** Observa los valores actuales de las variables.
- **Call Stack:** Verifica cómo se llegó al punto actual en la ejecución.
- **Scope:** Examina el ámbito de las variables en diferentes niveles (local, closure, global).

Ejemplo:

Utilizando el ejemplo anterior con la función multiplicar, cuando la ejecución se pausa en `debugger` ;:

1. Variables:

- `a` tiene el valor 4.
- `b` tiene el valor 5.
- `producto` tiene el valor 20.

2. Call Stack:

- Muestra la función actual y cómo se llamó (en este caso, directamente desde el script principal).

3. Scope:

- Puedes ver variables locales dentro de multiplicar y variables globales como `resultado`.

1.3.2.6 Pasar y Avanzar en la Ejecución (Stepping Through) Mientras la ejecución está pausada, puedes controlar cómo avanzar:

- Continuar (Continue): Reanuda la ejecución hasta el siguiente breakpoint.
- Paso a Paso (Step Over): Ejecuta la siguiente línea de código sin entrar en funciones llamadas.
- Paso Dentro (Step Into): Entra dentro de la función llamada en la línea actual.
- Paso Fuera (Step Out): Sale de la función actual y vuelve al contexto anterior.

Ejemplo:

Usando el ejemplo de calcularSuma:

- Establece un breakpoint en `let suma = a + b;`.
- Cuando la ejecución se pausa, usa **Step Over** para ejecutar la línea y pasar a `return suma` .
- Usa **Step Into** si hay una función llamada dentro de `calcularSuma` y quieras depurarla.
- Usa **Continue** para reanudar la ejecución hasta el siguiente breakpoint o hasta el final.

1.3.2.7 Usar Puntos de Interrupción Condicionales Puedes establecer breakpoints que solo se activen cuando se cumpla una cierta condición, lo que es útil para bucles o casos específicos. Ejemplo:

```
<!DOCTYPE html>
<html>
<head>
  <title>Breakpoint Condicional</title>
  <script>
```

```

        for (let i = 0; i < 10; i++) {
            console.log("Iteración:", i);
        }
    </script>
</head>
<body>
    <h1>Depuración Condicional</h1>
</body>
</html>

```

Pasos:

- Abre las herramientas de desarrollo y ve a la pestaña Sources (Google Chrome) o Depurador (Firefox).
- Establece un breakpoint en la línea `console.log("Iteración:", i);`.
- Haz clic derecho en el breakpoint y selecciona **Edit breakpoint** o **Agregar condición**.
- Ingresa una condición, por ejemplo: `i === 5`.
- Recarga la página. La ejecución se pausará solo cuando `i` sea 5.

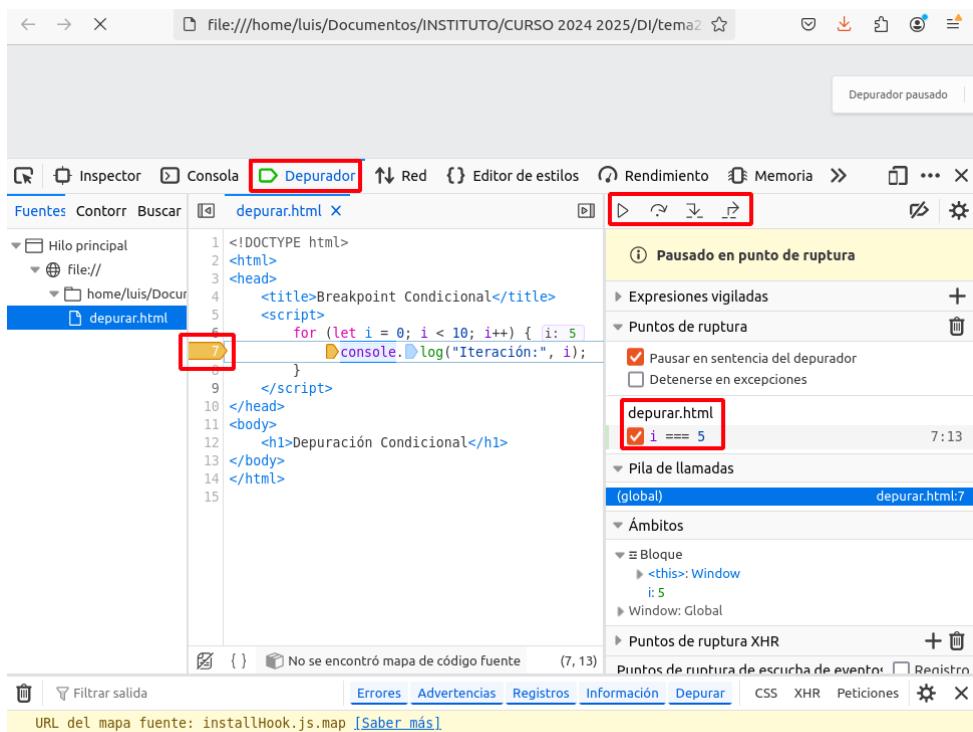


Figura 4: Depuración Condicional en las herramientas de desarrollador de Mozilla Firefox

1.3.2.8 Utilizar la Pestaña de Network para Depurar Solicitudes La pestaña **Network** (Red) permite ver todas las solicitudes de red realizadas por la página, lo que es útil para depurar API calls, cargar recursos, etc.

Ejemplo:

Supongamos que tienes una llamada fetch en tu código:

```
fetch('https://api.example.com/data')
  .then(response => response.json())
  .then(data => {
    console.log("Datos recibidos:", data);
  })
  .catch(error => {
    console.error("Error:", error);
});
```

Pasos para Depurar:

- Abre las herramientas de desarrollo y ve a la pestaña **Network**.
- Recarga la página o ejecuta la función que hace la solicitud `fetch`.
- Observa la solicitud en la lista de la pestaña **Network**.
- Haz clic en la solicitud para ver detalles como encabezados, respuesta, tiempo de carga, etc.
- Si hay errores, aparecerán resaltados en rojo y podrás inspeccionarlos.

1.3.3 Depurar Código Asíncrono

Depurar promesas, `async/await` y callbacks puede ser más complejo. Las herramientas de desarrollo modernas facilitan esto permitiendo pausas en puntos específicos.

Ejemplo con `async/await`:

```
async function obtenerDatos() {
  try {
    let respuesta = await fetch('https://api.example.com/data');
    let datos = await respuesta.json();
    console.log("Datos obtenidos:", datos);
  } catch (error) {
    console.error("Error al obtener datos:", error);
  }
}

obtenerDatos();
```

Pasos para Depurar:

- Establece breakpoints en las líneas con `await`.
- Cuando la ejecución se pause, puedes inspeccionar el estado de las variables antes y después de las llamadas asíncronas.
- Utiliza **Step Into** para entrar en funciones asíncronas si es necesario.



Algunos conceptos como la programación asíncrona, promesas, callbacks, etc. se verán con más detalle en el siguiente volumen de este curso.

Repasa este apartado cuando tengas que depurar código JavaScript.

1.3.4 Node.js

Node.js (comúnmente abreviado como Node) es un entorno de ejecución de JavaScript basado en el **motor V8 de Chrome**, diseñado para ejecutar aplicaciones del lado del servidor, lo que lo convierte en una herramienta ideal para desarrollar backends y servicios web.



Figura 5: Logotipo de Node.js

Además de su uso en el desarrollo del lado del servidor, Node.js es una dependencia fundamental para el ecosistema de desarrollo frontend, especialmente en frameworks y bibliotecas como [React](#) y [Angular](#). Esto se debe a que muchas herramientas clave para la construcción, transpilación y empaquetado de aplicaciones frontend están escritas en JavaScript y se ejecutan en Node.js. Estas herramientas, como [Babel](#), [Webpack](#) y [Vite](#), permiten a los desarrolladores transformar su código moderno en versiones optimizadas que pueden ejecutarse eficientemente en navegadores web o dispositivos móviles. Así, Node.js juega un papel crucial tanto en el desarrollo backend como en el frontend, facilitando un flujo de trabajo integral para aplicaciones web modernas.



Figura 6: Motor V8 de Google. Usado en Chromium, Microsoft Edge, Brave, Opera, Chrome, etc.

Para gestionar diferentes versiones de Node.js en tu sistema, se recomienda usar **NVM (Node Version Manager)**. Esta herramienta permite instalar, desinstalar y cambiar entre versiones de Node.js de manera sencilla.

- Instalación de **NVM** en **linux**:

- Consulta el repositorio oficial: <https://github.com/nvm-sh/nvm>.
- Abre una terminal y ejecuta el siguiente comando:

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.40.0/install.sh | bash
```

- Reinicia la terminal y ejecuta el siguiente comando para verificar la instalación:

```
nvm --version
```

- Instalación en **Windows**:

- Utiliza el siguiente repositorio: <https://github.com/coreybutler/nvm-windows/releases> para descargar el archivo **nvm-setup.zip** de la última versión.
- Extrae el archivo ZIP y ejecuta el instalador **nvm-setup.exe**.
- Sigue las instrucciones del instalador para completar la instalación.
- Abre una nueva ventana de Command Prompt o PowerShell y verifica la instalación ejecutando:

```
nvm version
```

Algunos comandos útiles incluyen:

- **nvm list**: Ver las versiones de Node.js instaladas.
- **nvm install node**: Instala la última versión estable de Node.js.
- **nvm install lts/fermium**: Instala una versión específica de Node.js.
- **node -version**: Muestra la versión de Node.js en uso.
- **nvm use 18**: Cambia a la versión 18 de Node.js.
- **nvm use default**: Cambia a la versión por defecto de Node.js.

Cuando ejecutas el comando **node** en un terminal, accedes a un intérprete de JavaScript donde puedes probar código de manera interactiva. Sin embargo, dado que Node.js no se ejecuta en un navegador, no puede modificar el **DOM**, por lo que instrucciones como **document.write()** o **document.createElement("p")** no funcionarán.

Node.js utiliza el motor V8 de JavaScript (de Google), que es altamente eficiente y rápido, y ofrece soporte completo para ES6. También es posible ejecutar un archivo JavaScript directamente desde la línea de comandos usando:

```
node programa.js
```

1.3.5 Recursos Adicionales

Además de ejecutar código JavaScript localmente, existen aplicaciones online como [JSFiddle](#) que permiten programar y probar JavaScript de manera interactiva.

1.4 Introducción a Babel

En el desarrollo moderno de aplicaciones web, el uso de nuevas versiones de JavaScript, como ES6 y superiores, es común debido a las mejoras en la sintaxis y las nuevas funcionalidades que ofrecen. Sin embargo, no todos los entornos de ejecución (motores de JavaScript) son compatibles con las versiones más recientes del lenguaje. Esto significa que un programa escrito utilizando las últimas características de ES6+ (Es6 o versiones posteriores) podría no ejecutarse en algunos entornos, especialmente en navegadores más antiguos.



Figura 7: Logotipo de Babel

Aquí es donde entra en juego [Babel](#), un transpilador que permite compilar (transpilar, para ser más precisos) código escrito en ES6+ a versiones más antiguas de JavaScript, como ES5, que tienen un soporte más amplio en diferentes entornos. Babel también permite añadir **polyfills**, que son fragmentos de código que permiten que las nuevas características del lenguaje sean interpretadas correctamente por navegadores que no las soportan nativamente como CSS3, SVG, LocalStorage, etc.

Existen otros transpiladores como [SWC](#) (Speedy Web Compiler) y [esbuild](#). Cuando estudiemos React usaremos [create-react-app](#) para crear el esqueleto de una aplicación React. [create-react-app](#) usa Babel para transpilar. [Vite](#), otra aplicación para crear proyectos web, utiliza [esbuild](#) como transpilador.

Al trabajar con React, se utiliza una sintaxis especial llamada **JSX**, que es una extensión de la sintaxis de JavaScript. **JSX** también necesita ser transpilado a JavaScript para que pueda ser ejecutado en los navegadores, ya que los motores de JavaScript no soportan esta sintaxis. **Babel** facilita este proceso, permitiendo que el código JSX se convierta en código JavaScript compatible.

1.4.1 Uso de Babel

Babel se puede utilizar de diferentes maneras según las necesidades del proyecto. Aquí se detallan algunas de las formas más comunes de utilizar Babel:

1.4.2 @babel/standalone

Esta versión de Babel permite incrustar código ES6 en una página web y transpilarlo en línea antes de su ejecución. Aunque esta opción es útil para pruebas rápidas, no se recomienda en entornos de

producción debido a su ineficiencia, ya que el código se transpila cada vez que se refresca la página. En navegadores modernos, esta funcionalidad es innecesaria debido al soporte casi completo de ES6.

Ejemplo:

```
<html>
  <body>
    <script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
    <script type="text/babel">
      const hello = () => {
        alert("Hello, world!");
      };
      hello();
    </script>
  </body>
</html>
```

Vite y **create-react-app** dejan preparado el proyecto para la transpilación con esbuild y babel, respectivamente, con lo que no es necesario configurar la transpilación de forma manual.



Figura 8: Logotipo de Vite

1.5 NPM (Node Package Manager)

NPM (Node Package Manager) es el **gestor de paquetes** predeterminado para Node.js. Es una herramienta fundamental en el ecosistema de JavaScript, utilizada principalmente para gestionar las dependencias (librerías y módulos) que un proyecto de Node.js puede necesitar.



Figura 9: Logotipo de NPM

Funciones principales de NPM:

- **Instalación de paquetes:** NPM permite instalar librerías y módulos de terceros, facilitando el desarrollo de aplicaciones. Por ejemplo, `npm install express` instala el framework Express en tu proyecto.

- **Gestión de dependencias:** NPM mantiene un archivo llamado package.json que contiene un listado de todas las dependencias de un proyecto, así como sus versiones, scripts de comandos, y otra información relevante.
- **Publicación de paquetes:** Con NPM, los desarrolladores pueden publicar sus propios paquetes para compartirlos con la comunidad. Esto permite a otros desarrolladores usar tu código fácilmente en sus proyectos.
- **Versionado y actualización:** NPM permite manejar versiones de los paquetes, facilitando la actualización de dependencias de forma segura.

Yarn es otro gestor de paquetes para JavaScript, desarrollado por Facebook en colaboración con otros desarrolladores como Google y Tilde. Yarn se creó como una alternativa a NPM, con el objetivo de abordar algunas de las limitaciones que tenía NPM en sus primeras versiones.

1.5.1 Ejemplo de un servidor web con Nodejs y Express:

En este ejemplo vamos a crear una aplicación web simple usando **Express**, un popular framework de Node.js para explicar el uso de npm.

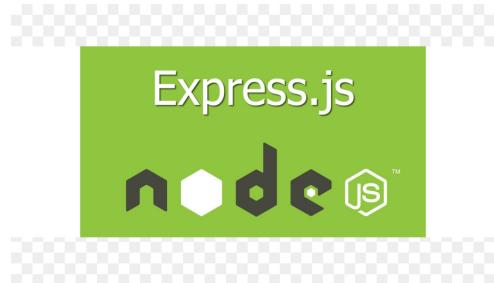


Figura 10: Logotipo de Express

1. Instalar Node.js (si no lo tienes instalado)

Primero, asegúrate de tener Node.js instalado en tu máquina. Node.js viene con NPM preinstalado.

Puedes verificar si ya lo tienes instalado usando los siguientes comandos en tu terminal:

```
node -v  
npm -v
```

2. Crear una carpeta para tu proyecto

Primero, crea una carpeta para tu proyecto y accede a ella desde la terminal:

```
mkdir mi-app  
cd mi-app
```

3. Inicializar un proyecto de Node.js

Para empezar, necesitas crear un archivo **package.json**, que almacenará la configuración de tu proyecto y la lista de dependencias (librerías que usa el proyecto). Usa el siguiente comando para inicializarlo:

```
npm init
```

Este comando te hará una serie de preguntas sobre tu proyecto, como el nombre, versión, descripción, etc. Si deseas aceptar los valores por defecto, simplemente presiona Enter para cada pregunta.

Al final, tendrás un archivo **package.json** en tu carpeta de proyecto.

4. Instalar Express como una dependencia

Ahora, puedes instalar **Express** (u otros paquetes que necesites) usando NPM. Para instalar Express, ejecuta:

```
npm install express
```

Este comando hará lo siguiente:

- Descargará el paquete express desde el registro de NPM.
- Guardará la información de la versión de Express dentro del archivo **package.json** bajo la sección **dependencies**.
- Creará una carpeta **node_modules** donde se descargarán y almacenarán todas las dependencias del proyecto.

5. Crear un archivo de servidor básico

Ahora, crea un archivo **index.js** que será el punto de entrada de tu aplicación:

index.js

```
const express = require('express');
const app = express();

// Ruta básica
app.get('/', (req, res) => {
  res.send('Hola Mundo');
});

// Iniciar el servidor en el puerto 3000
app.listen(3000, () => {
  console.log('Servidor escuchando en http://localhost:3000');
});
```

6. Ejecutar la aplicación

Para ejecutar tu aplicación, usa el siguiente comando:

```
node index.js
```

Esto iniciará el servidor en `http://localhost:3000`. Si abres un navegador y visitas esa dirección, deberías ver el mensaje “¡Hola Mundo!”.

7. Añadir scripts de NPM (Opcional)

En tu archivo `package.json`, puedes agregar scripts personalizados. Por ejemplo, puedes añadir un script para iniciar tu aplicación más fácilmente:

```
"scripts": {  
  "start": "node index.js"  
}
```

Ahora, puedes iniciar tu aplicación simplemente ejecutando:

```
npm start
```

8. Administrar dependencias (Opcional)

- Para actualizar una dependencia: Usa `npm update nombre_del_paquete`.
- Para eliminar una dependencia: Usa `npm uninstall nombre_del_paquete`.
- Para instalar todas las dependencias listadas en `package.json`: Usa `npm install` (esto es útil cuando clonas un proyecto y necesitas instalar todas sus dependencias).

9. Fichero .gitignore

Conforme un proyecto crece, el tamaño de la carpeta `node_modules` puede llegar a ser muy grande. Es recomendable añadir un fichero `.gitignore` en la raíz del proyecto para que Git ignore la carpeta `node_modules` y no la incluya en los commits.

```
.gitignore
```

```
node_modules
```

1.5.2 Ejemplo de una aplicación react

Ahora vamos a usar `npm`, `node` y `babel` para crear una pequeña aplicación en `React`.

1. Crear la estructura del proyecto

Primero, crea la carpeta para tu proyecto y la estructura básica de directorios y archivos:

```
mkdir mi-proyecto-babel  
cd mi-proyecto-babel  
mkdir src dist  
touch src/index.jsx dist/index.html
```

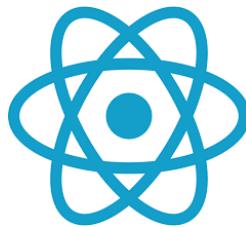


Figura 11: Logotipo de React

La estructura del proyecto será:

```
mi-proyecto-babel/
  src/
    index.jsx
  dist/
    index.html
  package.json
```

2. Inicializar el proyecto con NPM

Inicializa el proyecto para crear un archivo **package.json**:

```
npm init -y
```

El comando `-y` acepta todos los valores predeterminados para simplificar el proceso.

3. Instalar Babel y plugins

Ahora, instala **Babel** y los plugins necesarios para transpilar **JSX** y **ES6+**:

```
npm install --save-dev @babel/core @babel/cli @babel/preset-env @babel/preset-react
```

Explicación de los paquetes:

- La opción `--save-dev`, indica que se trata de una dependencia de desarrollo. Es decir este paquete es necesario mientras estamos programando pero no es necesario en el código final.
- `@babel/core`: El núcleo de Babel, que realiza la transformación de código.
- `@babel/cli`: Una interfaz de línea de comandos para Babel, para que puedas ejecutarlo desde la terminal.
- `@babel/preset-env`: Un conjunto de reglas que permiten a Babel transpilar ES6+ a ES5.
- `@babel/preset-react`: Un conjunto de reglas para transpilar JSX y otras características específicas de React.

4. Configurar Babel

Crea un archivo `.babelrc` en la raíz del proyecto para configurar Babel:

```
touch .babelrc
```

Y añade la siguiente configuración:

```
{  
  "presets": ["@babel/preset-env", "@babel/preset-react"]  
}
```

Esta configuración le dice a Babel que utilice los presets para ES6+ y JSX.

5. Escribir el código JSX

Ahora, escribe un simple componente en `src/index.jsx`:

`src/index.jsx`

```
import React from 'react';  
import ReactDOM from 'react-dom';  
  
const App = () => {  
  return (  
    <div>  
      <h1>Hola, Mundo desde React con Babel!</h1>  
    </div>  
  );  
};  
  
ReactDOM.render(<App />, document.getElementById('root'));
```

6. Configurar el archivo HTML

Crea un archivo `dist/index.html` que servirá como plantilla para tu aplicación:

```
<!-- dist/index.html -->  
<!DOCTYPE html>  
<html lang="en">  
<head>  
  <meta charset="UTF-8">  
  <meta name="viewport" content="width=device-width, initial-scale=1.0">  
  <title>Mi Proyecto con Babel</title>  
</head>  
<body>  
  <div id="root"></div>  
  <script src="bundle.js"></script>  
</body>  
</html>
```

Este archivo HTML incluye un div con el ID root, donde se montará tu aplicación React. El archivo `bundle.js` es donde se incluirá el JavaScript transpilado.

7. Transpilar JSX a JavaScript

Añade un script en el archivo `package.json` para transpilar tu código usando Babel:

```
"scripts": {  
  "build": "babel src -d dist --extensions \".js,.jsx\""  
}
```

Este script le dice a Babel que tome los archivos `.js` y `.jsx` de la carpeta `src` y los transpile en la carpeta `dist`.

Para ejecutar el script, usa:

```
npm run build
```

Esto generará un archivo `bundle.js` en la carpeta `dist` con el código transpilado.

8. Ejecutar la aplicación

Ahora, abre el archivo `dist/index.html` en un navegador. Deberías ver el texto “Hola, Mundo desde React con Babel!” en la página.

1.6 Empaquetado de Proyectos

Cuando un proyecto crece, puede tener cientos o miles de archivos JavaScript. Para mejorar la eficiencia de carga en los navegadores, es importante empaquetar el código en unos pocos archivos optimizados. El proceso de empaquetado generalmente incluye:

- **Transpilar:** Convertir el código a una versión compatible de JavaScript.
- **Minimizar:** Reducir el tamaño del código eliminando espacios, comentarios, y renombrando variables.
- **Ofuscar:** Hacer que el código sea más difícil de leer para proteger la propiedad intelectual.
- **Empaquetar:** Combinar múltiples archivos JavaScript en unos pocos archivos para reducir el número de peticiones HTTP.

1.6.1 Parcel: Un Empaquetador sin Configuración

Parcel es una herramienta de empaquetado que requiere cero configuración, ideal para desarrolladores que desean una solución simple y efectiva. A continuación se muestra cómo crear un proyecto básico usando Parcel:



Figura 12: Logotipo de Parcel

Iniciar el proyecto:

```
npm init -y
```

Instalar Parcel:

```
npm install -D parcel
```

Crear un archivo HTML y JavaScript:

index.html

```
<!doctype html>
<html>
  <head>
    <title>Parcel Demo</title>
  </head>
  <body>
    <script src=".js/index.js"></script>
  </body>
</html>
```

js/index.js

```
import component from './component';
document.body.appendChild(component('Hola mundo'));
```

js/component.js

```
const miTitulo = (msg) => {
  let title = document.createElement('H1');
  title.innerHTML = msg;
  return title;
};
export default miTitulo;
```

Crear scripts de npm:

En **package.json**, agrega los siguientes scripts:

```
"scripts": {
  "start": "parcel index.html",
  "build": "parcel build index.html"
}
```

Ejecutar el proyecto:

```
npm start
```

Esto inicia un servidor de desarrollo en <http://localhost:1234>.

Construir para producción:

```
npm run build
```

Parcel empaqueta, minimiza y optimiza el proyecto para despliegue en producción.

Al final del proceso, el código generado en la carpeta **dist** está listo para ser desplegado en un servidor web, asegurando una carga rápida y eficiente para los usuarios.

1.7 Linter

Un **linter** es una herramienta que analiza el código fuente de manera estática, es decir, sin ejecutarlo. Su principal función es detectar errores de sintaxis, problemas de formato y advertir sobre posibles malas prácticas. Además, los linters pueden sugerir mejoras en el código y ayudar a mantener un estilo de codificación consistente.

El uso de linters en el proceso de desarrollo ofrece múltiples ventajas, entre las que se incluyen:

- **Detección de errores temprana:** Identifica errores de sintaxis antes de ejecutar el código.
- **Mejora de la calidad del código:** Sugerencias para mejorar la legibilidad y mantener un estándar de codificación.
- **Consistencia:** Refuerza un estilo de codificación uniforme en todo el proyecto, esencial para equipos de desarrollo.
- **Facilita la revisión de código:** Reduce la necesidad de correcciones manuales durante la revisión.

1.7.1 ESLint

ESLint es uno de los linters más populares en el ecosistema JavaScript. Fue creado para proporcionar una herramienta extensible y altamente configurable que analiza el código en busca de problemas.

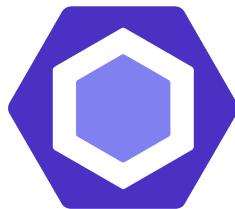


Figura 13: Logotipo de ESLint

- Detecta **errores de sintaxis**.
- Advierte sobre **malas prácticas** de programación.
- Ofrece **sugerencias de mejora** para el código.
- Ayuda a **mantener un estilo de codificación consistente**.
- Permite reforzar **reglas internas específicas** de un equipo de desarrollo.

1.7.2 Proyecto con ESLint

1. Proyecto NPM

Para comenzar, es necesario crear un proyecto de Node.js:

```
npm init -y
```

2. Instalar ESLint

Instala ESLint como una dependencia de desarrollo:

```
npm install --save-dev eslint
```

Nota: Es importante instalar ESLint como una dependencia de desarrollo, ya que no se necesita en producción.

3. Configurar ESLint

Inicia la configuración de ESLint ejecutando:

```
npx eslint --init
```

npx es una herramienta de línea de comandos que forma parte del ecosistema de Node.js y npm (Node Package Manager). Su propósito principal es permitir la ejecución de paquetes npm sin necesidad de instalarlos globalmente en el sistema o incluso localmente en el proyecto.

Aunque ESLint ya está instalado localmente en tu proyecto, usar npx es más cómodo. Alternativamente, podrías ejecutar ESLint directamente:

```
./node_modules/.bin/eslint --init
```

Otra opción es crear un script en el archivo package.json para simplificar la ejecución.

4. Ejecutar el Configurador de ESLint

También puedes iniciar la configuración con el siguiente comando:

```
npm init @eslint/config
```

El configurador de ESLint te guiará a través de una serie de preguntas para crear un archivo de configuración que se adapte a tus necesidades. Aquí te mostramos una configuración recomendada:

- ¿Para qué quieras usar ESLint?: To check syntax, find problems, and enforce code style.
- ¿Qué tipo de módulos usas?: JavaScript modules (import/export).
- ¿Usas un framework?: None.
- ¿Usas TypeScript?: No.
- ¿Dónde se ejecutará tu código?: Node.
- ¿Qué estilo de código te gustaría usar?: Use a popular style guide (Airbnb).
- Formato del archivo de configuración: JSON.
- ¿Quieres instalar las reglas de Airbnb?: Yes.

- ¿Qué gestor de paquetes prefieres usar?: npm.

Finalmente, ESLint creará un archivo `.eslintrc.json` con la configuración especificada.

4. Probando ESLint en tu Proyecto

Creación de un Archivo de Prueba

Vamos a crear un archivo `index.js` con algunos errores de sintaxis y estilo para probar ESLint:

```
function nombre__completo() {  
    return nombre + " " + apellidos;  
}  
  
var nombre = 'Luis';  
var apellidos = "Molina";  
  
console.log(nombre__completo());  
  
let personas = new Array(nombre__completo(), "Antonio Perez");  
console.log(personas[1]);  
console.log(personas[2]);
```

Ejecución de ESLint

Para ejecutar el linter y analizar tu archivo, usa el siguiente comando:

```
npx eslint *.js
```

Observa los errores y advertencias que muestra ESLint, y procede a solucionarlos.

Automatizando el Uso de ESLint

Si planeas usar ESLint frecuentemente, es recomendable añadir un script en el archivo `package.json`:

```
"scripts": {  
    "lint": "eslint . --ext .js"  
}
```

Ahora puedes ejecutar ESLint con:

```
npm run lint
```

2 Comentarios, literales, identificadores

2.1 Comentarios en JavaScript

Los **comentarios** en JavaScript funcionan de manera similar a los de Java. Son útiles para agregar notas, explicaciones o descripciones dentro del código, que no son ejecutadas por el intérprete. Los

comentarios ayudan a que el código sea más legible y comprensible tanto para el autor como para otros desarrolladores que lo lean en el futuro.

2.1.1 Comentario de una línea

Para comentar **una sola línea** en JavaScript, se utiliza `//`. Todo lo que siga a `//` en esa línea será ignorado por el intérprete.

Ejemplo:

```
// Esto es un comentario de una sola línea
let x = 5; // También se puede hacer aquí
```

En este caso, la primera línea es un comentario completo y la segunda línea tiene un comentario al final, explicando que se está asignando el valor 5 a la variable `x`.

2.1.2 Comentario de múltiples líneas

Para comentar **varias líneas** a la vez, se utiliza `/* */`. Todo el texto entre `/*` y `*/` será ignorado por el intérprete.

Ejemplo:

```
/*
Este es un comentario
de múltiples líneas
que puede abarcar varias
líneas de código
*/
let y = 10;
```

Este tipo de comentario es útil cuando necesitas explicar un bloque de código o dejar una nota más extensa.

2.2 Literales en JavaScript

Los literales son valores fijos que se escriben directamente en el código. No son variables, sino valores constantes que se asignan a variables o se utilizan directamente en las expresiones. Ejemplos de Literales

- **Números:** Los números pueden ser enteros o decimales.

```
let num = 123; // Número entero
let num2 = 123.45; // Número decimal
```

- **Cadenas de texto (Strings):** Las cadenas de texto pueden escribirse utilizando comillas dobles " o comillas simples '.

```
let cadena = "mi cadena"; // Usando comillas dobles
let cadena_comillas_simples = 'mi cadena'; // Usando comillas simples
```

- **Booleanos:** Los valores booleanos son **true** o **false**.

```
const bandera = true; // Literal booleano verdadero
const cansado = false; // Literal booleano falso
```

- **Null:** El literal **null** representa la ausencia de un valor.

```
let objeto = null; // La variable 'objeto' no tiene ningún valor asignado
```

2.3 Identificadores en JavaScript

Los identificadores son nombres utilizados para identificar variables, funciones, clases u otros elementos dentro del código. Siguen ciertas reglas para ser válidos en JavaScript.

Reglas para crear identificadores

- Debe comenzar con una letra (a-z, A-Z), un guion bajo _ o un símbolo de dólar \$.
- Después del primer carácter, pueden incluirse números (0-9), letras, guiones bajos _ y símbolos de dólar \$.
- No pueden comenzar con un número.
- No pueden incluir caracteres especiales como +, -, *, &, etc.

Ejemplos válidos:

```
const nombre_variable_1 = "valor"; // Comienza con una letra
const _x = 10; // Comienza con un guion bajo
const $variable = "dólares"; // Comienza con un símbolo de dólar
```

Ejemplos no válidos:

```
const 1a = 20; // No válido: comienza con un número
const suma+ = 15; // No válido: contiene un carácter especial '+'
```

Ejemplos adicionales para mayor claridad

- Variable válida y declaración:

```
let resultado = 100; // 'resultado' es un identificador válido
```

Aquí, resultado es un identificador válido porque comienza con una letra y no contiene caracteres especiales.

Función con un identificador válido:

```
function calcularSuma(a, b) {  
    return a + b;  
}
```

`calcularSuma` es un identificador válido para una función que toma dos parámetros (a y b) y devuelve su suma.

Identificadores no válidos y corrección:

```
// Incorrecto  
// const 2variable = 10; // No puede comenzar con un número  
  
// Correcto  
const variable2 = 10; // Cambiado para comenzar con una letra
```

En este caso, `2variable` es incorrecto porque comienza con un número. Lo corregimos cambiando el nombre a `variable2`, que comienza con una letra.

3 Palabras reservadas, unicode, punto y coma

3.1 Palabras reservadas

Las **palabras reservadas** son identificadores que tienen un significado especial en el lenguaje y, por lo tanto, no pueden ser utilizados como nombres de variables, funciones o etiquetas. Estas palabras están reservadas por el lenguaje para mantener la sintaxis y las reglas de JavaScript.

Ejemplos:

Algunas de las **palabras reservadas** más comunes en JavaScript son:

- **if, else, for, while, switch**: utilizadas para el control de flujo.
- **var, let, const**: utilizadas para declarar variables.
- **function**: utilizada para declarar funciones.
- **return**: utilizada para devolver un valor de una función.
- **class, extends, super**: utilizadas en la programación orientada a objetos con clases.
- **try, catch, finally**: utilizadas para el manejo de excepciones.

Ejemplo en Código:

```
// Esto es correcto  
let nombre = "Juan";  
  
// Esto es incorrecto y causará un error  
let if = "algo"; // "if" es una palabra reservada y no puede usarse como nombre de variable  
  
// Correcto uso de una palabra reservada
```

```
if (nombre === "Juan") {  
    console.log("El nombre es Juan");  
}
```

3.2 Unicode en JavaScript

JavaScript utiliza **Unicode**, un estándar de codificación que permite representar la mayoría de los caracteres escritos del mundo. Esto incluye caracteres de alfabetos no latinos, símbolos especiales, emojis, etc.

Ejemplo:

Puedes utilizar caracteres Unicode directamente en tus cadenas de texto o variables. Incluso, puedes utilizar Unicode en los nombres de las variables, aunque esto no es recomendable.

Ejemplo en Código:

```
// Ejemplo con caracteres Unicode en una cadena  
let saludo = "Hola, Τζάβασκριπτ"; // Javascript en griego  
  
// Ejemplo con caracteres Unicode en nombres de variables (poco común pero posible)  
let π = 3.14159;  
  
console.log(saludo); // Output: Hola, Τζάβασκριπτ  
console.log("Valor de pi: " + π); // Output: 3.14159
```

Escape Unicode:

JavaScript también permite usar **secuencias de escape Unicode** para representar caracteres. Estas secuencias comienzan con \u seguido de un código hexadecimal de cuatro dígitos.

```
let corazon = "\u2764";  
console.log(corazon); // ❤
```

3.3 Uso del Punto y Coma en JavaScript

El punto y coma (;) en JavaScript se utiliza para terminar una instrucción. Sin embargo, JavaScript tiene una característica llamada **Automatic Semicolon Insertion (Inserción Automática de Punto y Coma, ASI)**, que permite al intérprete agregar puntos y comas automáticamente en algunos casos donde faltan.

Ejemplos:

```
// Uso explícito del punto y coma  
let nombre = "Juan";  
console.log(nombre);  
  
// JavaScript puede agregar un punto y coma automáticamente  
let apellido = "Pérez"  
console.log(apellido) // Aunque falta el punto y coma, no causará un error
```

Importancia del Punto y Coma:

Aunque ASI (Automatic Semicolon Insertion) ayuda a evitar errores, hay casos donde no poner el punto y coma puede llevar a un comportamiento inesperado. Por ejemplo:

```
// Ejemplo donde la falta de punto y coma causa un problema
let suma = 5 + 5
(function() {
    console.log("Esto es una función IIFE");
})();

// Esto causará un error, porque JavaScript intenta interpretar como una sola instrucción:
// let suma = 5 + 5(function() { console.log("Esto es una función IIFE");})();
```

4 Declaración de variables

4.1 Declaración de Variables con var

var es la forma tradicional de declarar variables en JavaScript, y ha sido utilizada desde las primeras versiones del lenguaje. Las variables declaradas con **var** tienen un **ámbito de función**, lo que significa que su visibilidad se limita a la función en la que se declara. Sin embargo, si se declara una variable con **var** fuera de cualquier función, ésta tendrá un **ámbito global**. Esto quiere decir que la variable es accesible en todo el documento.

Ejemplo básico de **var**

```
var a = 10;

if (a > 9) {
    var b = 2;
}

console.log(a); // Output: 10
console.log(b); // Output: 2 (b existe fuera del bloque)
```

En este ejemplo, la variable **b** sigue existiendo y es accesible fuera del bloque **if**. Esto se debe a que **var** no tiene un **ámbito de bloque**, sino de función o global. Este comportamiento puede llevar a errores si no se tiene en cuenta.

```
function mifuncion() {
    var c = 3;
    console.log(b); // Output: 2 (b sigue existiendo dentro de la función)
    console.log(c); // Output: 3
}

mifuncion();
console.log(c); // Error: c no está definida fuera de la función
```

Aquí, la variable **c** se declara dentro de la función **mifuncion**, por lo que no es accesible fuera de ella.

En otros lenguajes de programación como Java, la variable `b` no existiría fuera del bloque `{}` donde fue declarada. Sin embargo, en JavaScript, al usar `var`, la variable `b` tiene visibilidad fuera del bloque donde se definió.



Evita usar `var` para declarar variables. Usa `let` o `const` en su lugar.

4.2 Hoisting

El **hoisting** es un comportamiento en JavaScript en el que las declaraciones de variables y funciones se mueven al comienzo del ámbito donde están declaradas. Este comportamiento afecta únicamente a las declaraciones, no a las asignaciones.

```
function mifuncion() {  
    console.log(c); // Output: undefined  
    var c = 3;  
    console.log(c); // Output: 3  
}
```

En este ejemplo, aunque la declaración de la variable `c` aparece después del primer `console.log`, JavaScript mueve la declaración al inicio de la función. Sin embargo, la asignación de `c` a 3 no se mueve, por lo que inicialmente `c` tiene el valor `undefined`. Es importante comprender que `undefined` significa que la variable está declarada aunque aún no tiene un valor definido.

En lenguajes como Java, el código anterior generaría un error en el primer `console.log`, ya que la variable `c` no existiría aún.

Puedes experimentar con el **hoisting** utilizando la sentencia `debugger` para detener la ejecución del código y observar el comportamiento de las variables en el navegador.

```
function pruebaHoisting() {  
    debugger;  
    console.log(x);  
    var x = 10;  
    console.log(x);  
}  
  
pruebaHoisting();
```

4.3 Declaración de Variables con `let` y `const` (ES6)

La palabra reservada `let` se introdujo en ES6 y permite declarar variables con un **ámbito de bloque**. Esto significa que la variable sólo es accesible dentro del bloque `{}` donde se declaró.

```
let a = 1;  
if (a > 0) {  
    let b = 2;
```

```

        console.log(b); // Output: 2
}
console.log(b); // Error: b no está definida

```

4.4 Declaración con const

La palabra reservada **const** también fue introducido en ES6 y se utiliza para declarar **constants**, es decir, variables cuyo valor no puede ser reasignado después de su declaración. Sin embargo, en el caso de objetos, no se puede modificar la referencia, pero sí el contenido.

```

const mensaje = "hola mundo";
mensaje = "adios"; // Error: no se puede modificar el valor de una constante

const persona = { nombre: "Lucia", apellidos: "Molina" };
persona.nombre = "Antonio"; // Esto es permitido
persona = {}; // Error: no se puede cambiar la referencia de un objeto constante

```

El ámbito de **let** y **const** es de bloque, lo que significa que sólo existen dentro del bloque {} donde fueron declarados, al igual que en otros lenguajes como **Java**.

```

let x = 10;
{
  let y = 20;
  const pi = 3.14;
  console.log(x); // Output: 10
  console.log(y); // Output: 20
  console.log(pi); // Output: 3.14
}
console.log(y); // Error: y no está definida
console.log(pi); // Error: pi no está definida

```



Opta en primer lugar por usar **const** para declarar todas las variables. Si el valor de la variable necesita cambiar, entonces usa **let**.

4.5 Tipado Dinámico en JavaScript

A diferencia de lenguajes como **Java**, donde el tipo de una variable se declara de manera explícita y no puede cambiar, en JavaScript el tipo de una variable **puede cambiar durante la ejecución**. Esto se conoce como **tipado dinámico**.

```

let variable = 1;
console.log(typeof variable); // Output: "number"

variable = "Ahora soy un string";
console.log(typeof variable); // Output: "string"

```



Aunque JavaScript permite el tipado dinámico, es preferible **evitar cambiar el tipo** de una variable una vez que se ha establecido, ya que esto puede llevar a errores difíciles de depurar.

Para depurar el código en JavaScript, utiliza herramientas como el debugger del navegador para observar cómo se comportan las variables y comprender mejor el flujo de tu programa.

5 Tipos de datos en Javascript

En este capítulo, exploraremos los **tipos de datos** disponibles en JavaScript a partir de la versión ES6 y posteriores. Estos tipos de datos son fundamentales para la manipulación de valores en el lenguaje y comprenden tanto **tipos primitivos** como **tipos de objetos**.

5.1 Tipos de Datos Primitivos

Los **tipos de datos primitivos** en JavaScript son los más básicos y no pueden ser divididos en partes más pequeñas.

Los tipos de datos primitivos en JavaScript son **inmutables**, lo que significa que una vez que un valor primitivo se ha creado, no se puede cambiar o modificar. Sin embargo, la variable que contiene ese valor primitivo **puede ser reasignada** para contener un valor diferente.

Por ejemplo, si tienes una variable que contiene un número y luego le asignas un nuevo número, lo que realmente sucede es que la variable ahora apunta a un nuevo valor, pero el valor original permanece inalterado.

```
let x = 10;
x = 20; // La variable 'x' ahora referencia al valor 20, pero el valor 10 no ha cambiado,
simplemente ya no está referenciado por 'x'.
```

Por ejemplo, si intentas modificar una cadena de texto (string), se generará una nueva cadena en lugar de alterar la original.

```
let saludo = "Hola";
let saludoModificado = saludo.toUpperCase(); // 'saludoModificado' es una nueva cadena "HOLA"
console.log(saludo); // "Hola" - La cadena original no ha cambiado
```

5.1.1 Métodos y Propiedades en Tipos Primitivos

Aunque los tipos primitivos en JavaScript no poseen métodos ni propiedades inherentes, en la práctica, se comportan como si los tuvieran. Esto es posible porque JavaScript realiza un proceso auto-

mático conocido como **auto-boxing**. Cuando intentas acceder a un método o propiedad de un valor primitivo, JavaScript temporalmente convierte ese valor primitivo en un **objeto envoltorio** (wrapper object) correspondiente.

JavaScript tiene clases nativas que actúan como envoltorios para cada tipo primitivo, como **String**, **Number**, **Boolean**, **Symbol**, y **BigInt**. Estos objetos envoltorios permiten que los primitivos “hereden” métodos y propiedades útiles, como **toUpperCase()** para cadenas o **toFixed()** para números.

```
let texto = "Hola";
console.log(texto.toUpperCase()); // "HOLA"
```

En el ejemplo anterior, cuando se llama al método **toUpperCase()** en la cadena de texto **texto**, JavaScript convierte temporalmente el valor primitivo “Hola” en un objeto String. Este objeto permite el uso del método **toUpperCase()**. Después de que el método se ejecuta, el objeto temporal se descarta y el resultado es devuelto como un nuevo valor primitivo.

```
const numero = 123.456;
console.log(numero.toFixed(2)); // "123.46" - 'numero' es temporalmente un objeto Number
```

En este caso, el valor primitivo 123.456 es convertido en un objeto **Number** temporalmente, lo que permite utilizar el método **toFixed()** para obtener una cadena con dos decimales. Después, el objeto temporal desaparece, y el valor primitivo sigue siendo inmutable.

Hay seis tipos de datos primitivos en ES6+:

5.1.2 Undefined

El tipo **undefined** representa una variable que ha sido declarada pero no inicializada. Cuando una variable es declarada sin asignarle un valor, su tipo es **undefined**.

```
let variable;
console.log(variable); // Output: undefined
```

5.1.3 Null

El tipo **null** es un valor especial que representa la ausencia intencional de cualquier valor u objeto. Es un valor assignable y se utiliza comúnmente para inicializar variables que se espera que luego contengan un objeto.

```
const obj = null;
console.log(obj); // Output: null
```

5.1.4 Boolean

En JavaScript, un valor booleano es un tipo de dato que solo puede tener uno de dos valores posibles: **true** (verdadero) o **false** (falso). Este tipo de dato es fundamental para realizar comparaciones y controlar el flujo del programa mediante estructuras condicionales como **if**, **else**, **while**, y **for**.

```
const esVerdadero = true;
const esFalso = false;
```

5.1.4.1 Conversiones de Booleanos En JavaScript, cualquier valor puede ser convertido a un booleano utilizando la función **Boolean()**, o simplemente evaluándolo en un contexto que requiere un valor booleano (como en una condición **if**).

5.1.4.1.1 Valores que se convierten a false (falsy values): Los siguientes valores son considerados como “falsy”, es decir, se convierten a **false** cuando son evaluados en un contexto booleano:

- **false**
- 0 (el número cero)
- “” (cadena de texto vacía)
- **null**
- **undefined**
- **Nan** (Not a Number)

```
console.log(Boolean(0));          // false
console.log(Boolean(""));        // false
console.log(Boolean(null));       // false
console.log(Boolean(undefined));  // false
console.log(Boolean(NaN));       // false
```

5.1.4.1.2 Valores que se convierten a true (truthy values): Cualquier valor que no sea uno de los “falsy” mencionados anteriormente es considerado “truthy”, es decir, se convierte a **true** en un contexto booleano.

```
console.log(Boolean(1));          // true
console.log(Boolean("Hola"));      // true
console.log(Boolean([]));         // true (un array vacío)
console.log(Boolean({}));         // true (un objeto vacío)
console.log(Boolean(function(){})); // true (una función)
```

5.1.4.1.3 Igualdad estricta (==) En JavaScript, existen dos operadores de igualdad principales: el operador de igualdad **==** y el operador de igualdad estricta **==**.

- Igualdad simple (**==**):

El operador `==` compara dos valores para ver si son iguales después de convertirlos a un tipo común, lo que puede llevar a resultados inesperados debido a la coerción de tipos.

```
console.log(1 == "1"); // true, debido a que "1" es convertido a número antes de la  
comparación  
console.log(true == 1); // true, porque `true` se convierte a 1  
console.log(null == undefined); // true, porque null y undefined se consideran iguales en  
este caso
```

- Igualdad estricta ($==$):

El operador `==` compara tanto el valor como el tipo sin realizar ninguna conversión. Esto significa que ambos deben ser exactamente iguales en **valor** y **tipo** para que la comparación devuelva `true`.

```
console.log(1 === "1"); // false, porque uno es número y el otro es cadena de texto  
console.log(true === 1); // false, porque uno es booleano y el otro es número  
console.log(null === undefined); // false, porque son tipos diferentes  
console.log(1 === 1); // true, porque ambos son números iguales
```



Usar el operador de igualdad estricta === es recomendable porque evita errores sutiles que pueden surgir por la **coerción** (conversión) de tipos cuando se usa ==. Con ===, te aseguras de que no haya conversión de tipos automática, lo que hace que el código sea más predecible y menos propenso a errores.

5.1.5 Number

El tipo **number** representa tanto números **enteros** como de **punto flotante**. JavaScript no distingue entre tipos de números como lo hacen otros lenguajes de programación.

```
const entero = 42;  
const decimal = 3.14;  
console.log(entero); // Output: 42  
console.log(decimal); // Output: 3.14
```

El tipo **number** también soporta una variedad de operadores matemáticos y de comparación:

- Operadores matemáticos: `+`, `-`, `*`, `/`, `%`, `**` (Potencia).
 - Operadores unarios: `++`, `--` (Prefijo y postfijo).
 - Operadores binarios: `>>`, `<<`.

Math es un objeto incorporado que proporciona una serie de métodos y propiedades para realizar operaciones matemáticas comunes. Math no es un constructor, lo que significa que no puedes instanciarlo como un objeto, sino que accedes a sus métodos y propiedades directamente desde el objeto Math.

El objeto Math incluye funciones para operaciones como redondeo, potencias, trigonometría, logaritmos, generación de números aleatorios, y más. Es una herramienta esencial para cualquier desarrollo que requiera cálculos matemáticos en JavaScript.

```
const resultado = Math.PI * Math.pow(2, 3); // Potencia
console.log(Math.abs(-5)); // Valor absoluto 5
console.log(Math.round(Math.PI)); // Redondea: 3
console.log(Math.ceil(1.9)); // Techo: 2
console.log(Math.floor(1.6)); // Piso: 1
console.log(Math.max(1, 3, 8, 9, 123, 5, 0)); // Máximo: 123
console.log(Math.min(5, -2, 100, 10, 9)); // Mínimo: -2
console.log(Math.random()); // Número aleatorio entre 0 y 1
console.log(Math.sqrt(9)); // Raíz cuadrada: 3
console.log(Math.sin(Math.PI / 2)); // Función seno: 1
```



El objeto **Math** es nativo de JavaScript y no necesita ser importado.

5.1.6 String

El tipo **string** se utiliza para representar texto. Los valores de tipo string se pueden definir utilizando comillas simples (' '), comillas dobles (" "), o plantillas de cadena (template literals) usando comillas invertidas (` `).

```
const cadena1 = "Hola, Mundo"; // Cadena con comillas dobles
const cadena2 = 'Hola, Mundo'; // Cadena con comillas simples
const cadena3 = `Hola, Mundo`; // Template String

const nombre = "Luis";
const saludoConTemplate = `Hola, ${nombre}`; // Template String. Interpolación de variables.
console.log(saludoConTemplate); // Output: Hola, Luis
```

Las tres formas de definir cadenas son equivalentes en términos de contenido, pero las **template strings** (` `) ofrecen **funcionalidades adicionales**, como la interpolación de variables y la facilidad para escribir cadenas multilínea.

5.1.6.1 Escapado de Caracteres Los strings pueden contener caracteres especiales que necesitan ser “escapados” utilizando la barra invertida (\ \). Algunos de los más comunes son:

```
const str = 'It's a beautiful day'; // Escapando la comilla simple
const str2 = "She said \"Hello\""; // Escapando la comilla doble
const str3 = "Una línea\nOtra línea"; // Nueva línea
```

```
const str4 = "C:\\\\Users\\\\Usuario"; // Barra invertida
```

5.1.6.2 Concatenación de Strings La **concatenación de strings** se puede hacer utilizando el operador + o el método `concat()`.

```
const saludo = "Hola";
const nombre = "Juan";
const mensaje = saludo + " " + nombre + "!";
console.log(mensaje); // "Hola Juan!"
console.log(saludo.concat(" ", nombre, ".")); // "Hola Juan."
```

5.1.6.3 Comparación de Strings Los strings en JavaScript se comparan utilizando operadores como ==, ===, !=, !==, <, >, etc. Las comparaciones de strings son sensibles a mayúsculas y minúsculas y se realizan en base al valor Unicode de los caracteres.

```
const str1 = "abc";
const str2 = "def";
const str3 = "ABC";
console.log(str1 < str2); // true, porque "a" es menor que "d" en Unicode
console.log(str1 < str3); // true, porque "abc" es lexicográficamente menor que "ABC"
```

5.1.6.4 Conversión de Otros Tipos a String Puedes convertir otros tipos de datos a strings usando el método `String()` o `toString()`.

```
let num = 123;
let str = String(num); // "123"
let str2 = num.toString(); // "123"
```

5.1.6.5 Propiedades y métodos Las cadenas en JavaScript vienen con varios métodos y propiedades útiles que permiten manipular y analizar el texto de manera eficiente.

- Propiedad `length`

La propiedad `length` devuelve el número de caracteres en una cadena, incluidos los espacios.

```
let texto = "Hola, Mundo!";
console.log(texto.length); // 12
```

Métodos de Manipulación:

- `toUpperCase()` y `toLowerCase()`: Convertir la cadena a mayúsculas o minúsculas.

```
let texto = "Hola, Mundo!";
console.log(texto.toUpperCase()); // "HOLA, MUNDO!"
console.log(texto.toLowerCase()); // "hola, mundo!"
```

- `charAt(index)`: Obtener el carácter en una posición específica.

```
let texto = "Hola";
console.log(texto.charAt(1)); // "o"
```

- `substring(start, end)`: Extraer una subcadena entre dos índices (el índice de end no se incluye).

```
let texto = "JavaScript";
console.log(texto.substring(0, 4)); // "Java"
```

- `slice(start, end)`: Similar a `substring()`, pero permite índices negativos para contar desde el final de la cadena.

```
let texto = "JavaScript";
console.log(texto.slice(-6)); // "Script"
```

- `split(separator)`: Divide la cadena en un array de subcadenas, utilizando un separador especificado.

```
let texto = "Hola, Mundo!";
let palabras = texto.split(" ");
console.log(palabras); // ["Hola,", "Mundo!"]
```

- `trim()`: Elimina los espacios en blanco al principio y al final de la cadena.

```
let texto = " Hola, Mundo! ";
console.log(texto.trim()); // "Hola, Mundo!"
```

- `replace(searchValue, newValue)`: Reemplaza una parte de la cadena con otra.

```
let texto = "Hola, Mundo!";
let nuevoTexto = texto.replace("Mundo", "JavaScript");
console.log(nuevoTexto); // "Hola, JavaScript!"
```

- `includes(substring)`: Devuelve true si la cadena contiene la subcadena especificada.

```
let texto = "Hola, Mundo!";
console.log(texto.includes("Mundo")); // true
```

- `indexOf(substring)` y `lastIndexOf(substring)`: Devuelve la posición de la primera o última aparición de la subcadena.

```
let texto = "Hola, Mundo! Hola!";
console.log(texto.indexOf("Hola")); // 0
console.log(texto.lastIndexOf("Hola")); // 13
```

5.1.6.6 Template Strings Los **Template Strings** (o **Template Literals**) son una característica avanzada de JavaScript introducida en ECMAScript 2015 (ES6) que ofrece una forma más flexible y potente de trabajar con cadenas de texto. A continuación, se explican en profundidad sus características, usos, y beneficios.

Los Template Strings se crean usando backticks (` `) en lugar de comillas simples (' ') o dobles (" "). Esto permite a los desarrolladores crear cadenas de texto de manera más dinámica y legible.

```
let saludo = `Hola Mundo`;
```

• Interpolación de Expresiones

Una de las características más poderosas de los Template Strings es la capacidad de **interpolar** (insertar) expresiones JavaScript directamente dentro de la cadena usando \${ }.

```
let nombre = "Juan";
let edad = 30;
let mensaje = `Mi nombre es ${nombre} y tengo ${edad} años.`;
console.log(mensaje); // "Mi nombre es Juan y tengo 30 años."
```

Dentro de \${ }, puedes colocar cualquier expresión válida de JavaScript, como operaciones matemáticas, llamadas a funciones, o incluso expresiones ternarias.

```
let a = 5;
let b = 10;
let resultado = `La suma de ${a} y ${b} es ${a + b}.`;
console.log(resultado); // "La suma de 5 y 10 es 15."
```

• Multi-línea

Los Template Strings permiten la creación de cadenas de texto que abarcan múltiples líneas sin necesidad de concatenar strings o usar secuencias de escape como \\n.

```
let mensaje = `Este es un mensaje
que se extiende
a través de varias líneas.`;
console.log(mensaje);
/*
Salida:
Este es un mensaje
que se extiende
a través de varias líneas.
*/
```

Este uso de **Template Strings** mejora enormemente la legibilidad y la gestión de textos largos o estructuras HTML en JavaScript.

• Tags o Funciones de Plantilla

Los template strings también soportan una característica avanzada llamada tagged templates o funciones de plantilla. Permiten que una función procese un template string antes de que se interprete.

Esta funcionalidad es útil para crear soluciones avanzadas como plantillas personalizadas, traducciones, o escapado de HTML.

```
function etiqueta(strings, ...valores) {
  console.log(strings); // ["Hola ", " soy ", ""]
  console.log(valores); // ["Juan", 30]
  return `Saludos, ${valores[0]}. Tienes ${valores[1]} años.`;
}

let nombre = "Juan";
let edad = 30;
let resultado = etiqueta`Hola ${nombre} soy ${edad}`;
console.log(resultado); // "Saludos, Juan. Tienes 30 años."
```

En este ejemplo, `strings` es un array que contiene las partes literales de la cadena, mientras que `valores` es un array que contiene las expresiones interpoladas.



Es importante recordar las funciones de plantilla de las template strings ya que las usaremos en los **styled components de React**.

- **Raw Strings**

Los template strings también tienen un método incorporado llamado `String.raw` que permite obtener la representación “cruda” de la cadena, es decir, sin procesar las secuencias de escape.

```
let path = String.raw`C:\Development\profile\aboutme.html`;
console.log(path); // "C:\Development\profile\aboutme.html"
```

En una cadena normal, tenemos que escapar las barras invertidas:

```
let path = "C:\\Development\\\\profile\\\\aboutme.html";
console.log(path); // "C:\\Development\\\\profile\\\\aboutme.html"
```

En el primer caso, las secuencias de escape no se procesan y se mantienen tal cual.

- **Tagged Templates para Sanitizar Entradas**

Un uso práctico de los **tagged templates** es para **sanitizar entradas**, como escapar caracteres potencialmente peligrosos para evitar ataques de inyección de HTML o SQL. Aquí un ejemplo simplificado de cómo podrías usarlo:

```
function escapeHTML(literals, ...values) {
  let resultado = "";
  for (let i = 0; i < literals.length; i++) {
    resultado += literals[i] + (values[i] || '') 
      .replace(/&/g, "&amp;")
      .replace(/</g, "&lt;")
      .replace(/>/g, "&gt;")
      .replace(/\"/g, "&quot;")
      .replace(/\'/g, "&#39;");
  }
  return resultado;
}
```

```
let userInput = "<script>alert('Malicious Code');</script>";
let sanitizedString = escapeHTML`Usuario dijo: ${userInput}`;
console.log(sanitizedString);
// "Usuario dijo: &lt;script&gt;alert(&#39;Malicious Code'&#39;);&lt;/script&gt;"
```

En este ejemplo, `escapeHTML` es una función que toma una cadena de plantilla y reemplaza caracteres especiales para prevenir ataques de inyección de HTML.

- **Usos Prácticos Comunes de las template strings**

Construcción de HTML dinámico:

```
let items = ['Manzana', 'Banana', 'Cereza'];
let listaHTML =
  `<ul>
    ${items.map(item => `<li>${item}</li>`).join('')}
  </ul>`;
console.log(listaHTML);
```

Generación de consultas SQL dinámicas:

```
let tableName = 'users';
let columns = ['name', 'age', 'email'];
let sqlQuery = `SELECT ${columns.join(', ')} FROM ${tableName}`;
console.log(sqlQuery); // "SELECT name, age, email FROM users"
```

5.1.6.7 Tratamiento de Cadenas como Arrays Las cadenas en JavaScript pueden ser tratadas de manera similar a un array de caracteres, permitiendo el acceso a caracteres individuales utilizando la notación de corchetes (`[]`).

```
let texto = "JavaScript";
console.log(texto[0]); // "J"
console.log(texto[4]); // "S"

const cadena = "Mi "cadena;
const letra = cadena[0];
for(let i=0; i < cadena.length; i++) {
  console.log(cadena[i]);
}
```

5.1.6.8 Expresiones Regulares Las **expresiones regulares**, o regex (por su nombre en inglés, regular expressions), son una herramienta poderosa para buscar y manipular cadenas de texto. En JavaScript, las expresiones regulares se utilizan para realizar operaciones de búsqueda, coincidencia y reemplazo en cadenas de texto. Vamos a explorar en profundidad cómo funcionan las expresiones regulares en JavaScript, su sintaxis, y algunos ejemplos prácticos.

1. Creación de Expresiones Regulares

En JavaScript, las expresiones regulares se pueden crear de dos formas:

- Usando la sintaxis literal: Se encierran entre barras / / .

```
let regex = /abc/;
```

- Usando el constructor `RegExp`: Este enfoque es útil cuando se desea crear una expresión regular de manera dinámica.

```
let regex = new RegExp('abc');
```

Ambas formas son equivalentes, pero la sintaxis literal es más común.

2. Elementos Básicos de las Expresiones Regulares

Las expresiones regulares están formadas por una combinación de caracteres literales y metacaracteres especiales. Aquí se describen algunos de los elementos más básicos y comunes:

- **Caracteres Literales**

Son los caracteres que se buscan tal cual en el texto.

```
let regex = /cat/;
let str = "The cat is on the roof.";
console.log(regex.test(str)); // true
```

- **Metacaracteres Especiales**

Estos caracteres tienen significados especiales en las expresiones regulares. Algunos de los más comunes incluyen:

- . (punto): Coincide con cualquier carácter, excepto con un salto de línea.

```
let regex = /c.t/;
console.log(regex.test("cat")); // true
console.log(regex.test("cut")); // true
console.log(regex.test("cmt")); // true
```

- \d: Coincide con cualquier dígito (equivalente a [0-9]).

```
let regex = /\d/;
console.log(regex.test("abc123")); // true
console.log(regex.test("abc")); // false
```

- \w: Coincide con cualquier carácter alfanumérico (letras, dígitos y guion bajo).

```
let regex = /\w/;
console.log(regex.test("hello_world")); // true
```

- \s: Coincide con cualquier carácter de espacio en blanco (espacios, tabulaciones, saltos de línea).

```
let regex = /\s/;
console.log(regex.test("hello world")); // true
```

- ^: Indica el inicio de una cadena.

```
let regex = /^hello/;
console.log(regex.test("hello world")); // true
console.log(regex.test("world hello")); // false
```

- \$: Indica el final de una cadena.

```
let regex = /world$/;
console.log(regex.test("hello world")); // true
console.log(regex.test("world hello")); // false
```

- *: Coincide con cero o más repeticiones del carácter o patrón anterior.

```
let regex = /ca*t/;
console.log(regex.test("ct")); // true
console.log(regex.test("cat")); // true
console.log(regex.test("caaat")); // true
```

- +: Coincide con una o más repeticiones del carácter o patrón anterior.

```
let regex = /ca+t/;
console.log(regex.test("cat")); // true
console.log(regex.test("caaat")); // true
console.log(regex.test("ct")); // false
```

- ?: Coincide con cero o una repetición del carácter o patrón anterior.

```
let regex = /ca?t/;
console.log(regex.test("cat")); // true
console.log(regex.test("ct")); // true
console.log(regex.test("caat")); // false
```

- {n}: Coincide exactamente con n repeticiones del carácter o patrón anterior.

```
let regex = /a{3}/;
console.log(regex.test("aaa")); // true
console.log(regex.test("aa")); // false
```

- {n,m}: Coincide con entre n y m repeticiones del carácter o patrón anterior.

```
let regex = /a{2,4}/;
console.log(regex.test("aa")); // true
console.log(regex.test("aaa")); // true
console.log(regex.test("aaaa")); // true
console.log(regex.test("a")); // false
```

• Conjuntos y Rangos

- [abc]: Coincide con cualquier carácter dentro de los corchetes.

```
let regex = /[abc]/;
console.log(regex.test("apple")); // true
console.log(regex.test("banana")); // true
console.log(regex.test("cherry")); // true
```

- [a-z]: Coincide con cualquier carácter en el rango especificado.

```
let regex = /[a-z]/;
console.log(regex.test("HELLO")); // false
console.log(regex.test("hello")); // true
```

- [^abc]: Coincide con cualquier carácter excepto los especificados.

```
let regex = /^[^abc]/;
console.log(regex.test("d")); // true
console.log(regex.test("a")); // false
```

3. Modificadores (Flags)

Las expresiones regulares en JavaScript pueden tener modificadores que cambian su comportamiento:

- i: Ignora la distinción entre mayúsculas y minúsculas.

```
let regex = /hello/i;
console.log(regex.test("HELLO")); // true
```

- g: Busca todas las coincidencias en lugar de detenerse en la primera.

```
let regex = /l/g;
let str = "hello world";
console.log(str.match(regex)); // ["l", "l", "l"]
```

- m: Habilita el modo multilínea, afectando ^ y \$ para que coincidan al inicio y al final de cada línea, no solo del inicio y final de la cadena completa.

```
let regex = /^world/m;
let str = `hello
world`;
console.log(regex.test(str)); // true
```

4. Métodos Comunes que Usan Expresiones Regulares

- test()

El método test() verifica si una cadena cumple con la expresión regular y devuelve true o false.

```
let regex = /world/;
console.log(regex.test("hello world")); // true
```

- exec()

El método `exec()` busca una coincidencia en una cadena y devuelve un array con la primera coincidencia encontrada o `null` si no hay coincidencia.

```
let regex = /world/;
console.log(regex.exec("hello world")); // ["world", index: 6, input: "hello world", groups: undefined]
```

- `match()`

El método `match()` busca una coincidencia en una cadena y devuelve las coincidencias encontradas. Si se usa con el modificador `g`, devuelve todas las coincidencias en un array.

```
let regex = /l/g;
let str = "hello world";
console.log(str.match(regex)); // ["l", "l", "l"]
```

- `replace()`

El método `replace()` busca una coincidencia en una cadena y reemplaza la primera (o todas si se usa `g`) coincidencia con una nueva subcadena.

```
let regex = /world/;
let str = "hello world";
let nuevoStr = str.replace(regex, "JavaScript");
console.log(nuevoStr); // "hello JavaScript"
```

- `search()`

El método `search()` busca una coincidencia en una cadena y devuelve el índice de la primera coincidencia, o `-1` si no hay coincidencia.

```
let regex = /world/;
let str = "hello world";
console.log(str.search(regex)); // 6
```

- `split()`

El método `split()` divide una cadena en un array de subcadenas utilizando una expresión regular como delimitador.

```
let regex = /\s/;
let str = "hello world";
console.log(str.split(regex)); // ["hello", "world"]
```

5. Grupos y Referencias

Las expresiones regulares permiten agrupar partes de una expresión y referirse a ellas más tarde. Los grupos se crean con paréntesis () y se pueden referenciar con `\1`, `\2`, etc., dependiendo de su orden.

```
let regex = /(cat|dog)s/;
console.log(regex.test("cats")); // true
console.log(regex.test("dogs")); // true
```

```
console.log(regex.test("mice")); // false
```

6. Ejemplos

- Validar una dirección de correo electrónico:

```
let regex = /^[^\w\.-]+@[^\w\.-]+\.\w{2,4}$/;
console.log(regex.test("example@mail.com")); // true
console.log(regex.test("example@.com")); // false
```

- Buscar números de teléfono:

```
let regex = /^(6|7|8|9)\d{8}$/;

let validPhoneNumber = "612345678";
let invalidPhoneNumber = "512345678";

console.log(regex.test(validPhoneNumber)); // true
console.log(regex.test(invalidPhoneNumber)); // false
```

- Extraer números de una cadena:

```
let regex = /\d+/g;
let str = "El precio es 50 dólares";
console.log(str.match(regex)); // ["50"]
```

- Validar si una cadena contiene solo letras y espacios

```
let regex = /^[A-Za-z\s]+$/;
let text = "Hello World";
console.log(regex.test(text)); // true
```

- Reemplazar todas las ocurrencias de un carácter o palabra

```
let regex = /foo/g;
let text = "foo bar foo baz";
let newText = text.replace(regex, "qux");
console.log(newText); // "qux bar qux baz"
```



Las expresiones regulares son ideales para validar datos, como correos electrónicos, números de teléfono, direcciones IP, entre otros. En el frontend puedes usarlas para realizar las validaciones en los formularios antes de enviar los datos al servidor.

5.1.7 Symbol

Introducido en ES6, el **tipo symbol** es un tipo de dato primitivo **único e inmutable** que se utiliza como identificador para propiedades de objetos. Los valores de tipo symbol son únicos, incluso si se crean con la misma descripción.

```
let id = Symbol('id');
console.log(id); // Output: Symbol(id)
```

5.1.7.1 Unicidad de Symbol Cada símbolo es único, incluso si dos símbolos tienen la misma descripción:

```
let simboloA = Symbol("etiqueta");
let simboloB = Symbol("etiqueta");

console.log(simboloA === simboloB); // false
```

Esto significa que `simboloA` y `simboloB` son completamente diferentes, a pesar de que fueron creados con la misma descripción.

5.1.7.2 Usos de Symbol `Symbol` es útil para crear propiedades en objetos que no colisionen con otras propiedades y que no sean accesibles de forma accidental. Esto es especialmente útil en el desarrollo de bibliotecas o frameworks, donde el riesgo de colisión de nombres es alto.

Ejemplo con objetos:

```
// Usar un símbolo como clave de una propiedad de objeto
const objeto = {};
const claveSimbolo = Symbol('claveUnica');
const claveSimbolo2 = Symbol('claveUnica');

objeto[claveSimbolo] = 'Valor secreto';
objeto[claveSimbolo2] = 'Valor secreto 2';
console.log(objeto[claveSimbolo]); // Output: Valor secreto
console.log(objeto[claveSimbolo2]); // Output: Valor secreto 2
```

En este ejemplo, la propiedad `id` es única y no se puede acceder a ella usando una cadena de texto como clave (`usuario.id` devuelve `undefined`).

5.1.7.3 Propiedades ocultas Las propiedades de un objeto definidas usando un `Symbol` no se enumeran en un bucle `for...in` y no se ven cuando se usa `Object.keys()` o `Object.getOwnPropertyNames()`. Sin embargo, se pueden obtener con `Object.getOwnPropertySymbols()`.

```
let id = Symbol("id");

let usuario = {
  nombre: "Juan",
  [id]: 123
};

for (let clave in usuario) {
  console.log(clave); // solo "nombre", no se ve la propiedad simbolo
}
```

```
console.log(Object.keys(usuario)); // ["nombre"]
console.log(Object.getOwnPropertySymbols(usuario)); // [Symbol(id)]
```



El uso de los **objetos** en Javascript se verá más adelante en el curso.
El uso de **Symbol** es muy poco común, pero es útil para crear propiedades de objetos que no se pueden acceder o modificar accidentalmente.

5.2 Tipos de Datos de Objeto

Además de los tipos primitivos, JavaScript tiene un tipo especial de objeto que se utiliza para almacenar colecciones de datos y funcionalidades más complejas. Los tipos de objetos incluyen:

5.2.1 Objetos (Object)

Un **objeto** es una colección de propiedades y métodos. Cada propiedad es una asociación entre una clave y un valor. Los objetos se crean utilizando llaves {} o utilizando constructores como [Object](#).

```
let persona = {
  nombre: 'Juan',
  edad: 30
};

console.log(persona.nombre); // Output: Juan
console.log(persona.edad); // Output: 30
```

5.2.2 Arrays (Array)

Un **array** es una colección ordenada de elementos, que puede contener valores de cualquier tipo. Los arrays se crean utilizando corchetes [] o utilizando el constructor [Array](#).

```
let colores = ['rojo', 'verde', 'azul'];
console.log(colores[0]); // Output: rojo
console.log(colores.length); // Output: 3
```

5.2.3 Funciones (Function)

Una **función** es un bloque de código que se puede definir y ejecutar cuando se necesite. Las funciones en JavaScript también son un tipo especial de objeto.

```
function saludar(nombre) {
  return `Hola, ${nombre}`;
}

console.log(saludar('Carlos')); // Output: Hola, Carlos
```

Como son objetos, las funciones pueden ser pasadas como argumentos a otras funciones, devueltas por otras funciones y asignadas a variables.

```
function saludar(nombre) {  
    return `Hola, ${nombre}`;  
}  
  
let saludo = saludar;  
console.log(saludo('Carlos')); // Output: Hola, Carlos
```

5.2.4 Mapas (Map)

Un **mapa** es una nueva estructura de datos introducida en ES6 que permite almacenar **pares clave-valor**, donde cualquier tipo de datos puede ser una clave.

```
let mapa = new Map();  
mapa.set('nombre', 'Pedro');  
mapa.set(1, 'uno');  
  
console.log(mapa.get('nombre')); // Output: Pedro  
console.log(mapa.get(1)); // Output: uno
```

5.2.5 Sets (Set)

Un **set** es una colección de valores únicos, es decir, **no permite duplicados**. Los sets se utilizan cuando se necesita asegurarse de que una colección de valores no tenga duplicados.

```
let set = new Set([1, 2, 3, 3, 4]);  
console.log(set); // Output: Set(4) {1, 2, 3, 4}
```

5.3 Tipos de Datos Especiales

Además de los tipos de datos mencionados, JavaScript tiene otros tipos y construcciones especiales.

5.3.1 BigInt

Introducido en ES2020, el tipo **BigInt** permite representar números enteros más grandes que el máximo permitido por el tipo **number**.

```
let numeroGrande = BigInt(9007199254740991);  
console.log(numeroGrande); // Output: 9007199254740991n
```

5.3.2 TypedArray

Los **TypedArray** son objetos similares a los arrays que proporcionan una vista sobre buffers binarios. Se utilizan cuando se trabaja con datos binarios en JavaScript.

```
let buffer = new ArrayBuffer(16);
let vista = new Uint8Array(buffer);
console.log(vista); // Output: Uint8Array(16) [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

5.4 El objeto global en Javascript

El **objeto global en JavaScript** es un objeto que está disponible en todo el entorno de ejecución, es decir, en cualquier lugar del código. Es el contenedor de todos los objetos globales, funciones y variables que están disponibles en un entorno específico de JavaScript (como un navegador web o Node.js).

Dependiendo del entorno, el nombre de este objeto global puede variar.

5.4.1 En entornos de navegador:

En los navegadores web, el objeto global es **window**. Todas las variables y funciones globales que declares se convierten en propiedades del objeto window.

```
var nombre = "John";
console.log(window.nombre); // "John"
```

Aquí, la variable **nombre** se convierte en una propiedad del objeto global **window**.

5.4.2 En Node.js:

En Node.js, el objeto global se llama **global**. Funciona de manera similar al objeto **window** en el navegador, pero es específico para el entorno de Node.js:

```
global.nombre = "John";
console.log(global.nombre); // "John"
```

5.4.3 Acceso a objetos globales:

El objeto global contiene muchas funciones y objetos estándar de JavaScript, como Math, Date, setTimeout, entre otros. Además, cualquier variable que no esté declarada con var, let o const se agregará automáticamente al objeto global (aunque esto es una mala práctica y debería evitarse).

```
function saludar() {  
    console.log("Hola");  
}  
  
console.log(window.saludar === saludar); // true en navegadores
```

5.4.4 Propiedades y métodos comunes:

El **objeto global** incluye propiedades y métodos que se pueden usar sin necesidad de hacer referencia explícita al objeto global:

- Propiedades y Objetos
 - `console`: Para mostrar mensajes en la consola (`log()`, `error()`, `warn()`, `table()`).
 - `window` (navegadores) / `global` (Node.js) / `globalThis` (todos): Objeto global que contiene todas las variables y funciones globales.
 - `Math`: Funciones matemáticas (`random()`, `floor()`, `ceil()`, `round()`, `max()`, `min()`).
 - `Date`: Trabaja con fechas y horas (`new Date()`, `Date.now()`).
 - `JSON`: Manipula datos JSON (`stringify()`, `parse()`).
 - `Infinity`: Representa un valor numérico infinito.
 - `Nan`: Indica que un valor no es un número.
 - `undefined`: Representa un valor no definido.
 - `document` (navegadores): Representa el DOM del documento HTML.
 - `navigator` (navegadores): Información sobre el agente de usuario (navegador).
 - `location`: Información sobre la URL actual.
 - `history`: Permite la manipulación del historial del navegador.
 - `process` (Node.js): Información sobre el proceso en ejecución.
 - `module` (Node.js): Información sobre el módulo actual.
 - `require` (Node.js): Carga módulos en Node.js.
- Funciones:
 - `setTimeout()`: Ejecuta una función después de un tiempo.
 - `setInterval()`: Ejecuta una función repetidamente con un intervalo fijo.
 - `clearTimeout()` / `clearInterval()`: Cancelan `setTimeout` o `setInterval`.
 - `parseInt()`: Convierte una cadena a un número entero.
 - `parseFloat()`: Convierte una cadena a un número de punto flotante.
 - `isNaN()`: Determina si un valor es NaN.
 - `isFinite()`: Verifica si un valor es un número finito.
 - `encodeURI()` / `decodeURI()`: Codifican y decodifican una URI completa.

- `encodeURIComponent()` / `decodeURIComponent()`: Codifican y decodifican componentes de una URI.
- `eval()`: Ejecuta código JavaScript representado como una cadena (se recomienda evitar su uso por seguridad).
- `alert()` (navegadores): Muestra un cuadro de diálogo de alerta.
- `prompt()` (navegadores): Sigue la entrada al usuario.
- `confirm()` (navegadores): Sigue la confirmación del usuario.
- `fetch()`: Realiza solicitudes HTTP/HTTPS y devuelve promesas.
- `atob()` / `btoa()`: Decodifica y codifica en Base64.

5.4.5 Módulos y el objeto global:

En JavaScript moderno, especialmente en entornos como Node.js o cuando se usa ES6 (o ECMAScript 2015) en navegadores, se utilizan módulos para evitar la contaminación del espacio de nombres global. Las variables y funciones dentro de los módulos no se agregan al objeto global, lo que promueve un código más limpio y evita conflictos entre diferentes partes de un programa.



El uso de **módulos** en Javascript se verá más adelante en el curso.

5.4.6 El uso de “globalThis”:

ECMAScript 2020 introdujo `globalThis`, un estándar que proporciona una forma consistente de acceder al objeto global en cualquier entorno, ya sea en el navegador, Node.js, Web Workers, etc.:

```
console.log(globalThis); // Accede al objeto global en cualquier entorno
```

`globalThis` resuelve el problema de tener que saber si estás en un navegador o en Node.js para acceder al objeto global.

5.5 Coerción

La **coerción en JavaScript** es el proceso mediante el cual el lenguaje convierte automáticamente un valor de un tipo de dato a otro, cuando es necesario para que una operación tenga sentido o sea posible. Esta característica puede facilitar la escritura de código, pero también puede llevar a comportamientos inesperados si no se comprende bien cómo funciona.

Existen dos tipos de coerción en JavaScript:

1. Coerción implícita

Ocurre cuando JavaScript convierte automáticamente un valor de un tipo a otro sin que el programador lo indique explícitamente.

- **Coerción a string (concatenación):**

```
let resultado = '5' + 3; // "53"
```

Aquí, el número 3 se convierte en una cadena '3' para que la operación de concatenación pueda ocurrir.

- **Coerción a número (suma, resta, etc.):**

```
let resultado = '5' - 3; // 2
```

En este caso, la cadena '5' se convierte en el número 5 para que la operación de resta pueda realizarse.

- **Coerción a booleano:**

```
if ('') {
  console.log('Esto no se muestra');
}
```

Aquí, la cadena vacía '' se convierte en **false**, por lo que el bloque **if** no se ejecuta.

2. Coerción explícita

Ocurre cuando el programador convierte manualmente un valor de un tipo a otro utilizando funciones o constructores específicos.

- **Convertir a string:**

```
let numero = 123;
let texto = String(numero); // "123"
```

- **Convertir a número:**

```
let texto = "456";
let numero = Number(texto); // 456
let n = parseInt("123"); // 123
let f = parseFloat("1234.12") // 1234.12
```

- **Convertir a booleano:**

```
let valor = 1;
let booleano = Boolean(valor); // true
```

- **Ejemplos de coerción en operaciones comunes**

- Coerción en Comparaciones con == (Doble Igual)

El operador == realiza coerción de tipos si los valores comparados son de diferentes tipos.

```
5 == '5'; // true
```

Aquí, JavaScript convierte la cadena '5' en el número 5 antes de comparar.

- Coerción en Comparaciones con === (Triple Igual)

El operador === no realiza coerción de tipos, compara tanto el valor como el tipo.

```
5 === '5'; // false
```

- Coerción en Operaciones Matemáticas

Cuando se utilizan operadores como +, -, *, /, JavaScript intenta convertir los operandos en números si no lo son.

```
'10' * 2; // 20
```

5.5.1 Peligros y Consideraciones

- Inconsistencia: La coerción implícita puede llevar a resultados inesperados, especialmente cuando se usan operadores como == en lugar de ===.
- Legibilidad del Código: La coerción explícita es generalmente preferible porque hace el código más claro y predecible, ya que es evidente cuándo y cómo se realiza la conversión de tipos.

5.5.2 Ejemplos problemáticos

- Concatenación de Strings con Números:

```
'10' + 1; // "101" (concatenación, no suma)
```

- Comparaciones Imprecisas:

```
[] == false; // true (coerción de [] a "")
```

6 Estructuras condicionales: if, switch, ternario

A continuación, se repasa el uso de las estructuras if, else, switch, y otros conceptos relacionados.

6.1 Estructura if, else if, else

```
let hora = 10;

if (hora < 12) {
    console.log("Buenos días");
}
// Salida: Buenos días
```

```
let hora = 14;

if (hora < 12) {
    console.log("Buenos días");
} else {
    console.log("Buenas tardes");
}
// Salida: Buenas tardes
```

La declaración else if permite agregar múltiples condiciones entre un if inicial y un else final.

```
let hora = 18;

if (hora < 12) {
    console.log("Buenos días");
} else if (hora < 18) {
    console.log("Buenas tardes");
} else {
    console.log("Buenas noches");
}
// Salida: Buenas noches
```

```
const nota = 8;

if (nota < 5) {
    console.log("Suspensos");
} else if (nota < 6) {
    console.log("Aprobado");
} else {
    console.log("Excelente");
}
```

6.2 Evaluación implícita de valores a booleanos

JavaScript convierte automáticamente los valores en condiciones a booleanos (true o false). Esto significa que no solo los valores true y false son válidos, sino también otros tipos de datos, como números, cadenas, arrays, objetos, etc.

```
const bandera = true;

if (bandera) {
    console.log('Entra en el if');
}
```

Aquí, cualquier valor que no sea 0, null, undefined, NaN, una cadena vacía "" o false, se considera true cuando se evalúa en un if. Puedes probar diferentes valores para bandera como 1, 0, "", "cadena", [], {},

[true], null, undefined, {x:1}, [1,2,3] y observar cómo se comporta el if.

6.3 Comprobación de inicialización de variables

Para comprobar si una variable ha sido inicializada, puedes utilizar una condición if. Sin embargo, declarar una variable sin inicializarla (solo con const) lanzará un error en JavaScript. La declaración correcta sería con let o var.

```
let variable;
if (variable) {
    console.log("Inicializada");
} else {
    console.log('No inicializada');
}
```

Este código verifica si variable tiene un valor considerado truthy. Si no ha sido inicializada o es un valor falsy (como null o undefined), se ejecutará el bloque else.

6.4 Uso de operadores lógicos

6.4.1 Operador lógico AND (&&)

El operador && evalúa expresiones de izquierda a derecha y detiene la evaluación tan pronto como una expresión es false. Esto se conoce como “corto-circuito AND”.

```
function A() { console.log('called A'); return null; }
function B() { console.log('called B'); return true; }

console.log(A() && B());
```

En este ejemplo, B nunca es llamada porque A() retorna null, que es un valor falsy. Como resultado, la evaluación se detiene y se retorna null.

6.4.2 Operador lógico OR (||)

El operador || también se evalúa de izquierda a derecha, pero se detiene tan pronto como una expresión es true, devolviendo el valor truthy.

```
function A() { console.log('called A'); return []; }
function B() { console.log('called B'); return true; }

console.log(A() || B());
```

Aquí, B no es llamada porque A() retorna un array [], que es un valor truthy. La evaluación se corta y se devuelve [].

6.5 Estructura switch

La estructura switch se utiliza para seleccionar uno entre varios bloques de código para ejecutar, según el valor de una expresión.

```
function aNotaNumerica(calificacion) {
    let nota = 0;

    switch(calificacion) {
        case "Suspensos":
            nota = 1;
            break;
        case "Aprobado":
            nota = 5;
            break;
        case "Sobresaliente":
            nota = 9;
            break;
        default:
            nota = 0; // Valor por defecto.
    }

    return nota;
}

let calificacion = "Sobresaliente";
console.log(aNotaNumerica(calificacion)); // 9
calificacion = "Otra cosa";
console.log(aNotaNumerica(calificacion)); // 0
```

En este ejemplo, switch compara el valor de calificacion con cada case y ejecuta el código asociado al primer case que coincide. Si no hay coincidencia, se ejecuta el bloque default.

6.6 Operador ternario

El operador ternario (`? :`) es una forma concisa de escribir un if-else. Se utiliza para evaluar una expresión y retornar un valor basado en la condición. Es una expresión a diferencia de if que es una estructura de control.

```
const activado = true;
const color = activado ? "green" : "red";
console.log(color); // "green"

const edad = 18;
const bebida = edad >= 18 ? "cerveza" : "cocacola";
console.log(bebida); // "cerveza"

// Gestionar valores nulos
let person = {name: "Luis"};
console.log(person ? person.name : ""); // "Luis"

person = null;
console.log(person ? person.name : ""); // ""
```

Este operador es útil para realizar asignaciones rápidas basadas en una condición, manteniendo el

código limpio y legible.

7 Bucles

Existen varios tipos de bucles que te permiten ejecutar un bloque de código varias veces, lo cual es útil cuando se trabaja con estructuras de datos como arrays, objetos, o incluso cuando necesitas repetir una operación bajo ciertas condiciones.

7.1 for

El bucle for es el tipo de bucle más tradicional en JavaScript. Se utiliza cuando conoces de antemano cuántas veces quieras que se ejecute el bloque de código.

```
for (let i = 0; i < 5; i++) {  
    console.log('Iteración número: ' + i);  
}
```

7.2 while

El bucle while se utiliza cuando no se conoce de antemano el número exacto de iteraciones. Continúa ejecutándose mientras la condición especificada sea verdadera.

```
let i = 0;  
while (i < 5) {  
    console.log('Iteración número: ' + i);  
    i++;  
}
```

7.3 do...while

El bucle do...while es similar a while, pero con una diferencia importante: el bloque de código se ejecuta al menos una vez antes de que la condición sea evaluada.

```
let i = 0;  
do {  
    console.log('Iteración número: ' + i);  
    i++;  
} while (i < 5);
```

7.4 for...in

El bucle for...in se utiliza para iterar sobre las propiedades enumerables de un objeto. Es útil cuando trabajas con objetos, no con arrays.

```
const persona = { nombre: 'Juan', edad: 30, ciudad: 'Madrid' };

for (let clave in persona) {
  console.log(clave + ': ' + persona[clave]);
}
```

7.5 for...of

El bucle for...of se utiliza para iterar sobre elementos iterables, como arrays, strings, o cualquier objeto que implemente el protocolo iterable (por ejemplo, Map, Set).

```
const array = ['a', 'b', 'c'];

for (let letra of array) {
  console.log(letra);
}
```

7.6 forEach

forEach es un método de los arrays en JavaScript que te permite ejecutar una función específica para cada elemento del array.

```
const numeros = [1, 2, 3, 4, 5];

numeros.forEach(function(numero) {
  console.log(numero);
});
```

Explicación:

- En este ejemplo, la función anónima se ejecuta para cada elemento en el array numeros.
- forEach no puede romperse (usar break), por lo que es menos flexible que un bucle for.

Más adelante veremos otras funciones iteradoras de arrays.

8 Objetos Javascript

En JavaScript, los **objetos son una estructura fundamental** que permite almacenar y manipular datos de manera flexible y poderosa. A diferencia de los tipos primitivos, que son inmutables y se

manejan por valor, los objetos son mutables y se manejan por referencia. Esto los convierte en una herramienta versátil para modelar datos complejos y comportamientos en aplicaciones.

¿Qué es un Objeto en JavaScript?

Un objeto en JavaScript es una **colección desordenada de propiedades**, donde cada propiedad tiene una **clave** (o key) y un **valor** asociado. La clave siempre es un tipo string o symbol, mientras que el valor puede ser de cualquier tipo: primitivo, objeto, función, etc.

Por ejemplo:

```
const persona = {
    nombre: "Luis",
    apellidos: "Molina",
    edad: 30,
    saludar: function() {
        console.log(`Hola, mi nombre es ${this.nombre}`);
    }
};
console.log(persona.nombre);
```

En este ejemplo, `persona` es un objeto con varias propiedades: `nombre`, `apellidos`, `edad` y `saludar`. Esta última es una función y, en este contexto, se llama “método”.



En JavaScript, los objetos son **mutables**, lo que significa que se pueden modificar después de su creación. Es una estructura fundamental para modelar datos y comportamientos en aplicaciones.

8.1 Herencia y Prototipos

JavaScript implementa un modelo de herencia basado en **prototipos**. Cada objeto tiene un prototipo (otra instancia de objeto) del cual hereda propiedades y métodos. Esta herencia se establece mediante una referencia al prototipo en la propiedad interna `[Prototype]` (accesible a través de `__proto__` o `Object.getPrototypeOf()`).

Por ejemplo:

```
const padre = {
    apellido: "Sánchez"
};

const hijo = Object.create(padre);
hijo.nombre = "Alfonso";

console.log(hijo.apellido); // "Sánchez"
```

En este caso, `hijo` hereda la propiedad `apellido` de `padre` a través de la cadena de prototipos.

8.2 Creación de Objetos

Existen varias formas de crear objetos en JavaScript:

- **Literales de Objetos:**

```
const vacio = {};
const punto = {x:0, y:0};
const libro = {
    titulo: "Introducción a JavaScript",
    paginas: 200,
    descripcion: "Un libro sobre JavaScript",
    autor: {
        nombre: "Alfonso",
        apellidos: "Sánchez"
    }
};
```

- **Con el Operador new:**

```
const objeto = new Object();
const fecha = new Date();
const lista = new Array(10);
```

- **Con el Método Object.create():**

```
const prototipo = { nombre: "Luis" };
const objeto = Object.create(prototipo);
console.log(objeto.nombre); // "Luis"
```

- Acabamos de crear un objeto “objeto” que hereda las propiedades de “prototipo”.
- `objeto` no tiene propiedades propias, solo heredadas.
- Para acceder al prototipo desde `objeto`:

```
console.log(objeto.__proto__);
console.log(Object.getPrototypeOf(objeto));
```

- El padre de `prototipo` es `Object`

```
console.log(Object.getPrototypeOf(prototipo));
```

- Los objetos creados mediante un literal tienen un prototipo: `Object.prototype`

```
Object.getPrototypeOf(prototipo) === Object.prototype;
```

- Por este motivo los objetos creados con literales tienen métodos heredados de `Object.prototype`:
 - `toString`
 - `valueOf`

- `isPrototypeOf`
- `hasOwnProperty`
- `toLocaleString`

- Podemos crear un objeto cuyo prototipo es `null`:

```
let o2 = Object.create(null);
Object.getPrototypeOf(o2); // null
o2.toString(); // error el método toString no existe.
```

- `Object.prototype` no tiene prototipo y por lo tanto finaliza la cadena de prototipos.

```
Object.getPrototypeOf(Object.prototype); //null
```

8.3 Propiedades y Configuración

En JavaScript, las **propiedades** de los objetos no solo almacenan valores, sino que también tienen **atributos** internos que determinan su comportamiento. Estos atributos controlan aspectos clave sobre cómo se puede interactuar con esas propiedades, lo que proporciona un control detallado sobre la estructura y la seguridad del objeto.

8.3.1 Atributos de las Propiedades

Cada propiedad de un objeto tiene tres atributos principales que pueden ser configurados:

- **Enumerable (enumerable)**: Indica si la propiedad aparecerá durante la enumeración del objeto, como cuando se utiliza un bucle `for...in` o el método `Object.keys()`. Si `enumerable` es `true`, la propiedad se incluye en estas enumeraciones.
- **Configurable (configurable)**: Determina si la propiedad puede ser eliminada del objeto y si sus atributos (excepto `writable`) pueden ser modificados posteriormente. Si `configurable` es `false`, no podrás eliminar la propiedad ni cambiar su configurabilidad o enumerabilidad.
- **Writable (writable)**: Define si el valor de la propiedad puede ser cambiado mediante asignación. Si `writable` es `false`, cualquier intento de modificar el valor será ignorado (en modo estricto, lanzará un error).

El método `Object.defineProperty()` se utiliza para definir o modificar una propiedad de un objeto, especificando los valores de estos atributos. Esto es útil cuando necesitas un control más granular sobre cómo se comportan las propiedades del objeto.

```
const persona = {};
Object.defineProperty(persona, 'edad', {
```

```

        value: 30,           // El valor de la propiedad
        writable: false,    // No se puede cambiar el valor de la propiedad
        enumerable: true,   // La propiedad aparecerá en la enumeración del objeto
        configurable: false // No se puede eliminar ni reconfigurar la propiedad
    });

console.log(persona.edad); // 30

persona.edad = 35; // No hará nada porque writable es false
console.log(persona.edad); // Sigue siendo 30

for (let key in persona) {
    console.log(key); // 'edad' aparecerá aquí porque enumerable es true
}

delete persona.edad; // Esto fallará porque configurable es false
console.log(persona.edad); // Sigue siendo 30

```

8.3.2 Atributos get y set

Además de los tres atributos mencionados, también puedes definir una propiedad utilizando funciones **getter y setter** en lugar de simplemente un valor. Estas funciones permiten controlar cómo se accede y modifica el valor de la propiedad.

```

Object.defineProperty(persona, 'nombreCompleto', {
    get() {
        return `${this.nombre} ${this.apellido}`;
    },
    set(value) {
        [this.nombre, this.apellido] = value.split(' ');
    },
    enumerable: true,
    configurable: true
});

persona.nombre = 'Juan';
persona.apellido = 'Pérez';

console.log(persona.nombreCompleto); // 'Juan Pérez'

persona.nombreCompleto = 'Carlos López';
console.log(persona.nombre); // 'Carlos'
console.log(persona.apellido); // 'López'

```

8.3.3 Métodos Object.defineProperties y Object.getOwnPropertyDescriptor

- **Object.defineProperties()**: Permite definir varias propiedades a la vez, proporcionando un objeto con múltiples descripciones de propiedades.
- **Object.getOwnPropertyDescriptor()**: Devuelve la descripción completa de una propiedad específica, lo que incluye sus atributos `value`, `writable`, `enumerable`, `configurable`, y funciones `get` y `set` si las tiene.

```

const persona = {};
Object.defineProperties(persona, {
    nombre: {
        value: 'Ana',
        writable: true,
        enumerable: true,
        configurable: true
    },
    edad: {
        value: 25,
        writable: false,
        enumerable: false,
        configurable: false
    }
});

console.log(Object.getOwnPropertyDescriptor(persona, 'nombre'));
// {
//   value: 'Ana',
//   writable: true,
//   enumerable: true,
//   configurable: true
// }

```

8.4 Acceso y Modificación de Propiedades

En JavaScript, las **propiedades** de los objetos pueden ser accedidas y modificadas utilizando dos notaciones principales:

- Notación de punto: `objeto.propiedad`
- Notación de corchetes: `objeto["propiedad"]`

```

const libro = {
    titulo: "Introducción a Javascript",
    paginas: 200,
    "descripción": "Este es un libro sobre los fundamentos de JavaScript", // Las claves (keys)
        // pueden ser strings con o sin comillas.
    autor: { // El valor de una propiedad puede ser otro objeto.
        nombre: "Alfonso",
        apellidos: "Sánchez"
    },
    "contenidos del libro": "Resumen del contenido del libro"
};
// Acceso a propiedades usando notación de punto
const titulo = libro.titulo; // "Introducción a JavaScript"
// Acceso a propiedades usando notación de corchetes
const paginas = libro["paginas"]; // 200

```

8.4.1 Ejemplos Avanzados con Herencia de Prototipos:

```

let o = {} // El prototipo de o es Object.prototype
o.x = 1;

let p = Object.create(o); // p hereda propiedades de o

```

```

p["y"] = 0; // Creación de una propiedad en p usando notación de corchetes.

let q = Object.create(p); // El prototipo de q es p
q.z = 3;

console.log(q); // {z: 3} - q tiene la propiedad z, y hereda x e y
console.log(p); // {y: 0} - p tiene la propiedad y, y hereda x
console.log(o); // {x: 1} - o tiene la propiedad x

// Acceso a propiedades heredadas:
console.log(q.x); // 1 - q hereda la propiedad x de o
console.log(q.y); // 0 - q hereda la propiedad y de p
console.log(q.z); // 3 - z es propiedad directa de q

// Ejemplo con método heredado:
console.log(q.toString()); // Devuelve una representación en string - Método heredado de
Object.prototype

```

Cadena de Prototipos en JavaScript: JavaScript sigue una cadena de prototipos cuando intenta acceder a una propiedad que no está presente en el objeto. Si la propiedad no se encuentra en el objeto actual, el motor de JavaScript busca en el prototipo del objeto, y continúa hasta que encuentra la propiedad o llega al final de la cadena de prototipos.

8.4.2 Modificación de Propiedades:

Cuando se asigna un valor a una propiedad, se modifica directamente la propiedad del objeto en sí, sin afectar las propiedades heredadas:

```

let o = {}; // El prototipo de o es Object.prototype
o.x = 1;

let p = Object.create(o); // p hereda propiedades de o
p["y"] = 0;

let q = Object.create(p); // El prototipo de q es p
q.z = 3;
q.x = 0; // Crea la propiedad x en q, no modifica x en o.

console.log(q); // {z: 3, x: 0} - x es ahora una propiedad directa de q
console.log(o); // {x: 1} - x en o permanece inalterada
console.log(q.x); // 0 - q tiene su propia propiedad x
console.log(o.x); // 1 - x en o no ha sido modificada

```

8.4.3 Acceso a Propiedades Inexistentes:

Cuando se intenta acceder a una propiedad que no existe en un objeto, el resultado es `undefined`. No se lanza una excepción en este caso.

```
console.log(q.a); // undefined - la propiedad a no existe en q
```

Sin embargo, intentar acceder a una propiedad de una propiedad inexistente genera una excepción:

```
q.a.x // Genera un TypeError porque q.a es undefined y no tiene propiedades.
```

8.4.4 Operador Opcional Encadenado (? .):

Para evitar excepciones cuando se accede a propiedades de objetos que podrían no existir, se utiliza el operador opcional encadenado (? .):

```
let apellidos = libro?.autor?.apellidos; // "Sánchez"  
// Si alguna propiedad en la cadena es null o undefined, el resultado será undefined, no se lanza excepción.
```

8.4.5 Eliminación de Propiedades:

Para eliminar una propiedad de un objeto, se utiliza el operador `delete`:

```
console.log(libro.autor.nombre); // "Alfonso"  
delete libro.autor.nombre;  
console.log(libro.autor.nombre); // undefined - la propiedad ha sido eliminada
```

8.4.6 Comprobación de la Existencia de Propiedades:

Existen varias maneras de verificar si una propiedad existe en un objeto:

- Operador `in`:

```
console.log("autor" in libro); // true - La propiedad autor existe en libro  
console.log("toString" in libro); // true - toString existe en Object.prototype, que es el prototipo de libro
```

- Método `hasOwnProperty()`:

```
console.log(libro.hasOwnProperty("autor")); // true - autor es una propiedad directa de libro  
console.log(libro.hasOwnProperty("toString")); // false - toString no es una propiedad directa de libro
```

- Comparación con `undefined`: Este método es menos fiable, porque una propiedad puede existir y tener un valor de `undefined`:

```
libro.autor = undefined;  
console.log(libro.autor !== undefined); // false - propiedad autor existe pero su valor es undefined  
console.log("autor" in libro); // true - la propiedad autor sigue existiendo
```

Este método puede llevar a errores al intentar determinar si una propiedad realmente existe o si simplemente su valor es `undefined`. Por ello, es más seguro utilizar `in` o `hasOwnProperty()` para verificar la existencia de una propiedad.

8.5 Enumeración de Propiedades

Para enumerar las propiedades de un objeto en JavaScript, una de las técnicas más comunes es utilizar el bucle `for-in`. Este bucle recorre todas las propiedades de un objeto, tanto las propias como las heredadas.

```
for (const key in libro) {  
    console.log(key);  
}
```

El bucle `for-in` itera sobre todas las propiedades enumerables de un objeto, incluidas aquellas heredadas a través de la cadena de prototipos. Sin embargo, es importante tener en cuenta que no todas las propiedades se incluyen en esta enumeración. Por ejemplo, las propiedades como `toString`, `valueOf`, y otras que forman parte de `Object.prototype` no aparecen porque son no enumerables.

```
const libro = {  
    titulo: "JavaScript: La Guía Definitiva",  
    autor: "David Flanagan",  
    año: 2020  
};  
  
for (const propiedad in libro) {  
    console.log(propiedad); // título, autor, año  
}
```

En este ejemplo, se listan las propiedades `titulo`, `autor` y `año`, pero no aparecerán otras propiedades heredadas o no enumerables.

8.5.1 Propiedades No Enumerables

Las propiedades no enumerables son aquellas que no se pueden recorrer mediante un bucle `for-in`. Estas propiedades son configuradas con un descriptor especial que establece su no enumerabilidad. Un ejemplo claro de esto son las propiedades como `toString` y `valueOf`, que existen en todos los objetos ya que forman parte de `Object.prototype`, pero no se incluyen en la enumeración.

```
Object.defineProperty(libro, 'editor', {  
    value: "O'Reilly Media",  
    enumerable: false  
});  
  
for (const propiedad in libro) {  
    console.log(propiedad); // título, autor, año (pero no 'editor')
```

```
}
```

En este ejemplo, la propiedad `editor` no aparecerá en la lista porque ha sido definida como no enumerable.

8.5.2 Otras Opciones para Enumerar Propiedades

Si deseas obtener únicamente las propiedades propias del objeto, sin incluir las heredadas, puedes utilizar los métodos `Object.keys()` y `Object.getOwnPropertyNames()`.

- `Object.keys(obj)`: Devuelve un array con las propiedades enumerables propias del objeto.
- `Object.getOwnPropertyNames(obj)`: Devuelve un array con todas las propiedades propias del objeto, incluidas las no enumerables.

```
const propiedadesEnumerables = Object.keys(libro);
console.log(propiedadesEnumerables); // ["titulo", "autor", "año"]

const todasLasPropiedades = Object.getOwnPropertyNames(libro);
console.log(todasLasPropiedades); // ["titulo", "autor", "año", "editor"]
```

En este ejemplo, `Object.keys(libro)` devuelve solo las propiedades enumerables, mientras que `Object.getOwnPropertyNames(libro)` devuelve todas las propiedades propias, incluidas las no enumerables como `editor`.

8.6 Extensión y Clonación de Objetos

En JavaScript, **extender un objeto** significa copiar las propiedades de un objeto a otro. Esto es útil cuando deseas combinar las propiedades de múltiples objetos en un solo objeto o clonar un objeto existente.

```
const origen = { y: 2, z: 3 };
const destino = { x: 1 };
```

Queremos extender las propiedades de `origen` en el objeto `destino`. Una forma sencilla de hacerlo es mediante un bucle `for...in`:

```
for (let key in origen) {
  destino[key] = origen[key];
}
```

Este código recorre todas las propiedades enumerables del objeto `origen` y las copia en el objeto `destino`.

- Usando `Object.keys()`:

Otra forma de lograr lo mismo es utilizando `Object.keys()`, que devuelve un array con las propiedades enumerables de un objeto. Podemos iterar sobre este array con un bucle `for...of`:

```
for (const key of Object.keys(origen)) {  
    destino[key] = origen[key];  
}
```

Aquí usamos `for...of` en lugar de `for...in` porque `Object.keys()` devuelve un array, y `for...of` es ideal para recorrer arrays. Ambos enfoques logran el mismo resultado.

- Usando `Object.assign()`:

JavaScript proporciona una función incorporada, `Object.assign()`, que es una forma más directa y concisa de extender un objeto. Este método copia todas las propiedades enumerables de uno o más objetos de origen a un objeto destino. La sintaxis básica es la siguiente:

```
Object.assign(destino, origen);
```

Esto copia todas las propiedades de `origen` a `destino`. Si necesitas combinar varios objetos, `Object.assign()` también lo permite:

```
Object.assign(nuevoObjeto, destino, origen);
```

En este caso, `nuevoObjeto` contendrá las propiedades combinadas de `destino` y `origen`. Si hay propiedades con el mismo nombre, las propiedades del último objeto (en este caso, `origen`) sobrescribirán las de los objetos anteriores.

- Consideraciones adicionales
 - **Inmutabilidad:** `Object.assign()` modifica el objeto `destino` directamente. Si necesitas mantener la inmutabilidad (es decir, no modificar los objetos originales), puedes crear un nuevo objeto combinando las propiedades de varios objetos:

```
const nuevoObjeto = Object.assign({}, destino, origen);
```

Aquí, `nuevoObjeto` es un nuevo objeto que contiene las propiedades combinadas de `destino` y `origen`, sin modificar los objetos originales.

- **Propiedades no enumerables:** `Object.assign()` solo copia propiedades enumerables. No copia propiedades no enumerables, getters/setters, ni la cadena de prototipos.
- **Métodos modernos:** A partir de ECMAScript 2018, también puedes usar el operador de propagación (...) para combinar objetos de manera aún más concisa:

```
const nuevoObjeto = { ...destino, ...origen };
```

Esto crea un nuevo objeto con las propiedades combinadas de `destino` y `origen`. Es una sintaxis más moderna y generalmente preferida en el código actual.

- **Serialización de Objetos**

Serializar un objeto significa convertirlo en una cadena de texto que puede ser fácilmente almacenada o transmitida. En JavaScript, esto se hace usando [JSON](#):

- Serialización:

```
const data = JSON.stringify(libro);
```

- Deserialización:

```
const copia = JSON.parse(data);
```



La **serialización y deserialización de objetos** es útil cuando necesitas transferir datos entre diferentes sistemas o almacenar datos en bases de datos.

JSON es un formato de texto estándar que se utiliza para representar datos estructurados. Es fácil de leer y escribir para los humanos y fácil de interpretar para las máquinas. Usaremos **JSON** para el intercambio de datos entre el frontend y el backend.

8.7 Propiedades Abreviadas, Computadas y Símbolos

A partir de ES6 (ECMAScript 2015), JavaScript introdujo nuevas funcionalidades para trabajar con objetos de una manera más conveniente y legible. Dos de las más destacadas son las propiedades abreviadas y las propiedades computadas.

1. Propiedades Abreviadas

Las propiedades abreviadas permiten crear objetos de manera más concisa cuando el nombre de la propiedad coincide con el nombre de la variable. En lugar de repetir el nombre de la propiedad y la variable, puedes simplemente escribir el nombre una vez.

```
let x = 1, y = 2;

// Manera tradicional
let o = {
  x: x,
  y: y
};

// Uso de propiedades abreviadas
let z = { x, y };

console.log(z); // Resultado: { x: 1, y: 2 }
```

En el objeto z, no es necesario escribir x: x y y: y porque los nombres de las propiedades coinciden con los nombres de las variables. Es suficiente con escribir { x, y }.

2. Propiedades Computadas

Las propiedades computadas permiten definir los nombres de las propiedades de un objeto de manera dinámica utilizando expresiones. Esto es útil cuando necesitas que el nombre de la propiedad sea el resultado de alguna operación o función.

```
function nombre(i) {
  return "propiedad" + i;
}

let p = {
  nombre: 1 // Aquí, la propiedad se llama literalmente 'nombre'
};

let q = {
  [nombre(1)]: 1 // Aquí, el nombre de la propiedad se calcula como 'propiedad1'
};

console.log(q); // Resultado: { propiedad1: 1 }
```

En el objeto `q`, la propiedad se define utilizando una expresión dentro de corchetes. La función `nombre(1)` retorna el string “propiedad1”, por lo que el objeto `q` tiene una propiedad con el nombre “propiedad1”, cuyo valor es 1.



Usaremos las propiedades computadas cuando queramos que el nombre de una propiedad se calcule en tiempo de ejecución. En React, lo usaremos en la validación de formularios para crear un objeto con los valores de los campos del formulario.

3. Propiedades con Símbolos

En JavaScript, los símbolos (introducidos en ES6) son un tipo de dato primitivo que se utiliza para crear identificadores únicos. Los símbolos pueden ser utilizados como claves de propiedades en objetos, lo que permite definir propiedades que son únicas y no colisionan con otras propiedades, incluso si tienen el mismo nombre.

```
const simbolo = Symbol("Mi nuevo símbolo");

let q = {
  [simbolo]: 1 // El símbolo se utiliza como clave para la propiedad
};

console.log(q); // Muestra un objeto con la propiedad cuya clave es un símbolo
console.log(q[simbolo]); // Resultado: 1
```

8.8 Operador spread en objetos

El operador spread (...) en objetos en JavaScript es una herramienta muy útil para copiar propiedades de un objeto a otro de una manera sencilla y elegante. Este operador se ha convertido en una

alternativa moderna a métodos tradicionales como `Object.assign`.

El operador spread se utiliza para “descomponer” un objeto en sus propiedades individuales. Esto permite crear un nuevo objeto que contenga las propiedades de uno o más objetos originales.

```
let posicion = {x: 0, y: 0};
let tam = {ancho: 100, alto: 100};
let rectangulo = {...posicion, ...tam};
// rectangulo = {x: 0, y: 0, ancho: 100, alto: 100}
```

Manejo de Propiedades Repetidas

Una característica importante del operador spread es cómo maneja las propiedades repetidas al combinar objetos. Las propiedades se añaden de izquierda a derecha, y si una propiedad ya existe, es sobrescrita por la última asignada.

```
let posicion = {x: 0, y: 0};
let tam = {ancho: 100, alto: 100};
let tam2 = {ancho: 200};
let rectangulo = {...posicion, ...tam, ...tam2};
// rectangulo = {x: 0, y: 0, ancho: 200, alto: 100}
```



En React, el operador spread se utiliza frecuentemente para hacer copias de las variables de estado.

El operador spread no crea una copia profunda del objeto, sino que crea una nueva referencia a los mismos datos. Si necesitas una copia profunda, puedes usar `JSON.parse(JSON.stringify(objeto))`.

8.9 Métodos en Objetos

En JavaScript, una propiedad de un objeto puede ser una función. Estas propiedades se denominan métodos. Los métodos pueden definirse de manera tradicional o usando la sintaxis abreviada de ES6:

Método tradicional:

```
let cuadrado = {
  lado: 10,
  area: function() {return this.lado * this.lado;}
};
cuadrado.area();
```

Métodos abreviados:

```
const cuadrado = {
  lado: 10,
  area() {
    return this.lado * this.lado;
}
};
```

```
console.log(cuadrado.area()); // 100
```

9 Arrays

Los **arrays** en JavaScript son estructuras de datos versátiles que permiten almacenar y manipular colecciones de valores. Estos valores pueden ser de distintos tipos, como números, cadenas, objetos, e incluso otros arrays.

- Un array es una colección de elementos ordenados que pueden ser accedidos mediante un índice.
- Índices en un Array: Los índices en un array comienzan desde 0, por lo que el primer elemento se accede mediante `array[0]`. El índice máximo teórico de un array en JavaScript es $2^{32} - 2$ (es decir, 4.294.967.294), aunque en la práctica, los arrays suelen ser mucho más pequeños.
- Propiedad length: Los arrays son objetos especializados en JavaScript que poseen una propiedad `length`, la cual indica el número de elementos en el array. Esta propiedad puede cambiar dinámicamente si se añaden o eliminan elementos.

```
const lista = [1, 2, 3, 4, 5];
console.log(lista.length); // 5
```

- Los arrays en JavaScript son dinámicos, lo que significa que pueden crecer o decrecer en tamaño durante la ejecución del programa.
- Agregar y eliminar elementos: Puedes agregar elementos usando métodos como `push()` o modificar directamente la longitud del array usando la propiedad `length`.

```
const lista = [1, 2, 3];
lista.push(4); // Añade 4 al final del array
console.log(lista); // [1, 2, 3, 4]
lista.length = 2; // Reduce el array a 2 elementos
console.log(lista); // [1, 2]
```

- JavaScript permite crear arrays dispersos, donde algunas posiciones del array pueden no estar definidas (es decir, no tienen valor asignado). A pesar de la dispersión, la propiedad `length` cuenta todas las posiciones, incluyendo las vacías.

```
const disperso = [0, 1, 2, 3, , , , 7, 8, 9];
console.log(disperso.length); // 10
console.log(disperso); // [0, 1, 2, 3, empty × 3, 7, 8, 9]
```

- Los índices de un array son en realidad propiedades del objeto array, y en JavaScript, las propiedades de los objetos son cadenas (strings). Sin embargo, los índices se tratan y comportan como números.

```
const array = [10, 20, 30];
console.log(array["1"]); // 20 (índice como cadena)
console.log(array[1]); // 20 (índice como número)
```

- Los arrays heredan métodos útiles de su prototipo, `Array.prototype`, que incluye funciones como `map()`, `filter()`, `reduce()`, entre otras.

```
const numeros = [1, 2, 3, 4, 5];
const cuadrados = numeros.map(x => x * x);
console.log(cuadrados); // [1, 4, 9, 16, 25]
```

- En JavaScript, las cadenas de caracteres (strings) se comportan de manera similar a los arrays, lo que permite acceder a cada carácter usando un índice.

```
const cadena = "Prueba";
console.log(cadena[0]); // 'P'
```

- Introducidos en ES6, los arrays tipados permiten manejar datos binarios de manera eficiente. Estos arrays tienen un tamaño fijo y solo pueden contener un tipo específico de datos, como enteros (`Int8Array`), enteros sin signo (`UInt8Array`), enteros grandes (`BigUint64Array`), entre otros.
- Son especialmente útiles en aplicaciones que manejan gráficos o datos binarios, como en WebGL.

```
const int8 = new Int8Array([10, 20, 30]);
console.log(int8); // Int8Array [10, 20, 30]
```

9.1 Creación de Arrays

- Literal de Corchetes:** Es la forma más común y simple de crear un array.

```
const vacio = []; // Array vacío
const lista = [1, 2, 3, 4, 5];
```

- Constructor Array:** Permite crear arrays con un tamaño fijo o con elementos iniciales específicos.

```
const milista = new Array(); // Array vacío
const otralista = new Array(100); // Array de 100 posiciones (disperso)
const elementos = new Array(1, 2, 3, "holá"); // Array con elementos iniciales
```

- Método `Array.of()`:** Crea un array con los elementos pasados como argumento, incluso si es uno solo.

```
let g = Array.of(10); // [10]
g = Array.of(1, 2, 3, 4, 5); // [1, 2, 3, 4, 5]
```

- **Método Array.from():** Crea un nuevo array a partir de un objeto iterable o similar a un array, copiando sus elementos.

```
const h = Array.from([1, 2, 3, 4]); // [1, 2, 3, 4]
const i = Array.from("Prueba"); // ['P', 'r', 'u', 'e', 'b', 'a']
```

9.2 Operador Spread (...)

El operador spread permite expandir un array en sus elementos individuales, lo que es útil para combinar arrays o copiar elementos.

```
const a = [1, 2, 3];
const b = [4, 5, 6];
const c = [...a, ...b]; // [1, 2, 3, 4, 5, 6]

const cadena = "Prueba";
const d = [...cadena]; // ['P', 'r', 'u', 'e', 'b', 'a']

let e = a; // Referencia al mismo array
const copia = [...a]; // Copia del array
copia[0] = 100;
console.log(a); // [1, 2, 3] (no cambia)
```

9.3 Leer y escribir elementos

En JavaScript, los arrays son objetos especiales que permiten almacenar múltiples elementos indexados, y como tales, pueden ser manipulados de diversas formas.

- **Acceso a Elementos:** Puedes acceder a un elemento de un array utilizando su índice, que es un número entero positivo. El índice comienza en 0 para el primer elemento.

```
const a = [1, 2, 3, 4, 5];
console.log(a[0]); // 1
```

- **Modificación de Elementos:** Para modificar un valor en un array, simplemente se asigna un nuevo valor al índice correspondiente.

```
a[0] = 0;
console.log(a); // [0, 2, 3, 4, 5]
```

- **Creación de Propiedades Adicionales:** Aunque los arrays son estructuras indexadas, también son objetos en JavaScript, lo que significa que puedes agregar propiedades personalizadas que no afectan la funcionalidad principal del array.

```
a.prop = true; // Se añade una propiedad llamada 'prop'
a.first = function() { return this[0]; };
a.last = function() { return this[this.length - 1]; };
```

```
console.log(a.first()); // 0
console.log(a.last()); // 5
```

- **Acceso a Índices No Existentes:** Si intentas acceder a un índice que no existe en el array, JavaScript no lanza una excepción, sino que retorna undefined.

```
console.log(a[100]); // undefined
```

- En JavaScript, los índices de un array son técnicamente propiedades de objeto, y por lo tanto, pueden ser accedidos también como cadenas de texto.

```
console.log(a["0"]); // 0
```

- **Índices Negativos o No Enteros:** Asignar valores a índices negativos o no enteros no cambia el tamaño del array, pero sí crea propiedades adicionales.

```
a[-1] = 0;
console.log(a[-1]); // 0
console.log(a.length); // 5 (el tamaño del array no cambia)
```

• Verificación de Existencia de Elementos

Puedes utilizar el operador in para verificar si un índice existe en el array.

```
console.log(0 in a); // true
console.log(100 in a); // false
```

- **Uso en Arrays Dispersos:** Este operador es particularmente útil en arrays dispersos, donde algunas posiciones pueden no estar definidas.

```
const disperso = [1, 2, , , 5];
console.log(2 in disperso); // false
console.log(4 in disperso); // true
```

- **Propiedad length:** La propiedad length de un array indica cuántos elementos tiene el array, pero también puede ser modificada manualmente para cambiar el tamaño del array.

```
const b = Array(10);
console.log(b.length); // 10
```

- Al modificar length, puedes aumentar o reducir el tamaño del array. Si reduces el tamaño, los elementos que queden fuera del nuevo límite se eliminan.

```
b[6] = "h";
console.log(b); // [ <6 empty items>, 'h', <3 empty items> ]
b.length = 20; // Aumenta el tamaño
console.log(b); // [ <6 empty items>, 'h', <13 empty items> ]
b[19] = "cadena";
console.log(b); // 'cadena' está en la última posición
b.length = 10; // Reduce el tamaño, eliminando 'cadena'
console.log(b); // [ <6 empty items>, 'h', <3 empty items> ]
```

- **Añadir Elementos:**

- `push()`: Añade uno o más elementos al final del array.
- `unshift()`: Añade uno o más elementos al inicio del array.

```
const a = [1, 2, 3, 4];
a.push(5); // [1, 2, 3, 4, 5]
a.unshift(0); // [0, 1, 2, 3, 4, 5]
```

- **Eliminar Elementos:**

- `pop()`: Elimina y retorna el último elemento del array.
- `shift()`: Elimina y retorna el primer elemento del array.

```
const ultimo = a.pop(); // Elimina 5, retorna 5
const primero = a.shift(); // Elimina 0, retorna 0
console.log(a); // [1, 2, 3, 4]
```

- **El operador delete:** en JavaScript se puede usar para eliminar elementos de un array, pero es importante notar que este operador no modifica el tamaño del array. En su lugar, simplemente elimina el valor en la posición especificada, dejando un espacio vacío (`undefined`) en su lugar.

```
const a = [1, 2, 3, 4, 5];
delete a[2]; // Elimina el valor en la posición 2
console.log(a); // [1, 2, undefined, 4, 5]

delete a[a.length - 1]; // Elimina el último valor
console.log(a.length); // 5 (el tamaño sigue siendo 5)
```

- Cuando se trabaja con arrays dispersos, es útil saber cómo iterar sobre ellos sin procesar los índices vacíos. Aquí se muestra cómo hacerlo usando un `for-of` loop y el operador spread:

```
const c = [1, 2, 3, , , 5, 6];

// Iteración con for-of ignorando los índices vacíos
for (const value of c) {
    if (value !== undefined) {
        console.log(value); // Solo imprimirá 1, 2, 3, 5, 6
    }
}

// Usando el operador spread con c.entries()
const entries = [...c.entries()];
console.log(entries); // Muestra los índices y valores, incluyendo los undefined

// Recorriendo c.entries() y desestructurando key y value
for (const [key, value] of c.entries()) {
    if (value !== undefined) {
        console.log(`Key: ${key}, Value: ${value}`);
    }
}
```

9.4 Arrays Multidimensionales: Matriz Dispersa de 10x10

Ahora vamos a crear una matriz dispersa de 10x10, donde cada elemento será la suma de su fila y columna.

```
const matrix = Array(10).fill().map(() => Array(10).fill());
for (let i = 0; i < 10; i++) {
  for (let j = 0; j < 10; j++) {
    matrix[i][j] = i + j;
  }
}
console.log(matrix);
```

Explicación de la Matriz Dispersa

- **Creación de la Matriz Dispersa:**
Usamos `Array(10).fill()` para crear un array de 10 elementos, todos inicialmente undefined
`map(() => Array(10).fill())` convierte cada elemento en un array de 10 elementos undefined, formando así una matriz dispersa.
- **Relleno de la Matriz:**
Recorremos cada fila `i` y cada columna `j`, asignando a cada posición `matrix[i][j]` la suma de `i + j`.
- **Resultado:**
El resultado es una matriz de 10x10 donde cada elemento es la suma de su índice de fila y columna.

9.5 Métodos iteradores de Array

Para ver los métodos disponibles en un array, puedes acceder al prototipo del array mediante `Object.getPrototypeOf([])` y compararlo con `Array.prototype`.

```
console.log(Object.getPrototypeOf([]) === Array.prototype); // true
```

En JavaScript, los métodos iteradores de arrays permiten realizar operaciones sobre cada elemento de un array. Estos métodos toman una función como argumento y la ejecutan para cada elemento del array. Es importante notar que en arrays dispersos, estos métodos no se ejecutan para los elementos vacíos.

1. **forEach:** Iterar sobre cada elemento

`forEach` ejecuta una función para cada elemento del array, pero no retorna un nuevo array. No se ejecuta para elementos vacíos en un array disperso.

```
const datos = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
let suma = 0;
datos.forEach((valor) => { suma += valor; });
console.log(suma); // 55
```

```
datos.forEach((valor, indice, array) => {
    array[indice] = valor * 2;
});
console.log(datos); // [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

2. **map:** Crear un nuevo array transformado

map crea un nuevo array con los resultados de llamar a la función proporcionada en cada elemento del array original.

```
const nuevo = datos.map((valor) => valor / 2);
console.log(nuevo); // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

3. **filter:** Filtrar elementos según una condición

filter crea un nuevo array con todos los elementos que pasen la condición implementada en la función proporcionada.

```
const menores = datos.filter((valor) => valor <= 5);
console.log(menores); // [2, 4, 6, 8, 10]

const pares = datos.filter((valor) => valor % 2 === 0);
console.log(pares); // [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

4. **find y findIndex:** Buscar elementos

find retorna el primer elemento que cumple la condición especificada en la función proporcionada. findIndex retorna el índice del primer elemento que cumple la condición.

```
const buscado = datos.find((valor) => valor < 5);
console.log(buscado); // 2

const indiceBuscado = datos.findIndex((valor) => valor < 5);
console.log(indiceBuscado); // 0
```

5. **every y some:** Verificar condiciones

every verifica si todos los elementos del array cumplen una condición y retorna true o false. some verifica si al menos un elemento cumple la condición.

```
const todos = datos.every((valor) => valor > 10);
console.log(todos); // false

const alguno = datos.some((valor) => valor > 10);
console.log(alguno); // true
```

6. **reduce y reduceRight:** Reducir un array a un solo valor

reduce aplica una función a un acumulador y cada valor del array (de izquierda a derecha) para reducirlo a un solo valor. reduceRight hace lo mismo pero de derecha a izquierda.

```
const sumaReduce = datos.reduce((acumulador, valor) => acumulador + valor, 0);
console.log(sumaReduce); // 110

const multiplicacion = datos.reduce((acumulador, valor) => acumulador * valor, 1);
console.log(multiplicacion); // 3715891200

const mayor = datos.reduce((max, valor) => (valor > max ? valor : max));
console.log(mayor); // 20
```

7. flat y flatMap: Aplanar arrays

flat aplana arrays anidados en un solo nivel o más dependiendo del parámetro. flatMap primero aplica map a cada elemento y luego aplana el resultado en un nuevo array.

```
const arrAnidado = [1, 2, 3, [4, 5, 6], [7, 8, [9, 10]]];
console.log(arrAnidado.flat(2)); // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

const frases = ["Arrays en Javascript", "Aprendiendo Javascript"];
const palabras = frases.flatMap((frase) => frase.split(" "));
console.log(palabras); // ['Arrays', 'en', 'Javascript', 'Aprendiendo', 'Javascript']
```



Es importante dominar los **métodos iteradores de arrays**, ya que son fundamentales para un estilo de programación funcional y para el manejo de datos en general.

9.6 Otros Métodos Útiles de Arrays

Además de los iteradores, hay otros métodos de arrays que puedes probar:

- `concat`: Combina dos o más arrays.
- `slice`: Retorna una copia de una parte del array.
- `splice`: Añade o elimina elementos en un array.
- `fill`: Rellena todos los elementos de un array con un valor estático.
- `copyWithin`: Copia una parte del array a otra ubicación en el mismo array.
- `indexOf`: Retorna el primer índice en el que se encuentra un elemento.
- `lastIndexOf`: Retorna el último índice en el que se encuentra un elemento.
- `includes`: Verifica si un array contiene un elemento.
- `sort`: Ordena los elementos de un array.
- `reverse`: Invierte el orden de los elementos en un array.
- `join`: Une todos los elementos de un array en una cadena.
- `toString`: Convierte el array a una cadena.
- `Array.isArray`: Verifica si un valor es un array.

10 Desestructuración

La **desestructuración en JavaScript** es una característica del lenguaje introducida en ECMAScript 6 (ES6) que permite extraer valores de arrays o propiedades de objetos y asignarlos a variables de manera más concisa y clara. Esto es particularmente útil para trabajar con estructuras complejas de datos, como objetos anidados o arrays multidimensionales. A continuación, se explica la desestructuración en detalle para ambos casos: arrays y objetos.

10.1 Desestructuración de Arrays

La desestructuración de arrays permite extraer valores de un array y asignarlos a variables de manera directa. La sintaxis básica es:

```
const array = [1, 2, 3, 4];
// Desestructuración
const [a, b, c] = array;

console.log(a); // 1
console.log(b); // 2
console.log(c); // 3
```

10.1.1 Características clave:

- **Asignación por posición:** Los valores son asignados a las variables en función de su posición en el array. En el ejemplo anterior, a toma el valor del primer elemento del array, b del segundo, y así sucesivamente.
- **Valores por defecto:** Puedes asignar valores por defecto a las variables en caso de que los elementos del array sean undefined.

```
const array = [1, 2];
const [a, b, c = 3] = array;

console.log(a); // 1
console.log(b); // 2
console.log(c); // 3
```

- **Omisión de valores:** Si no te interesa algún valor intermedio del array, puedes omitirlo usando una coma.

```
const array = [1, 2, 3, 4];
const [a, , c] = array;

console.log(a); // 1
console.log(c); // 3
```

- **Rest operator:** Puedes capturar el resto de los elementos del array en una variable utilizando el operador rest (...).

```
const array = [1, 2, 3, 4];
const [a, b, ...rest] = array;

console.log(a); // 1
console.log(b); // 2
console.log(rest); // [3, 4]
```

10.2 Desestructuración de Objetos

La desestructuración de objetos permite extraer propiedades de un objeto y asignarlas a variables. La sintaxis básica es:

```
const objeto = { x: 1, y: 2, z: 3 };

// Desestructuración
const { x, y } = objeto;

console.log(x); // 1
console.log(y); // 2
```

10.2.1 Características clave:

- **Asignación por nombre:** A diferencia de los arrays, la desestructuración de objetos se basa en los nombres de las propiedades. En el ejemplo anterior, x y y son asignados a las variables correspondientes con los mismos nombres.
- **Asignación a nuevos nombres de variables:** Puedes asignar las propiedades a variables con nombres diferentes.

```
const objeto = { x: 1, y: 2 };
const { x: a, y: b } = objeto;

console.log(a); // 1
console.log(b); // 2
```

- **Valores por defecto:** Similar a los arrays, puedes definir valores por defecto para propiedades que no existen o son undefined.

```
const objeto = { x: 1 };
const { x, y = 2 } = objeto;

console.log(x); // 1
console.log(y); // 2
```

- **Desestructuración anidada:** Puedes desestructurar objetos anidados.

```
const objeto = {
  a: 1,
  b: {
    c: 2,
    d: 3
  }
};

const { b: { c, d } } = objeto;

console.log(c); // 2
console.log(d); // 3
```

- **Rest operator:** También es posible usar el operador rest para capturar el resto de las propiedades en un nuevo objeto.

```
const objeto = { x: 1, y: 2, z: 3 };
const { x, ...resto } = objeto;

console.log(x); // 1
console.log(resto); // { y: 2, z: 3 }
```

10.3 Aplicaciones Prácticas

La desestructuración es muy útil en diversas situaciones, como:

- **Intercambio de valores:** Intercambiar valores entre dos variables sin una variable temporal.

```
let a = 1, b = 2;
[a, b] = [b, a];
console.log(a); // 2
console.log(b); // 1
```

- **Extracción de datos de funciones:** Cuando una función retorna un objeto, puedes desestructurarlo directamente en la llamada.

```
function obtenerCoordenadas() {
  return { x: 10, y: 22 };
}

const { x, y } = obtenerCoordenadas();
console.log(x, y); // 10 22
```

- **Parámetros de función:** Puedes desestructurar directamente en los parámetros de una función.

```
function imprimirCoordenadas({ x, y }) {
  console.log(`Coordenadas: ${x}, ${y}`);
}

imprimirCoordenadas({ x: 10, y: 22 });
```

11 Funciones

En JavaScript, las funciones son bloques fundamentales de código que permiten encapsular lógica reutilizable. Existen varias formas de declarar y utilizar funciones, cada una con sus propias características y usos específicos.

La forma más común de declarar una función en JavaScript es utilizando la palabra clave `function`. Aquí un ejemplo básico:

```
function suma(x, y) {
    return x + y;
}
const s = suma(349, 123);
imprime(s);

function imprime(msg) {
    console.log(msg);
    // En ausencia de un return explícito, JavaScript retorna undefined por defecto.
}
```

11.1 Hoisting en Funciones

El **hoisting** es un comportamiento en JavaScript donde las declaraciones de funciones y variables se mueven al inicio de su contexto de ejecución. Esto significa que puedes invocar una función antes de su declaración:

```
console.log(suma(5, 7)); // Funciona debido al hoisting

function suma(x, y) {
    return x + y;
}
```

En este ejemplo, la función `suma` se puede llamar antes de su declaración porque JavaScript eleva su definición al comienzo del ámbito.

11.2 Funciones como Expresiones

En JavaScript, las funciones también pueden ser expresiones, lo que significa que pueden asignarse a variables o pasarse como argumentos a otras funciones:

```
const cuadrado = function(x) {
    return x * x;
};
console.log(cuadrado(8));
```

En este caso, hemos creado una función anónima (una función sin nombre) y la hemos asignado a la variable `cuadrado`. Esta función no está sujeta al hoisting, por lo que no puede ser llamada antes de su definición.

```

function calcularAreaRectangulo(base, altura) {
    return base * altura;
}

function calcularAreaTriangulo(base, altura) {
    return (base * altura) / 2;
}

function calcularArea(figura, base, altura, calcularAreaFunc) {
    console.log(`El área del ${figura} es: ${calcularAreaFunc(base, altura)} `);
}

calcularArea('rectángulo', 5, 10, calcularAreaRectangulo); // El área del rectángulo es: 50
calcularArea('triángulo', 5, 10, calcularAreaTriangulo); // El área del triángulo es: 25

```

En este ejemplo, la función `calcularArea` acepta una función como argumento. Esto permite que la función `calcularArea` pueda operar con diferentes tipos de figuras geométricas sin tener que modificar su propia implementación.

11.3 Funciones Anónimas y Recursividad

A veces, incluso una función expresada puede tener un nombre, particularmente en casos de recursividad:

```

const f = function factorial(x) {
    if (x <= 1) return 1;
    else return x * factorial(x - 1);
};

console.log(f(5)); // 120

```

En este ejemplo, aunque `factorial` es el nombre dentro de la función, la función es llamada a través de `f`.

11.4 Funciones Arrow (Funciones Flecha)

Introducidas en ES6, las funciones flecha (arrow functions) proporcionan una sintaxis más concisa:

```

const suma1 = (x, y) => {
    return x + y;
};

console.log(suma1(1, 3)); // 4

// Simplificando aún más:
const suma2 = (x, y) => x + y;
console.log(suma2(4, 7)); // 11

// Con un solo parámetro:
const polinomio = x => x * x + 2 * x;
console.log(polinomio(2)); // 8

// Sin parámetros:
const dospí = () => 3.1415 * 2;
console.log(dospí()); // 6.283

```

11.4.1 Consideraciones sobre `this` en Funciones Arrow

Una característica clave de las funciones arrow es que no tienen su propio contexto `this`; en su lugar, heredan `this` del contexto donde se definieron:

```
let o = {
  m: function () {
    console.log(this === o); // true
    const f = () => {
      console.log(this === o); // true
    };
    f();
  }
};
o.m();
```

Por esta razón, las funciones arrow son especialmente útiles en funciones anidadas para evitar el uso de trucos como `const that = this`.

```
let o = {
  m: function () {
    let that = this;
    console.log(this === o);
    f(); //hoisting
    function f() {
      console.log(this === o); //false. Vemos que no se hereda el valor de this en la
      //función anidada.
      console.log(this === global); //this es el objeto global
      console.log(that === o); // truco para poder usar o en funciones anidadas.
    }
  }
};
```

Uso Inadecuado de Funciones Arrow en Objetos

Sin embargo, debido a cómo manejan `this`, no se recomienda usar funciones arrow como métodos en objetos:

```
const calculadora = {
  op1: 100,
  op2: 200,
  suma: () => this.op1 + this.op2 // Esto no funcionará como se espera
};
```

Aquí, `this` no se refiere al objeto `calculadora`, sino al contexto donde fue definida la función, lo que podría ser el objeto global o `undefined` en modo estricto.

11.4.2 Parámetros en Funciones

11.4.2.1 Parámetros Opcionales y Valores por Defecto En ES6, es posible asignar valores por defecto a los parámetros de una función:

```
function params(a, b = [], c = 10) {
```

```

        console.log(a, b, c);
    }
params(100); // 100 [] 10

```

Esto es útil para definir comportamientos predeterminados cuando no se pasan ciertos argumentos.

11.4.2.2 Operador Rest El operador rest (...) permite a una función aceptar un número indefinido de argumentos como un array:

```

function max(first = -Infinity, ...rest) {
    let maxValue = first;
    for (let n of rest) {
        if (n > maxValue) maxValue = n;
    }
    return maxValue;
}
console.log(max(1, 2, 3, 4, 5, 6, 7)); // 7

```

En este ejemplo, rest recoge todos los argumentos adicionales, lo que permite a la función trabajar con una cantidad variable de entradas.

11.5 Objeto arguments

Antes de ES6, el objeto `arguments` permitía acceder a todos los parámetros pasados a una función:

```

function max() {
    let maxValue = -Infinity;
    for (let i = 0; i < arguments.length; i++) {
        if (arguments[i] > maxValue) maxValue = arguments[i];
    }
    return maxValue;
}
console.log(max(1, 2, 3)); // 3

```

Aunque `arguments` sigue siendo útil, es menos eficiente y claro que el operador rest.

11.6 Operador Spread

El operador spread (...) permite expandir un array en argumentos individuales para una función:

```

let numbers = [1, 2, 3, 4, 5, 6];
let nmin = Math.min(...numbers);
console.log(nmin); // 1

let nmax = Math.max(...numbers);
console.log(nmax); // 6

```

Esto simplifica mucho la manipulación de arrays cuando se trabaja con funciones que requieren múltiples argumentos.

11.7 Desestructuración de Argumentos

La desestructuración permite extraer valores de arrays u objetos directamente en los parámetros de una función:

```
function vectorSuma([x1, y1], [x2, y2]) {  
    return [x1 + x2, y1 + y2];  
}  
console.log(vectorSuma([1, 2], [3, 4])); // [4, 6]  
  
function toHtml({ titulo, contenido }) {  
    return `<h1>${titulo}</h1><article>${contenido}</article>`;  
}  
const html = toHtml({ titulo: 'Javascript', contenido: 'Es muy importante saber Javascript',  
    color: 'red', autor: 'Antonia' });  
console.log(html);
```

Este enfoque hace el código más legible y reduce errores relacionados con el acceso manual a las propiedades de objetos y elementos de arrays.



El uso de la desestructuración de argumentos es muy habitual en los frameworks de JavaScript modernos como React, al crear componentes funcionales.

12 Ejercicios I de strings



Los siguientes ejercicios están resueltos en el siguiente apartado. Antes de mirar las soluciones **intenta resolverlos por ti mismo**.

- a. Crea una cadena multilínea usando comillas dobles.
- b. Agrega a la cadena un retorno de carro y tabuladores utilizando los símbolos de escape correspondientes.
- c. Incluye en la cadena el carácter \.
- d. Concatena otra cadena utilizando el operador +.
- e. Concatena cadenas usando un template string. Muestra el valor de varias variables en el template string.
- f. Separa un texto que contenga varias frases en un array, donde cada elemento del array sea una frase.

- g. Convierte un texto dado a minúsculas.
- h. Convierte un texto dado a mayúsculas.
- i. Recorre el texto carácter por carácter usando un bucle e imprímelo.
- j. Busca una subcadena dentro de un texto.
- k. Extrae una subcadena desde la posición 3 hasta el final del texto y guárdala en una variable.
- l. Extrae una subcadena desde la posición 3 hasta la primera ocurrencia de una palabra en el texto y guárdala en una variable.
- m. Reemplaza todos los espacios en el texto por un guion -.
- n. Elimina los espacios en blanco antes y después del texto.
- o. Crea una cadena que no contenga ningún espacio partiendo de otra cadena dada.
- p. Crea una función que invierta una cadena de texto.
- q. Usa una expresión regular para comprobar si la cadena contiene números.
- r. Usa una expresión regular para comprobar si la cadena termina en un punto.
- s. Usa una expresión regular para comprobar si la cadena comienza con una mayúscula.
- t. Usa una expresión regular para comprobar si la cadena contiene un número de teléfono con código internacional.
- u. Reemplaza cualquier ocurrencia de un + seguido de números por la cadena “SECRETO”.

13 Ejercicios II sobre desestructuración

- a. Desestructura el día, mes y año e imprime la fecha dado el array ['06', 'Octubre', '2021'].
- b. Dado un array de números, desestructura los números en posiciones impares.
- c. Desestructura el primer número, el segundo y el resto en otra variable.
- d. Desestructura nombre, apellidos y teléfono del siguiente objeto:

```
const person = { nombre: 'Luis', apellidos: 'Molina', telefono: '+34666554433' };
```

- e. Cambia el siguiente bucle para desestructurar cada entrada e imprimir llave y valor por separado:

```
for (const [key, value] of Object.entries(person)) {  
    console.log(key, value);  
}
```

- f. Dado `[{x: 1, y: 2}, {x: 3, y: 4}]`, desestructura los puntos en las variables `x1, y1, x2, y2`.
- g. Crea una función a la que le pasas un único objeto como parámetro con 5 propiedades cualesquiera, incluida la propiedad nombre y apellidos. Desestructura en la función la propiedad nombre y apellidos e imprime el nombre completo:
- h. Dados dos objetos, combínalos en uno solo utilizando el operador spread. Después, elimina alguna de las propiedades:
- i. Crea una función que retorna un array con 3 valores. Usa la función y desestructura los valores:
- j. Realiza una clonación profunda de un objeto que contiene otros objetos o arrays como propiedades:

14 Ejercicios III sobre arrays

- a. Crea un array “datos” vacío con un literal.
- b. Añade a “datos” los números del 1 al 50 con un bucle for.
- c. Elimina los elementos del 25 al 50 asignando un nuevo tamaño a la propiedad length.
- d. Usa el operador spread para hacer una copia del array anterior.
- e. Crea un array de tamaño 50 con el constructor Array.
- f. Copia el array anterior a otro con la factoría from.
- g. Crea un array multidimensional de 10 filas (i) y 10 columnas (j). Inicializa cada celda con el valor $i*j$.
- h. Crea un array con la factoría of con los números del 1 al 5. Después, añade un elemento en la posición 10 y otro en la 50. Recorre el array con un for imprimiendo los valores, y después con forEach. ¿Cuál es la diferencia? ¿Cuál es el tamaño del array?
- i. Elimina dos elementos con delete.
- j. Calcula el producto de todos los números del array “datos” con forEach.
- k. Cada elemento x del array “datos” debe cambiarse por $x*x$. Usa forEach.
- l. Crea un nuevo array con map recorriendo cada elemento x de “datos”, donde cada elemento sea un string “El valor es: x”. Usa template strings.
- m. Crea un nuevo array mediante map que incremente cada elemento de “datos” en 5 unidades.
- n. Mediante filter, quédate con los números impares en un nuevo array impares.
- o. Usa find para buscar el número 13.

- p. Usa every para comprobar si todos los números son positivos.
 - q. Calcula la sumatoria del array “datos” mediante reduce.
 - r. Calcula el valor más pequeño del array mediante reduce.
 - s. Usa flat para aplanar el array multidimensional que creaste anteriormente.
 - t. Tenemos la cadena: “Vamos a usar flatMap. Es igual que map. Pero aplana los arrays”. Separa mediante split las distintas frases. Después, mediante map, quita los espacios sobrantes (trim). A continuación, usa flatMap para extraer todas las palabras de cada frase en un único array.
 - u. Crea el array a = [1,2,3,4,5] y b = [6,7,8,9,10] con literales. Concatena los arrays a y b con concat. Después, usa el operador spread. Crea una variable const cola. Usa unshift y shift para añadir y quitar elementos. Dado el array resultante de la concatenación de a y b, obtén el subarray desde el índice 2 hasta el penúltimo elemento (slice). Usa splice para quitar los 2 últimos elementos de un array.
 - v. Rellena con fill un array de 100 elementos con -1.
 - w. Crea un array de cadenas. Busca con indexOf una cadena.
 - x. Comprueba si la cadena “hola” está dentro del array anterior.
 - y. Ordena la lista de cadenas anterior de forma alfabética con sort.
 - z. Crea un array vacío de 50 posiciones. Con forEach, asigna valores aleatorios entre 0 y 100. Después, ordena con sort de menor a mayor. Cambia y ordena de mayor a menor.
- aa. Usa reverse para invertir el array anterior.

15 Ejercicios IV sobre lógica de programación

1. Crea una función que cuente el número de vocales de una cadena de caracteres.
2. Crea una función que determine si una cadena es un palíndromo, es decir, que se lee igual hacia delante que hacia atrás.
3. Crea una función que capitalice una cadena de texto. Es decir que todas las palabras empiecen por mayúscula.
4. Dado un array de cadenas y una longitud n, crea una función que filtre el array dejando solo las cadenas de menor longitud que n.
5. Crea una función que cree el acrónimo de una cadena de caracteres tomando la primera letra de cada palabra y convirtiéndola a mayúscula. Por ejemplo la frase anterior sería CUFQCEADU....

6. Crea una función que cuente las frases, palabras y letras presentes en un texto.
7. Crea una función que identifique si hay elementos duplicados en un array.
8. Crea una función que debe retornar verdadero si alguno de los elementos de un array está repetido n veces.
9. Crea un array que intercale dos arrays dados. Por ejemplo dados [a,b,c,d] y [1,2,3,4] el resultado sería [a,1,b,2,c,3,d,4]
10. Crea una función que rote los elementos de un array n posiciones. Por ejemplo, dado el array [1,2,3,4,5,6] y el número 2 el resultado será: [5,6,1,2,3,4]
11. Crea una función que elimine de una cadena los caracteres dados en un array.
12. Crea una función que rote una matriz de tamaño nxn, 90 grados a la derecha. Ejemplo: [1,2,3] [7, 4, 1] [4,5,6] => [8, 5, 2] [7,8,9] [9, 6, 3]
13. Crea una función que determine si los paréntesis presentes en una cadena de texto están balanceados. Por ejemplo (a(b)) → Balanceado, (a(b(a))) → No balanceado.
14. Busca una submatriz dentro de una matriz más grande. El resultado debe ser las coordenadas donde se encuentra dicha matriz.
15. Crea una función que verifique si una matriz de 9x9 es una solución de un sudoku. Una cuadrícula válida de Sudoku es aquella que cumple las siguientes condiciones:
 - a. Filas Únicas: Cada fila debe contener los números del 1 al 9 sin repetición.
 - b. Columnas Únicas: Cada columna debe contener los números del 1 al 9 sin repetición.
 - c. Subcuadrículas Únicas: Cada una de las nueve subcuadrículas de 3x3 debe contener los números del 1 al 9 sin repetición.

16 Ejercicios V de Javascript

1. Crea un programa que genere un array con 1000 números aleatorios del 0 al 99.
 - a. Crea una función que calcule la media aritmética.
 - b. Calcula la frecuencia de cada número del 0 al 99. Es decir, si el número 0 aparece 80 veces en el array de 1000 posiciones, se guardará un 80 en la posición 0 del nuevo array. Si el número 1 aparece 50 veces, se guardará un 50 en la posición 1, etc.
 - c. Crea una función que ordene el array de menor a mayor sin usar métodos de Javascript.
 - d. Ahora, usa una función de Javascript para realizar la ordenación.
2. Crea una función que calcule el factorial de un número usando un bucle.

3. Crea una función que busque todas las ocurrencias de una palabra en un texto dado. La función retorna un array con las posiciones encontradas.
4. El programa FizzBuzz se ha usado habitualmente para descartar candidatos en pruebas de selección de personal. Consiste en escribir un programa que muestre en pantalla los números del 1 al 100, sustituyendo los múltiplos de 3 por la palabra “fizz”, los múltiplos de 5 por “buzz”, y los múltiplos de ambos, es decir, los múltiplos de 3 y 5 (o de 15), por la palabra “fizzbuzz”.
5. Crea una página que genere un acertijo matemático simple formado por sumas y restas. El usuario inserta la respuesta y el script le indica si ha acertado o no.
6. Crea una página web que genere contraseñas aleatorias con una mezcla de letras, números y símbolos. La página tiene un campo de entrada numérico en el que el usuario especifica el tamaño de la contraseña a generar. La contraseña se mostrará al pulsar un botón.
7. Implementa un script que cuente el número de vocales y consonantes en una cadena.
8. Crea una página web con un formulario que contenga los campos nombre, correo electrónico y edad. Cuando el usuario pulse el botón enviar, se validará el formulario comprobando que el nombre tiene más de 3 caracteres, la edad es un número entre 0 y 120, y el email se valida mediante una expresión regular.
9. Implementa el cifrado César, un tipo de cifrado por sustitución en el que cada letra en el texto original es reemplazada por otra que se encuentra un número fijo de posiciones más adelante en el alfabeto. Crea una página web con un área de texto que contenga el texto original. Al pulsar un botón, se mostrará debajo, en otro área de texto, el texto cifrado.
10. Crea una página web con un cronómetro que muestre minutos y segundos. Deberá contar con los botones Comenzar, Parar y Reiniciar. Ayúdate de las funciones setInterval, clearInterval, y setTimeout.
11. Crea una página web que imprima, mediante un script, la tabla de multiplicar especificada por el usuario al cambiar un valor en un desplegable.
12. Crea una página web para jugar a Piedra, Papel o Tijera contra la máquina. Gana el primero que gane 2 de 3 jugadas. En cualquier momento, el usuario puede interrumpir la partida e iniciar una nueva.