

---


# JavaScript: El Lenguaje del Navegador

## 1. Variables y Constantes

JavaScript es dinámico y flexible cuando se trata de variables. Vamos a profundizar:

- `let` y `const` son las formas modernas de declarar variables en JavaScript, introducidas con ECMAScript 6 (ES6). A diferencia de `var`, que tiene un alcance de función, `let` y `const` tienen un **alcance de bloque** (funcionan dentro de llaves `{}`).
- `let`: Se usa cuando la variable puede cambiar su valor.


javascript

 Copiar código

```
let edad = 25;  
edad = 26; // Aquí cambiamos el valor de 'edad'
```

- `const`: Se usa para variables cuyo valor no cambia. Ideal para valores fijos como **constantes** o referencias a objetos que no cambian.


javascript

 Copiar código

```
const PI = 3.14159;  
// PI no puede cambiar su valor
```

**Tip importante:** En JavaScript, aunque no puedes reasignar una constante, si es un objeto o array, sus propiedades internas sí pueden cambiar:

javascript

 Copiar código

```
const persona = { nombre: "Andrea", edad: 25 };  
persona.edad = 26; // Esto es válido
```


## 2. Manipulación del DOM (Document Object Model)

El DOM es la estructura jerárquica que representa el contenido HTML de una página web. JavaScript nos permite **interactuar y modificar** esta estructura, lo que nos da un control total sobre la apariencia y el comportamiento de una web.

- **Acceder a elementos del DOM:** Existen varios métodos útiles para seleccionar elementos de la página:

- `document.getElementById('id')` : Selecciona un elemento por su `id`.


javascript

 Copiar código

```
const titulo = document.getElementById('miTitulo');  
console.log(titulo.textContent); // Muestra el texto dentro del elemento
```

- `document.querySelector('selector')` : Selecciona el **primer** elemento que coincida con el selector CSS.


javascript

 Copiar código

```
const parrafo = document.querySelector('.miClase');
```

- `document.querySelectorAll('selector')` : Selecciona **todos los elementos** que coincidan con el selector CSS.

javascript


 Copiar código

```
const todosLosParrafos = document.querySelectorAll('p');  
todosLosParrafos.forEach(parrafo => console.log(parrafo.textContent));
```

- **Modificar contenido:** Puedes modificar texto, atributos y estilos de los elementos.

- Cambiar el texto de un elemento:


javascript

 Copiar código

```
const parrafo = document.getElementById('miParrafo');  
parrafo.textContent = "Este es un nuevo texto";
```

- Modificar atributos (como el `src` de una imagen):


javascript

 Copiar código

```
const imagen = document.querySelector('img');  
imagen.src = 'nueva-imagen.jpg';
```

- Cambiar estilos:

javascript


 Copiar código

```
const boton = document.getElementById('miBoton');  
boton.style.backgroundColor = 'blue';  
boton.style.color = 'white';
```

- Agregar y quitar clases: Las clases CSS permiten aplicar estilos de manera dinámica.

- Añadir una clase:


javascript

 Copiar código

```
boton.classList.add('activo');
```

- Quitar una clase:


javascript

 Copiar código

```
boton.classList.remove('activo');
```

- Alternar una clase (toggle):

javascript

 Copiar código

```
boton.classList.toggle('activo'); // La añade si no está, la quita si está
```

### 3. Eventos en JavaScript

Los eventos permiten hacer que una página web sea interactiva. Un **evento** es una acción como un clic, un teclado pulsado o el envío de un formulario.

- **Agregar un evento:** Los eventos se manejan con el método `addEventListener()`. Este método escucha un evento específico en un elemento y ejecuta una función cuando el evento ocurre.
  - **Clic en un botón:**

javascript

Copiar código

```
const boton = document.getElementById('miBoton');
boton.addEventListener('click', () => {
  alert("¡Botón clickeado!");
});
```

- **Tipos de eventos más comunes:**
  - `click`: Cuando un usuario hace clic en un elemento.
  - `mouseover`: Cuando el ratón pasa por encima de un elemento.
  - `keydown`: Cuando una tecla es presionada.
  - `submit`: Cuando se envía un formulario.

Ejemplo:

javascript

Copiar código

```
const formulario = document.querySelector('form');
formulario.addEventListener('submit', (evento) => {
  evento.preventDefault(); // Previene el comportamiento por defecto (enviar el for
  console.log('Formulario enviado');
});
```


### 4. Programación Orientada a Objetos (POO) en JavaScript

La **POO** permite crear objetos que contienen tanto **datos** como **comportamientos** (métodos) asociados a esos datos. JavaScript no siempre fue un lenguaje orientado a objetos en su esencia, pero con la introducción de **clases** en ES6, ahora podemos estructurar código de manera mucho más clara y modular.

- **Clases y objetos:** Una clase es como un molde que define cómo serán los objetos creados a partir de ella. Cada objeto creado desde una clase tiene sus propios valores pero comparte los mismos métodos.

- Definir una clase:

javascript

 Copiar código


```
class Persona {
  constructor(nombre, edad) {
    this.nombre = nombre;
    this.edad = edad;
  }

  saludar() {
    console.log(`Hola, me llamo ${this.nombre}`);
  }
}

const persona1 = new Persona('Andrea', 25);
persona1.saludar(); // "Hola, me llamo Andrea"
```

- Herencia:** La herencia en JavaScript permite que una clase hija herede propiedades y métodos de una clase padre. Usamos `extends` para crear una subclase.
- Ejemplo de herencia:

javascript

 Copiar código

```
class Estudiante extends Persona {
  constructor(nombre, edad, carrera) {
    super(nombre, edad); // Llama al constructor de la clase padre
    this.carrera = carrera;
  }

  estudiar() {
    console.log(`${this.nombre} está estudiando ${this.carrera}`);
  }
}

const estudiante1 = new Estudiante('Carlos', 21, 'Ingeniería');
estudiante1.saludar(); // "Hola, me llamo Carlos"
estudiante1.estudiar(); // "Carlos está estudiando Ingeniería"
```

## Encapsulamiento en JavaScript

El **encapsulamiento** es uno de los pilares de la **Programación Orientada a Objetos (POO)**, y su objetivo es restringir el acceso directo a algunos de los componentes de un objeto. La idea es proteger los **datos internos** y exponer solo lo que es necesario, para evitar que los datos sean modificados de manera inesperada o incorrecta.

### Encapsulamiento en ES6 con `#` (Propiedades Privadas)

JavaScript introdujo las **propiedades privadas** a través del símbolo `#`. Antes de esta mejora, el encapsulamiento no existía de forma nativa en JavaScript; los datos internos de una clase eran accesibles desde fuera. Con esta nueva sintaxis, ahora es posible definir atributos (propiedades) que solo pueden ser accedidos y modificados **dentro de la clase**.

- ¿Cómo funciona el `#`?

Al anteponer el símbolo `#` a una propiedad dentro de una clase, la conviertes en **privada**, lo que significa que no se puede acceder a ella desde fuera de la clase directamente.

```
class CuentaBancaria {  
    #saldo; // Propiedad privada  
  
    constructor() {  
        this.#saldo = 0; // Inicialización dentro de la clase  
    }  
  
    // Método para depositar dinero en la cuenta  
    depositar(cantidad) {  
        if (cantidad > 0) {  
            this.#saldo += cantidad;  
            console.log(`Depósito realizado. Saldo actual: ${this.#saldo}`);  
        } else {  
            console.log('Cantidad inválida');  
        }  
    }  
  
    // Método para obtener el saldo actual (getter)  
    obtenerSaldo() {  
        return this.#saldo; // Sólo se puede acceder desde dentro de la clase  
    }  
}  
  
const miCuenta = new CuentaBancaria();  
miCuenta.depositar(100); // "Depósito realizado. Saldo actual: $100"  
console.log(miCuenta.obtenerSaldo()); // 100  
console.log(miCuenta.#saldo); // Error: Propiedad privada, no accesible desde fuera
```

### Beneficios del Encapsulamiento:

1. **Protección de los Datos:** Los atributos privados no pueden ser modificados directamente desde fuera de la clase, lo que evita alteraciones accidentales o malintencionadas.
2. **Control sobre el Acceso:** Los datos se pueden modificar solo a través de **métodos públicos** que controlan cómo y cuándo se pueden cambiar los valores internos.
3. **Ocultar la Complejidad:** Al mantener detalles internos ocultos, puedes exponer solo lo esencial a otros programadores que usen tu código, mejorando la simplicidad y la comprensión del sistema.

### Métodos Getters y Setters:

Aunque las propiedades privadas protegen los datos, a menudo es necesario acceder a ellos de manera controlada. Aquí es donde los **getters** y **setters** entran en juego.

- **Getter:** Método que **devuelve** el valor de una propiedad privada.
- **Setter:** Método que **modifica** el valor de una propiedad privada.

```
class Persona {
  #nombre;
  #edad;

  constructor(nombre, edad) {
    this.#nombre = nombre;
    this.#edad = edad;
  }

  // Getter para obtener el nombre
  get nombre() {
    return this.#nombre;
  }

  // Setter para cambiar el nombre
  set nombre(nuevoNombre) {
    if (nuevoNombre) {
      this.#nombre = nuevoNombre;
    } else {
      console.log('Nombre inválido');
    }
  }
}
```



```
// Setter para cambiar la edad
set edad(nuevaEdad) {
  if (nuevaEdad > 0) {
    this.#edad = nuevaEdad;
  } else {
    console.log('Edad inválida');
  }
}

const persona1 = new Persona('Andrea', 25);
console.log(persona1.nombre); // Andrea


persona1.nombre = 'Code'; // Cambiamos el nombre usando el setter
console.log(persona1.nombre); // Code
```

## 5. Funciones Flecha

Las **funciones flecha** (`arrow functions`) son una manera más compacta de escribir funciones. Tienen la ventaja de no cambiar el valor de `this` dentro de su contexto.

- Ejemplo de función flecha:

javascript

 Copiar código

```
const sumar = (a, b) => a + b;
console.log(sumar(2, 3)); // 5
```

Las funciones flecha no tienen su propio `this`. Esto puede ser útil cuando trabajas con objetos o dentro de funciones callback.

## 6. Promesas y Async/Await

JavaScript es un lenguaje asíncrono. Esto significa que puede ejecutar tareas de manera no secuencial, como las solicitudes de red o la lectura de archivos. Para gestionar tareas asíncronas, se utilizan **Promesas** o la sintaxis moderna **async/await**.

- **Promesas:** Una promesa representa una operación que **aún no se ha completado**, pero que se completará en el futuro.

```
const promesa = new Promise((resolve, reject) => {
  let exito = true;
  if (exito) {
    resolve('Éxito');
  } else {
    reject('Fracaso');
  }
});

promesa.then(resultado => console.log(resultado)) // Éxito
  .catch(error => console.error(error));
```

Async/Await: `async` y `await` permiten trabajar con promesas de una manera más clara y secuencial.

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

## 7. Desestructuración de Objetos y Arrays

La desestructuración es una forma rápida de extraer valores de objetos y arrays en variables separadas.

- Objetos:

```
const persona = { nombre: 'Andrea', edad: 25 };
const { nombre, edad } = persona;
console.log(nombre); // Andrea
```

- Arrays:


```
const numeros = [1, 2, 3];
const [primero, segundo] = numeros;
console.log(primero); // 1
```

## 8. Modularización: import y export

La modularización permite dividir el código en archivos separados, facilitando su mantenimiento y reutilización.

- Exportar:


javascript

 Copiar código

```
export const nombre = 'Andrea';
export function saludar() {
  console.log('Hola');
}
```

- Importar:

javascript

 Copiar código


```
import { nombre, saludar } from './modulo.js';
saludar(); // Hola
```

## 9. JSON (JavaScript Object Notation)

JSON es un formato ligero y estándar para intercambiar datos entre el cliente (navegador) y el servidor.

- Convertir un objeto JavaScript a JSON:


javascript

 Copiar código

```
const persona = { nombre: 'Andrea', edad: 25 };
const json = JSON.stringify(persona);
console.log(json); // {"nombre":"Andrea","edad":25}
```

- Convertir JSON a un objeto JavaScript:

javascript

 Copiar código

```
const json = '{"nombre":"Andrea","edad":25}';
const persona = JSON.parse(json);
console.log(persona.nombre); // Andrea
```

## 10. Fetch API

La **Fetch API** te permite hacer solicitudes HTTP para obtener recursos, como datos de una API. Es mucho más moderna y flexible que `XMLHttpRequest`.

- Hacer una solicitud GET con Fetch:

```
javascript Copiar código  
  
fetch('https://api.example.com/data')  
  .then(response => response.json())  
  .then(data => console.log(data))  
  .catch(error => console.error('Error:', error));
```

- Usando async/await con Fetch:

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

### Métodos Getters y Setters:

Aunque las propiedades privadas protegen los datos, a menudo es necesario acceder a ellos de manera controlada. Aquí es donde los **getters** y **setters** entran en juego.

- **Getter**: Método que **devuelve** el valor de una propiedad privada.
- **Setter**: Método que **modifica** el valor de una propiedad privada.

Ejemplo con Getters y Setters:

```
class Persona {  
    #nombre;  
    #edad;  
  
    constructor(nombre, edad) {  
        this.#nombre = nombre;  
        this.#edad = edad;  
    }  
  
    // Getter para obtener el nombre  
    get nombre() {  
        return this.#nombre;  
    }  
}
```

```
    // Setter para cambiar el nombre  
    set nombre(nuevoNombre) {  
        if (nuevoNombre) {  
            this.#nombre = nuevoNombre;  
        } else {  
            console.log('Nombre inválido');  
        }  
    }  
  
    // Getter para obtener la edad  
    get edad() {  
        return this.#edad;  
    }  
  
    // Setter para cambiar la edad  
    set edad(nuevaEdad) {  
        if (nuevaEdad > 0) {  
            this.#edad = nuevaEdad;  
        } else {  
            console.log('Edad inválida');  
        }  
    }  
}
```

```
const persona1 = new Persona('Andrea', 25);  
console.log(persona1.nombre); // Andrea  
  
persona1.nombre = 'Code'; // Cambiamos el nombre usando el setter  
console.log(persona1.nombre); // Code
```

## 1. Closures (Clausuras)

Un **closure** es una función que "recuerda" el contexto donde fue creada, incluso después de que ese contexto haya terminado. Esto permite que las funciones internas accedan a variables definidas en su ámbito externo, incluso después de que la función externa haya terminado de ejecutarse.

Ejemplo de Closure:

```
javascript Copiar código

function crearContador() {
  let contador = 0;

  return function() { // Esta función interna "recuerda" el valor de 'contador'
    contador++;
    console.log(contador);
  };
}

const miContador = crearContador();
miContador(); // 1
miContador(); // 2
miContador(); // 3
```

El contador sigue incrementando porque la función interna tiene acceso al valor de `contador` gracias a la clausura, a pesar de que la función `crearContador` haya terminado.

## 2. Callbacks

Un **callback** es una función que se pasa como argumento a otra función y se ejecuta después de que ocurra un evento o una tarea específica.

Los callbacks son esenciales en JavaScript debido a su naturaleza asíncrona. Son utilizados para tareas como manejar eventos o hacer solicitudes a servidores.

Ejemplo de Callback:

```
function saludar(nombre, callback) {
  console.log(`Hola, ${nombre}`);
  callback();
}

function despedirse() {
  console.log('Adiós');
}

saludar('Andrea', despedirse); // Primero saluda, luego se despide
```

En este ejemplo, `despedirse` es la función callback que se pasa como argumento a `saludar` y se ejecuta después.

### 3. Promesas y Async/Await (Más a fondo)

#### Promesas:

Las **promesas** son fundamentales para manejar operaciones asíncronas. Representan un valor que puede estar disponible ahora, en el futuro, o nunca.

- Estados de una Promesa:
  1. Pendiente (Pending): La operación aún no ha terminado.
  2. Resuelta (Fulfilled): La operación ha sido completada con éxito.
  3. Rechazada (Rejected): La operación falló.

Ejemplo avanzado:

```
const miPromesa = new Promise((resolve, reject) => {
  const exito = true;

  setTimeout(() => {
    if (exito) {
      resolve('Operación exitosa');
    } else {
      reject('Operación fallida');
    }
  }, 2000); // Simulamos una tarea que tarda 2 segundos
});

miPromesa
  .then(resultado => console.log(resultado)) // "Operación exitosa"
  .catch(error => console.error(error))
  .finally(() => console.log('Promesa terminada')); // Se ejecuta al final, haya éxito o no
```

### Async/Await:

La sintaxis **async/await** permite escribir código asíncrono de manera más clara y secuencial, como si fuera código sincrónico.

- Ejemplo avanzado con async/await:

```
javascript Copiar código

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

obtenerDatos(); // Llama a la función asíncrona
```

## 4. Prototype y Herencia Prototípica

En JavaScript, cada objeto tiene una propiedad interna llamada `prototype`, que es una referencia a otro objeto. Esto es la base de la herencia prototípica.

### Ejemplo básico:

```
javascript Copiar código

function Persona(nombre, edad) {
  this.nombre = nombre;
  this.edad = edad;
}

Persona.prototype.saludar = function() {
  console.log(`Hola, me llamo ${this.nombre}`);
};

const persona1 = new Persona('Andrea', 25);
persona1.saludar(); // "Hola, me llamo Andrea"
```

Aquí, el método `saludar` se define en el `prototype` de `Persona`, lo que permite que todas las instancias de `Persona` lo compartan sin duplicar el método en cada objeto.



## 5. Event Loop y Concurrency

El **event loop** es el mecanismo que permite a JavaScript realizar tareas asíncronas (como temporizadores o peticiones a servidores) en un solo hilo, sin bloquear la ejecución del código.

- JavaScript ejecuta el código de manera sincrónica en el **Call Stack**.
- Las tareas asíncronas (promesas, callbacks, eventos) se manejan en la **cola de tareas** y son ejecutadas cuando el Call Stack está vacío.

### 1. Tipos de Datos en JavaScript:

- Primitivos: `number`, `string`, `boolean`, `undefined`, `null`, `symbol`, `bigint`.
- Referencia: Objetos, arrays, funciones.

### 2. Error Handling (Manejo de Errores):

- Usa `try`, `catch` y `finally` para manejar errores y evitar que el código se detenga inesperadamente.

Ejemplo:

```
javascript Copiar código

try {
  // Código que podría fallar
  let resultado = 10 / 0;
  console.log(resultado);
} catch (error) {
  console.error('Ha ocurrido un error:', error);
} finally {
  console.log('Este bloque siempre se ejecuta.');
```

### 3. AJAX y XHR:

- AJAX (Asynchronous JavaScript and XML) te permite actualizar una parte de la página sin recargarla completamente.
- Aunque `fetch` es la nueva API, **XMLHttpRequest** (XHR) es el método más antiguo y todavía usado en algunas bibliotecas antiguas.

## Índice de Temas para Profundizar:

1. Hoisting (Elevación)
2. Scope (Alcance)
3. Strict Mode
4. This en JavaScript
5. Funciones Constructoras
6. Destructuring (Desestructuración) Avanzada
7. Operadores Importantes en JavaScript
8. Map, Filter y Reduce
9. Set y Map (Colecciones en JavaScript)
10. Clonación de Objetos y Arrays
11. Expresiones Regulares (Regex)
12. Módulos de ES6 (Import y Export avanzados)
13. Web Storage (localStorage y sessionStorage)

### 1. Hoisting (Elevación)


El **hoisting** es el comportamiento por el cual las **declaraciones de variables y funciones** se "elevan" al principio de su contexto de ejecución, **antes** de que el código se ejecute.

#### Variables:

Con `var`, JavaScript eleva la declaración de la variable, pero no su inicialización.

Ejemplo:


javascript

 Copiar código

```
console.log(nombre); // undefined  
var nombre = "Andrea";
```

Aquí, JavaScript trata esto como si fuera:

javascript

 Copiar código

```
var nombre; // Hoisting de la declaración  
console.log(nombre); // undefined  
nombre = "Andrea"; // Inicialización después
```

Con `let` y `const`, el hoisting ocurre pero con una diferencia: estas variables no se pueden utilizar **antes** de su declaración (esto se llama el "Temporal Dead Zone").

Ejemplo con `let`:

```
javascript Copiar código  
  
console.log(nombre); // ReferenceError: Cannot access 'nombre' before initialization  
let nombre = "Andrea";
```

### Funciones:

Las funciones **declarativas** se elevan completamente, tanto la declaración como su contenido.

Ejemplo:

```
javascript Copiar código  
  
saludar(); // "Hola"  
function saludar() {  
    console.log("Hola");  
}
```

Sin embargo, las **funciones expresadas** no son elevadas de la misma manera. Solo se eleva la declaración de la variable, no la asignación.

Ejemplo con función expresada:

```
javascript Copiar código  
  
saludar(); // TypeError: saludar is not a function  
var saludar = function() {  
    console.log("Hola");  
};
```

## 2. Scope (Alcance)

El **scope** se refiere al contexto en el que las variables y funciones son accesibles. Hay dos tipos principales de scope en JavaScript:

- **Scope Global:** Las variables declaradas fuera de cualquier función están en el ámbito global y son accesibles desde cualquier parte del código.
- **Scope Local:** Las variables declaradas dentro de una función solo son accesibles dentro de esa función.

### Block Scope (Alcance de Bloque):

Las variables declaradas con `let` y `const` tienen **alcance de bloque**, lo que significa que solo existen dentro del bloque `{ }` en el que están.

Ejemplo:

```
if (true) {  
  let nombre = "Andrea"; // Solo accesible dentro del bloque  
  console.log(nombre); // Andrea  
}  
console.log(nombre); // ReferenceError: nombre is not defined
```

---

## 3. Strict Mode

El "Strict Mode" en JavaScript es un modo que fuerza una ejecución más estricta de las reglas del lenguaje, ayudando a evitar errores comunes y peligrosos.

Para habilitarlo, se coloca `"use strict";` al principio de un archivo o una función.

### Beneficios del Strict Mode:

- Previene la creación de variables globales accidentales.
- Lanza errores cuando se intenta modificar propiedades de solo lectura.
- Hace que `this` sea `undefined` en funciones no vinculadas, en lugar de referirse al objeto global.

### Ejemplo:

```
javascript Copiar código  
  
"use strict";  
nombre = "Andrea"; // Error: nombre no está definido
```

Sin `"use strict";`, JavaScript hubiera permitido la creación de `nombre` como una variable global de manera implícita.

## 4. This en JavaScript

`this` en JavaScript hace referencia al **contexto de ejecución** actual. Su valor cambia dependiendo de cómo y dónde se use la función.

### Casos Comunes de `this`:

1. En el ámbito global o dentro de una función regular:

```
javascript Copiar código  
  
console.log(this); // En el navegador, `this` es el objeto `window`
```

2. Dentro de un método de un objeto:

```
javascript Copiar código  
  
const persona = {  
  nombre: "Andrea",  
  saludar() {  
    console.log(this.nombre); // `this` se refiere al objeto `persona`  
  }  
};  
persona.saludar(); // Andrea
```

3. Dentro de una función en modo estricto: En Strict Mode, `this` dentro de una función regular será `undefined`.

```
javascript Copiar código  
  
"use strict";  
function saludar() {  
    console.log(this); // undefined  
}  
saludar();
```

4. En una función flecha (`=>`): Las funciones flecha no tienen su propio `this`, sino que lo heredan del contexto en el que fueron definidas.

```
javascript Copiar código  
  
const persona = {  
    nombre: "Andrea",  
    saludar: () => {  
        console.log(this.nombre); // `this` hace referencia al contexto global, no a  
    }  
};  
persona.saludar(); // undefined
```

## 5. Funciones Constructoras

Antes de que las clases fueran introducidas en ES6, JavaScript usaba funciones constructoras para crear objetos y simular la orientación a objetos.

### Ejemplo:

```
javascript Copiar código  
  
function Persona(nombre, edad) {  
    this.nombre = nombre;  
    this.edad = edad;  
}  
  
const persona1 = new Persona('Andrea', 25);  
console.log(persona1.nombre); // Andrea
```


Estas funciones usan el operador `new` para crear una nueva instancia del objeto, asignando `this` al nuevo objeto y devolviendo automáticamente el objeto.

## 6. Destructuring (Desestructuración) Avanzada

Ya hablamos de la desestructuración básica. Vamos a profundizar en más usos avanzados:

- **Desestructuración de parámetros:** Es útil cuando una función recibe un objeto y deseas extraer propiedades específicas.

javascript

 Copiar código


```
const persona = { nombre: "Andrea", edad: 25 };

function saludar({ nombre, edad }) {
  console.log(`Hola, me llamo ${nombre} y tengo ${edad} años.`);
}

saludar(persona); // Hola, me llamo Andrea y tengo 25 años.
```

- **Valores por defecto en la desestructuración:** Si una propiedad no está presente, se puede asignar un valor por defecto.

javascript

 Copiar código


```
const persona = { nombre: "Andrea" };
const { nombre, edad = 30 } = persona;

console.log(edad); // 30 (porque no había edad en el objeto)
```

## 7. Operadores Importantes en JavaScript

1. Operador Ternario ( `? :` ): Es un atajo para una declaración `if` sencilla.


javascript

 Copiar código

```
const edad = 20;  
const esMayor = edad >= 18 ? "Mayor de edad" : "Menor de edad";
```

2. Operador Nullish Coalescing ( `??` ): Devuelve el lado derecho si el lado izquierdo es `null` o `undefined`.


javascript

 Copiar código

```
const nombre = null;  
const saludo = nombre ?? "Desconocido";  
console.log(saludo); // Desconocido
```

3. Operador Lógico OR ( `||` ): Devuelve el primer valor verdadero.

javascript

 Copiar código

```
const nombre = "" || "Anonimo";  
console.log(nombre); // Anonimo
```




## 8. Map, Filter y Reduce

Estas son tres funciones esenciales para trabajar con arrays de manera funcional:

1. `map()` : Aplica una función a cada elemento de un array y devuelve un nuevo array.


javascript

 Copiar código

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

2. `filter()` : Filtra los elementos de un array que cumplen con una condición.


javascript

 Copiar código

```
const numeros = [1, 2, 3, 4];  
const pares = numeros.filter(n => n % 2 === 0);  
console.log(pares); // [2, 4]
```

3. `reduce()` : Acumula los valores de un array en un solo valor.

javascript

 Copiar código


```
const numeros = [1, 2, 3, 4];  
const suma = numeros.reduce((acumulador, actual) => acumulador + actual, 0);  
console.log(suma); // 10
```

## 9. Set y Map (Colecciones en JavaScript)

Los **Set** y **Map** son colecciones que se introdujeron en ES6 y ofrecen funcionalidades avanzadas para manejar datos.

1. **Set**: Almacena valores únicos. No permite duplicados.


javascript

 Copiar código

```
const miSet = new Set([1, 2, 3, 3, 4]);  
console.log(miSet); // Set { 1, 2, 3, 4 }
```

2. **Map**: Almacena pares clave-valor, pero a diferencia de los objetos, las claves pueden ser cualquier tipo de dato.

javascript

 Copiar código


```
const miMapa = new Map();  
miMapa.set('nombre', 'Andrea');  
console.log(miMapa.get('nombre')); // Andrea
```

## 11. Expresiones Regulares (Regex)

Las expresiones regulares son una herramienta poderosa para trabajar con cadenas, permitiendo buscar y manipular patrones.

Ejemplo básico:

javascript

 Copiar código


```
const regex = /\d+/; // Coincide con cualquier secuencia de dígitos  
const resultado = "La edad es 25".match(regex);  
console.log(resultado); // ["25"]
```

## 10. Clonación de Objetos y Arrays

Clonar objetos o arrays es importante cuando quieres copiar datos sin modificar el original.

- Clonación superficial con `Object.assign`:


javascript

 Copiar código

```
const original = { nombre: 'Andrea', edad: 25 };  
const copia = Object.assign({}, original);
```

- Clonación profunda con `JSON.parse` y `JSON.stringify`: Para estructuras más complejas (con arrays u objetos anidados), necesitas una clonación profunda:

javascript

 Copiar código


```
const original = { nombre: 'Andrea', datos: { edad: 25 } };  
const clon = JSON.parse(JSON.stringify(original));
```

## 12. Módulos de ES6 (Import y Export avanzados)

Con ES6, los módulos permiten organizar y dividir mejor el código.

- Importar todo un módulo:


javascript

 Copiar código

```
import * as utils from './utilidades.js';  
utils.miFuncion();
```

- Exportar por defecto:

javascript

 Copiar código


```
export default function saludar() {  
  console.log("Hola");  
}
```

### 13. Web Storage (localStorage y sessionStorage)

El **Web Storage** permite almacenar datos clave-valor en el navegador, lo que es útil para persistir información entre visitas.

1. **localStorage**: Los datos persisten incluso después de cerrar el navegador.


javascript

 Copiar código

```
localStorage.setItem('nombre', 'Andrea');  
console.log(localStorage.getItem('nombre')); // Andrea
```

2. **sessionStorage**: Los datos solo persisten durante la sesión actual.

javascript

 Copiar código

```
sessionStorage.setItem('edad', '25');  
console.log(sessionStorage.getItem('edad')); // 25
```

## 11. Expresiones Regulares (Regex)

Las expresiones regulares (abreviadas como **Regex** o **RegExp**) son una herramienta extremadamente poderosa para trabajar con cadenas de texto, ya que permiten buscar, validar, o manipular patrones en textos de manera eficiente. Se usan para realizar búsquedas y coincidencias complejas que de otra manera serían difíciles de implementar con simples métodos de cadena.

### Sintaxis Básica de Regex

Una expresión regular está compuesta por **patrones de caracteres** especiales. Aquí tienes algunos de los más importantes:

- **.** (Punto): Representa **cualquier carácter** excepto un salto de línea.
- **\d**: Coincide con cualquier **dígito** (0-9).
- **\w**: Coincide con cualquier carácter **alfanumérico** (letras, números y **\_**).
- **\s**: Coincide con cualquier carácter de **espacio en blanco** (espacio, tabulación, salto de línea).
- **\*** (Asterisco): Coincide con **cero o más** repeticiones del carácter anterior.
- **+** (Más): Coincide con **uno o más** repeticiones del carácter anterior.
- **?** (Interrogación): Coincide con **cero o una** aparición del carácter anterior.
- **[]**: Define un **conjunto** de caracteres que coinciden. Ej: **[abc]** coincide con "a", "b" o "c".
- **^** (Caret): Coincide con el **inicio de una línea**.
- **\$** (Dólar): Coincide con el **final de una línea**.
- **{}** (Llaves): Especifica la **cantidad exacta** de repeticiones. Ej: **{3}** para tres repeticiones exactas.

## Ejemplos Prácticos:

1. Coincidencia de números:

```
javascript Copiar código  
  
const regex = /\d+/; // Coincide con uno o más dígitos  
const texto = "La edad es 25";  
const resultado = texto.match(regex);  
console.log(resultado); // ["25"]
```

2. Validar un formato de email:

```
javascript Copiar código  
  
const regexEmail = /^[^\w-\.\.]+\@([\w-]+\w-)+[\w-]{2,4}$/;  
const email = "test@example.com";  
const esValido = regexEmail.test(email); // `test` devuelve true o false  
console.log(esValido); // true
```

3. Buscar palabras que empiezan con "J":

```
javascript Copiar código  
  
const regex = /\bJ\w+/g; // Coincide con palabras que empiezan con "J"  
const texto = "JavaScript es un lenguaje de programación.";   
const resultado = texto.match(regex);  
console.log(resultado); // ["JavaScript"]
```

4. Reemplazar números en una cadena:

```
javascript Copiar código  
  
const texto = "La casa cuesta 1000 dólares.";   
const nuevoTexto = texto.replace(/\d+/g, 'mil');  
console.log(nuevoTexto); // "La casa cuesta mil dólares."
```

### Flags en Expresiones Regulares:

Las expresiones regulares tienen varias "banderas" que modifican su comportamiento:

- **g**: Búsqueda **global**, encuentra todas las coincidencias.
- **i**: Búsqueda **insensible a mayúsculas y minúsculas**.
- **m**: Búsqueda en modo **multilínea**, donde `^` y `$` coinciden con el inicio y fin de cada línea.

Ejemplo con flags:

```
javascript Copiar código  
  
const regex = /hola/gim;  
const texto = "Hola\nhola";  
const coincidencias = texto.match(regex);  
console.log(coincidencias); // ["Hola", "hola"]
```

## 12. Módulos de ES6 (Import y Export avanzados)

Los **módulos de ES6** permiten organizar y estructurar el código en diferentes archivos. Esto es esencial en proyectos grandes para mantener el código **limpio, legible y reutilizable**.

### Conceptos Básicos de Módulos:

1. **Exportar (Export)**: Permite que **funciones, variables, objetos o clases** sean accesibles desde otros archivos.
2. **Importar (Import)**: Se usa para **traer elementos** de otro archivo y utilizarlos.

### Exportar y Importar en ES6

Existen dos tipos de exportación en JavaScript:

#### 1. Exportación Nombrada (Named Export):


Permite exportar varias variables o funciones con nombre, y cuando las importas debes referirte a ellas por el mismo nombre.



- Ejemplo de Exportación Nombrada:

archivo.js:

javascript


 Copiar código

```
export const nombre = 'Andrea';  
export function saludar() {  
  console.log("Hola");  
}  
export const edad = 25;
```

- Ejemplo de Importación Nombrada:

main.js:


javascript

 Copiar código

```
import { nombre, saludar, edad } from './archivo.js';  
console.log(nombre); // Andrea  
saludar(); // "Hola"  
console.log(edad); // 25
```

También puedes **renombrar** los elementos durante la importación:

javascript

 Copiar código

```
import { nombre as alias, saludar } from './archivo.js';  
console.log(alias); // Andrea
```



## 2. Exportación por Defecto (Default Export):

Cada archivo puede tener una única exportación por defecto, que se puede importar sin utilizar llaves `{}`.

- Ejemplo de Exportación por Defecto:

archivo.js:

```
javascript Copiar código  
  
export default function() {  
  console.log("Soy la exportación por defecto");  
}
```

- Ejemplo de Importación por Defecto:

main.js:

```
javascript Copiar código  
  
import saludarDefecto from './archivo.js';  
saludarDefecto(); // "Soy la exportación por defecto"
```

### Combinación de Exportaciones:

Puedes tener tanto una exportación por defecto como exportaciones nombradas en un mismo archivo.

archivo.js:

```
javascript Copiar código  
  
export const edad = 25;  
export default function saludar() {  
  console.log("Hola desde la exportación por defecto");  
}
```

main.js:

```
javascript Copiar código  
  
import saludar, { edad } from './archivo.js';  
console.log(edad); // 25  
saludar(); // "Hola desde la exportación por defecto"
```


### Métodos Comunes:

- `setItem(key, value)` : Guarda un valor bajo una clave.
- `getItem(key)` : Obtiene el valor almacenado bajo una clave.
- `removeItem(key)` : Elimina el valor asociado a la clave.
- `clear()` : Borra todos los datos almacenados.

### Ejemplos con localStorage:

1. Guardar datos en `localStorage` :

javascript

 Copiar código

```
localStorage.setItem('nombre', 'Andrea');  
localStorage.setItem('edad', '25');
```

## 13. Web Storage (localStorage y sessionStorage)

El **Web Storage** es una API de JavaScript que permite almacenar **datos clave-valor** directamente en el navegador. Esto es muy útil cuando quieres **guardar datos del usuario** para que persistan entre sesiones o durante la navegación, sin necesidad de usar cookies.

### Tipos de Web Storage:

1. `localStorage`:

- Los datos almacenados en `localStorage` **persisten incluso después de cerrar el navegador**.
- El almacenamiento es específico por **origen** (dominio, protocolo, puerto).
- No tiene fecha de expiración.

2. `sessionStorage`:

- Los datos en `sessionStorage` **persisten solo durante la sesión actual** del navegador (es decir, se borran cuando se cierra la pestaña o el navegador).
- Los datos son específicos por pestaña o ventana.


### Métodos Comunes:

- `setItem(key, value)` : Guarda un valor bajo una clave.
- `getItem(key)` : Obtiene el valor almacenado bajo una clave.
- `removeItem(key)` : Elimina el valor asociado a la clave.
- `clear()` : Borra todos los datos almacenados.

### Ejemplos con localStorage:

1. Guardar datos en `localStorage` :

javascript

 Copiar código


```
localStorage.setItem('nombre', 'Andrea');  
localStorage.setItem('edad', '25');
```

2. Recuperar datos:

```
const nombre = localStorage.getItem('nombre');  
const edad = localStorage.getItem('edad');  
console.log(nombre); // "Andrea"  
console.log(edad); // "25"
```

3. Eliminar un dato específico:


javascript

 Copiar código

```
localStorage.removeItem('edad');
```

4. Borrar todo el almacenamiento:

javascript


 Copiar código

```
localStorage.clear();
```

Ejemplos con sessionStorage:

1. Guardar datos en sessionStorage:


javascript

 Copiar código

```
sessionStorage.setItem('usuario', 'Andrea');
```

2. Recuperar datos:


javascript

 Copiar código

```
const usuario = sessionStorage.getItem('usuario');  
console.log(usuario); // "Andrea"
```

3. Eliminar un dato específico:


javascript

 Copiar código

```
sessionStorage.removeItem('usuario');
```

4. Borrar todos los datos de la sesión:

javascript

 Copiar código

```
sessionStorage.clear();
```

¿Cuándo usar localStorage y sessionStorage?

- Usa localStorage cuando necesites **persistir datos** entre sesiones del navegador, por ejemplo, para almacenar preferencias del usuario.
- Usa sessionStorage cuando solo necesites almacenar datos **temporales** que se utilicen durante una sesión particular de navegación, como datos de un formulario que no deberían persistir.


Vilma Ponce-Desarrollo **FullStack Javascript**-Nodo Tecnológico Catamarca.

### Limitaciones del Web Storage:

- Ambos ( `localStorage` y `sessionStorage` ) están limitados a 5-10MB de almacenamiento, dependiendo del navegador.
- No puedes almacenar **objetos complejos** directamente. Debes convertirlos en cadenas usando `JSON.stringify` y `JSON.parse`.

### Ejemplo con Objetos en `localStorage` :

javascript

 Copiar código

```
const persona = { nombre: "Andrea", edad: 25 };

// Guardar objeto
localStorage.setItem('persona', JSON.stringify(persona));

// Recuperar objeto
const personaRecuperada = JSON.parse(localStorage.getItem('persona'));
console.log(personaRecuperada.nombre); // "Andrea"
```