CORFO
BecasCapital Humano

Adalid
aspasia
LA FORMACIÓN DE TU FUTURO

MÓDULO 3

PROGRAMACIÓN AVANZADA EN JAVASCRIPT

Manual del Módulo 3: Programación Avanzada en JavaScript

Introducción

Este módulo cubre las características avanzadas de JavaScript, proporcionando a los estudiantes los conocimientos necesarios para desarrollar aplicaciones más complejas. Se exploran conceptos fundamentales como asincronía, estructuras de datos, algoritmos, y técnicas de programación funcional y orientada a objetos. La meta es que los estudiantes puedan implementar software de mediana complejidad utilizando herramientas avanzadas de JavaScript.

1. Generalidades del Lenguaje JavaScript

1.1 Historia de JavaScript

JavaScript fue creado por Brendan Eich en 1995, en solo 10 días, como un lenguaje de scripting para Netscape Navigator. Aunque fue diseñado inicialmente para agregar interactividad básica a los sitios web, ha evolucionado en un lenguaje versátil usado tanto en el frontend como en el backend (gracias a Node.js).

1.2 Uso en Navegadores Web

JavaScript es el único lenguaje de programación que los navegadores ejecutan de forma nativa para manejar la interactividad y la manipulación del DOM (Document Object Model). Permite modificar el contenido de una página en tiempo real, responder a eventos del usuario, validar formularios, y más.

Ejemplo:

```
document.getElementById('btn').addEventListener('click', function() {
```

```
    alert('¡Botón clickeado!');
});
```

1.3 Entornos Virtuales de JavaScript

Además de ejecutarse en los navegadores, JavaScript puede ejecutarse en servidores utilizando **Node.js**. Esto permite usar JavaScript para desarrollar aplicaciones backend, servicios REST, o incluso sistemas IoT.

1.4 Diferencias entre JavaScript y Otros Lenguajes

- **Tipado dinámico**: A diferencia de lenguajes como **Java** o **C++** que son tipados estáticamente, JavaScript es dinámico y flexible.

- **Interpretado**: JavaScript se ejecuta línea por línea en lugar de ser compilado previamente.

- **Multiparadigma**: Soporta programación orientada a objetos, funcional e imperativa.

1.5 Paradigmas de Programación Soportados por JavaScript

JavaScript soporta los siguientes paradigmas:

- **Imperativo**: Realiza tareas paso a paso mediante instrucciones explícitas.
- **Funcional**: Usa funciones puras, inmutabilidad y evita el estado global.
- **Orientado a objetos**: Utiliza clases y objetos para modelar el comportamiento del sistema.

Ejemplo (Imperativo vs Funcional):

```
// Imperativo

let numeros = [1, 2, 3, 4];
let resultado = [];
```

```
for (let i = 0; i < numeros.length; i++) {
    resultado.push(numeros[i] * 2);
}

console.log(resultado);
```

```
// Funcional

let resultadoFuncional = numeros.map(num => num * 2);

console.log(resultadoFuncional);
```

1.6 Fortalezas del Lenguaje y Forma Correcta de Uso

- **Asincronía**: JavaScript usa un modelo basado en eventos, lo que permite realizar múltiples tareas sin bloquear la ejecución.

- **Extensibilidad**: Puede extenderse mediante bibliotecas y frameworks (React, Node.js, etc.).

- **Flexibilidad**: Gracias a su naturaleza dinámica, permite una gran flexibilidad a la hora de manipular datos y estructuras.

Buenas prácticas:

- Evitar el uso de `var`; preferir `let` y `const` para una mejor gestión del alcance.
- Utilizar `async/await` para manejar operaciones asíncronas.

1.7 JavaScript como Lenguaje Asíncrono

JavaScript utiliza un modelo asíncrono, gestionado a través del **event loop**, para manejar tareas como peticiones HTTP, sin bloquear el hilo principal.

Ejemplo:

```
setTimeout(() => {
    console.log('Esto se ejecuta después de 2 segundos');
}, 2000);

console.log('Este mensaje aparece primero');
```

1.8 Sintaxis e Indentación

La sintaxis de JavaScript es sencilla pero potente. Seguir una buena convención de indentación es clave para mantener el código limpio y legible.

Ejemplo:

```
function saludar(nombre) {
    if (nombre) {
        console.log(`Hola, ${nombre}`);
    } else {
        console.log('Hola, visitante');
    }
}
```

2. Evolución del Lenguaje

2.1 Lenguaje Interpretado vs. Compilado

JavaScript es un **lenguaje interpretado**, lo que significa que es ejecutado línea por línea por el motor de JavaScript del navegador o servidor (Node.js). Esto es diferente de lenguajes como C++ o Java, que son compilados a código máquina antes de ejecutarse.

2.2 El Estándar ECMAScript

JavaScript está basado en el estándar **ECMAScript**, gestionado por ECMA International. Cada versión de ECMAScript introduce nuevas características que mejoran el lenguaje.

2.3 JavaScript vs. ECMAScript

- **JavaScript**: Es la implementación del estándar ECMAScript que sigue los navegadores y entornos como Node.js.

- **ECMAScript**: Define el comportamiento y las funcionalidades del lenguaje.

2.4 Evolución de ECMAScript: desde ES3 a ES9

La evolución de ECMAScript ha sido crucial para la modernización de JavaScript. Las versiones más relevantes incluyen:

- **ES5 (2009)**: Introdujo JSON nativo, `strict mode`, y mejoras en la manipulación de objetos.

- **ES6 (2015)**: Introdujo `let`, `const`, clases, módulos, promesas, y funciones flecha.

- **ES9 (2018)**: Mejoras en la gestión asíncrona, `Promise.finally()`, y operadores spread/rest.

2.5 Acerca de TypeScript y sus Características

TypeScript es un superconjunto de JavaScript que añade **tipado estático** opcional, lo que facilita el desarrollo de grandes aplicaciones al detectar errores en tiempo de desarrollo.

Ejemplo TypeScript:

```
let edad: number = 25;
```

```
console.log(edad);
```

2.6 ¿Dónde y Cómo se aplica TypeScript?

TypeScript es ampliamente utilizado en proyectos a gran escala donde el tipado estático es crucial para mantener el código escalable y robusto. Es el lenguaje principal en frameworks como **Angular**.

3. El Stack JavaScript

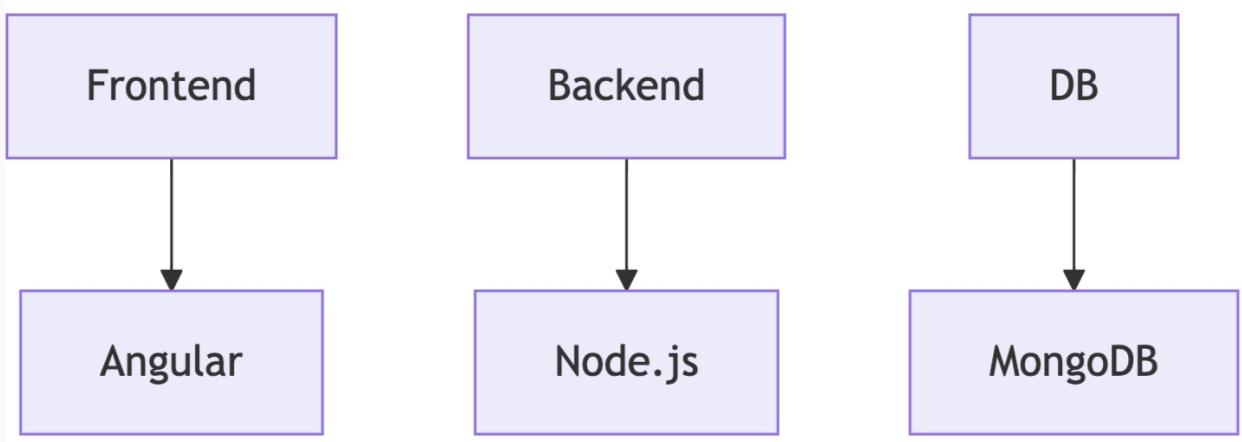
3.1 Qué es un Stack de Desarrollo

Un **stack de desarrollo** es un conjunto de tecnologías utilizadas para construir aplicaciones completas. En el caso de JavaScript, permite desarrollar tanto el frontend como el backend utilizando el mismo lenguaje.

3.2 Stacks Basados en JavaScript (MEAN, MERN)

- **MEAN**: MongoDB, Express.js, Angular, Node.js.
- **MERN**: MongoDB, Express.js, React, Node.js.

Ambos stacks permiten desarrollar aplicaciones completas con JavaScript en todas las capas del desarrollo.



3.3 Aplicaciones para Frontend

JavaScript es esencial en el desarrollo frontend para crear **Single Page Applications (SPA)**, utilizando frameworks como **React**, **Vue.js** o **Angular** para gestionar la interfaz de usuario y la lógica de interacción.

3.4 Aplicaciones para Backend

En el backend, **Node.js** permite a los desarrolladores utilizar JavaScript para construir servidores escalables, manejar bases de datos y procesar peticiones HTTP.

3.5 Frameworks Basados en JavaScript

JavaScript cuenta con una variedad de frameworks:

- **Frontend**: React, Angular, Vue.js.
- **Backend**: Express.js, Koa.js, Nest.js.

3.6 Acceso a Datos con JavaScript

JavaScript puede interactuar con bases de datos no relacionales como **MongoDB** usando bibliotecas como **Mongoose**, o bases de datos SQL mediante **Sequelize**.

3.7 Lenguajes Derivados de JavaScript

JavaScript ha inspirado lenguajes como **TypeScript** y **CoffeeScript**, que añaden características adicionales y una sintaxis más clara para proyectos más grandes.

3.8 Utilitarios

JavaScript tiene una gran cantidad de bibliotecas y utilitarios para mejorar la productividad:

- **Lodash**: Facilita el manejo de arrays y objetos.
- **Moment.js**: Facilita la manipulación de fechas.

3.9 El Garbage Collector en JavaScript

El **Garbage Collector** de JavaScript se encarga de liberar la memoria que ya no es utilizada por la aplicación, permitiendo una gestión eficiente de los recursos.

4. Entorno de Ejecución JavaScript

4.1 Entorno de Ejecución

El entorno de ejecución de JavaScript depende del **navegador** o de **Node.js**. En el navegador, se utiliza para manipular el DOM y responder a eventos de usuario, mientras que en Node.js se usa para manejar peticiones HTTP y gestionar el backend.

4.2 La Consola de Comandos

La consola es una herramienta poderosa para depurar y ejecutar comandos JavaScript directamente en el navegador o en Node.js.

4.3 Setting de Variables

JavaScript permite declarar variables utilizando `var`, `let` o `const`, cada uno con diferentes reglas de alcance y comportamiento.

4.4 console.log()

`console.log()` es la función más utilizada para imprimir información en la consola durante el desarrollo.

Ejemplo:

```
let nombre = 'Carlos';

console.log(nombre);
```

4.5 Estructura de un Programa JavaScript (import, require)

En **ES6**, JavaScript soporta la modularización nativa mediante `import` y `export`, facilitando la organización del código en múltiples archivos.

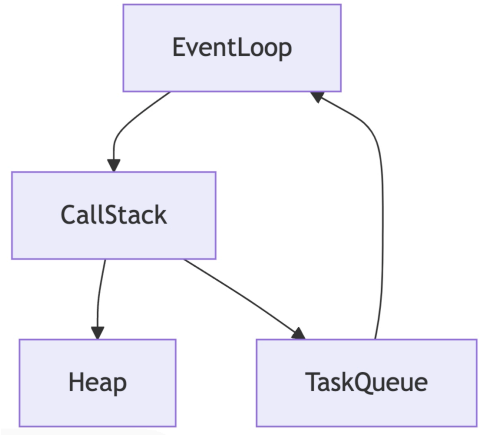
Ejemplo:

```
// modulo.js

export function saludo() {
    return "Hola";
}
```

```
// main.js

import { saludo } from './modulo.js';

console.log(saludo());
```

4.6 El Event Loop (stack, heap, queue)

El **event loop** es el corazón del modelo asincrónico de JavaScript. Es responsable de manejar la ejecución de código, la recolección de eventos y la delegación de tareas asincrónicas (como temporizadores y peticiones HTTP).



5. Desarrollo de Aplicaciones JavaScript

5.1 Variables

5.1.1 Scope de Variables

El **scope** (alcance) de una variable define desde dónde se puede acceder a ella en el código. En JavaScript, existen tres tipos de alcance:

- **Global**: Las variables declaradas fuera de cualquier función tienen un alcance global y pueden ser accedidas desde cualquier parte del código.
- **De función**: Las variables declaradas dentro de una función solo pueden ser accedidas desde esa función.

- **De bloque**: Introducido con `let` y `const` en ES6, estas variables solo son accesibles dentro del bloque `{}` en el que se declaran.

Ejemplo:

```
let x = 10; // Variable global

function ejemplo() {
    let y = 20; // Variable de función

    if (true) {
        let z = 30; // Variable de bloque
        console.log(z); // 30
    }

    // console.log(z); // Error: z no está definida
}

ejemplo();
```

5.1.2 Hoisting

El **hoisting** es un comportamiento en JavaScript donde las declaraciones de variables (usando `var`) y funciones se "elevan" al inicio de su contexto de ejecución. Sin embargo, las inicializaciones de las variables no son elevadas, lo que puede causar comportamiento inesperado.

Ejemplo:

```
console.log(nombre); // undefined debido al hoisting

var nombre = "Juan";
```

En este ejemplo, la declaración `var nombre` se eleva, pero su asignación no, lo que resulta en `undefined` en lugar de un error.

5.1.3 var vs. let vs. const

- **`var`**: Declara variables con scope de función. Propensa a problemas de hoisting y redeclaración accidental.
- **`let`**: Introducida en ES6, tiene alcance de bloque y no permite redeclaración en el mismo contexto.
- **`const`**: Similar a `let`, pero no permite reasignar valores después de la inicialización.

Ejemplo:

```
const PI = 3.1416;
let edad = 30;
edad = 31; // Válido

// PI = 3.14; // Error: no se puede reasignar una constante
```

5.2 Tipos de Datos Primitivos

Los **tipos de datos primitivos** en JavaScript son:

- **String**: Cadenas de texto.
- **Number**: Números, tanto enteros como decimales.
- **Boolean**: Valores verdaderos (`true`) o falsos (`false`).
- **Undefined**: El valor por defecto de variables no inicializadas.
- **Null**: Representa un valor intencionalmente vacío.
- **Symbol**: Introducido en ES6, es usado para crear identificadores únicos.

Ejemplo:

```
let nombre = "Carlos";

let edad = 25;

let esEstudiante = true;

let indefinido; // undefined por defecto
```

```
let vacio = null;
```

5.3 Operadores

JavaScript incluye diferentes tipos de operadores:

- **Operadores Aritméticos**: `+`, `-`, `*`, `/`, `%` (modulo).
- **Operadores de Comparación**: `==`, `===`, `!=`, `!==`.
- **Operadores Lógicos**: `&&` (AND), `||` (OR), `!` (NOT).
- **Operadores de Asignación**: `=`, `+=`, `-=`, `*=`, `/=`.

Ejemplo:

```
let a = 10, b = 20;

console.log(a + b); // 30

console.log(a == b); // false

console.log(a !== b); // true
```

5.4 Diferencia entre `null` y `undefined`

- **`null`**: Representa la ausencia intencional de un valor.
- **`undefined`**: Representa la ausencia de asignación de valor en una variable.

Ejemplo:

```
let noAsignada;
let vacia = null;

console.log(noAsignada); // undefined
```

```
console.log(vacia); // null
```

5.5 El Operador `typeof`

El operador `typeof` se utiliza para determinar el tipo de una variable o valor.

Ejemplo:

```
console.log(typeof 42); // "number"

console.log(typeof "Hola"); // "string"

console.log(typeof true); // "boolean"
```

5.6 El Comparador `==` vs. `===`

- `==`: Realiza una comparación débil, que convierte los tipos antes de compararlos.
- `===`: Realiza una comparación estricta, sin convertir los tipos.

Ejemplo:

```
console.log(2 == "2"); // true (conversión implícita)

console.log(2 === "2"); // false (comparación estricta)
```

5.7 El Tipado en JavaScript

JavaScript es un lenguaje de **tipado débil**, lo que significa que las variables pueden cambiar de tipo en tiempo de ejecución.

Ejemplo:

```
let variable = 42; // Tipo número

variable = "Texto"; // Cambia a tipo string
```

5.8 Instrucciones Condicionales

Las estructuras condicionales permiten ejecutar diferentes bloques de código basados en condiciones.

Ejemplo:

```
let edad = 20;

if (edad >= 18) {
    console.log("Eres mayor de edad");
} else {
    console.log("Eres menor de edad");
}
```

5.9 Ciclos e Iteraciones

Los ciclos permiten repetir un bloque de código hasta que se cumpla una condición. En JavaScript, existen ciclos como `for`, `while` y `do...while`.

5.9.1 Ciclos y Condiciones Anidados (`break`, `continue`)

- **`break`**: Termina el ciclo actual inmediatamente.
- **`continue`**: Salta a la siguiente iteración del ciclo.

Ejemplo:

```
for (let i = 0; i < 10; i++) {
```

```
    if (i === 5) {
        break; // Termina cuando i es 5
    }

    console.log(i); // Imprime del 0 al 4
}
```

5.10 `try/catch` y Manejo de Errores

El manejo de errores se realiza utilizando `try` y `catch`. Si ocurre un error en el bloque `try`, el flujo del programa pasa al bloque `catch`.

Ejemplo:

```
try {
    let resultado = noDefinida; // Esto provocará un error
} catch (error) {
    console.log("Se ha producido un error:", error.message);
}
```

5.11 Expresiones Regulares

Las **expresiones regulares** permiten buscar patrones dentro de cadenas de texto.

Ejemplo:

```
let texto = "Aprende JavaScript en 2023";
let patron = /\d+/; // Busca números en el texto

console.log(texto.match(patron)); // ["2023"]
```

5.12 Arreglos

Los **arreglos** en JavaScript son estructuras que permiten almacenar múltiples valores en una sola variable.

Ejemplo:

```
let numeros = [10, 20, 30];

console.log(numeros[0]); // 10
```

5.13 Debugger

El comando `debugger` detiene la ejecución de un programa, permitiendo inspeccionar el estado del programa en las herramientas de desarrollo del navegador.

Ejemplo:

```
let x = 10;

debugger; // Pausa la ejecución

x++;

console.log(x);
```

5.14 Manipulación de Archivos

JavaScript puede leer y escribir archivos en el servidor utilizando **Node.js**.

Ejemplo (Node.js):

```
const fs = require('fs');

fs.writeFile('archivo.txt', 'Contenido del archivo', (err) => {
```

```
    if (err) throw err;
    console.log('El archivo ha sido guardado');
});
```

6. Notación de Objetos JavaScript (JSON)

6.1 Objetos JSON

JSON (JavaScript Object Notation) es un formato de intercambio de datos ligero, fácil de leer y escribir para los humanos, y fácil de analizar y generar para las máquinas. Es ampliamente utilizado en aplicaciones web para intercambiar datos entre el cliente y el servidor.

6.1.1 Sintaxis y Notación

La sintaxis de JSON es similar a la de los objetos de JavaScript. Un objeto JSON está formado por pares clave-valor donde las claves son cadenas (strings) y los valores pueden ser strings, números, booleanos, objetos, arrays o `null`.

Ejemplo:

```
{
  "nombre": "Carlos",
  "edad": 25,
  "activo": true,
  "cursos": ["JavaScript", "Node.js"],
  "direccion": {
    "calle": "Av. Principal",
    "ciudad": "Santiago"
  }
}
```

6.1.2 Tipos de Datos en Objetos JSON

Los valores de los objetos JSON pueden ser de los siguientes tipos:

- **String**: "cadena de texto"
- **Number**: 10, 3.14
- **Boolean**: `true` o `false`
- **Array**: [1, 2, 3, "texto"]
- **Object**: {"clave": "valor"}
- **null**: Representa un valor nulo

6.1.3 Objetos Anidados

Los objetos JSON pueden contener otros objetos como valores. Esto es útil para representar relaciones jerárquicas o estructuras de datos complejas.

Ejemplo:

```
{
  "producto": "Computadora",
  "detalles": {
    "marca": "XYZ",
    "precio": 800
  }
}
```

6.1.4 Funciones Dentro de Objetos

JSON no permite funciones como valores, ya que está diseñado únicamente para el intercambio de datos. Sin embargo, en JavaScript los objetos sí pueden contener funciones.

Ejemplo de Objeto en JavaScript con Funciones:

```
let persona = {
    nombre: "Juan",
    edad: 30,
```

```
    saludar: function() {
        console.log(`Hola, soy ${this.nombre}`);
    }
};

persona.saludar(); // "Hola, soy Juan"
```

6.1.4 Asignación por Destructuración

JavaScript permite extraer propiedades de un objeto y asignarlas a variables mediante la destructuración.

Ejemplo:

```
let persona = { nombre: "Carlos", edad: 28 };
let { nombre, edad } = persona;

console.log(nombre); // "Carlos"
console.log(edad);   // 28
```

6.2 Operaciones con Objetos JSON

6.2.1Clonación

Para clonar un objeto JSON en JavaScript, puedes usar `JSON.parse()` y `JSON.stringify()` o el operador spread (`...`).

Ejemplo:

```
let objetoOriginal = { nombre: "Pedro", edad: 30 };
let objetoClonado = JSON.parse(JSON.stringify(objetoOriginal));

console.log(objetoClonado); // { nombre: "Pedro", edad: 30 }
```

6.2.2 Merge (Fusión de Objetos)

La fusión de objetos combina las propiedades de dos o más objetos en uno solo. Puedes usar `Object.assign()` o el operador spread.

Ejemplo:

```
let obj1 = { nombre: "Ana" };
let obj2 = { edad: 25 };

let fusionado = { ...obj1, ...obj2 };
console.log(fusionado); // { nombre: "Ana", edad: 25 }
```

6.2.3 Recorrido, `parse` y `stringify`

- **`JSON.parse()`**: Convierte una cadena JSON en un objeto JavaScript.
- **`JSON.stringify()`**: Convierte un objeto JavaScript en una cadena JSON.

Ejemplo:

```
let cadena = '{"nombre": "Laura", "edad": 22}';

let objeto = JSON.parse(cadena);

console.log(objeto.nombre); // Laura

let nuevoJSON = JSON.stringify(objeto);

console.log(nuevoJSON); // '{"nombre":"Laura","edad":22}'
```

6.3 JSON como Base para el Protocolo API REST

JSON es el formato más comúnmente utilizado para el intercambio de datos en las APIs REST. Cuando un cliente solicita datos a un servidor, el servidor suele responder con un objeto JSON que contiene la información solicitada.

Ejemplo de respuesta JSON en una API:

```
{
  "usuario": {
    "id": 123,
    "nombre": "María",
    "email": "maria@ejemplo.com"
  }
}
```

6.4 Otros Formatos de Intercambio de Información

6.4.1 El Lenguaje YAML

YAML (YAML Ain't Markup Language) es otro formato legible por humanos que se utiliza para la serialización de datos. A diferencia de JSON, es más flexible en su sintaxis y es usado en archivos de configuración.

Ejemplo de YAML:

```
usuario:

  nombre: "Pedro"
  edad: 25
  activo: true
```

6.4.2 Comparación con XML

XML (Extensible Markup Language) es otro formato de intercambio de datos, pero su sintaxis es más compleja y verbosa que JSON. Aunque es más flexible en su estructura, JSON es más ligero y fácil de usar para la mayoría de las aplicaciones web.

Ejemplo de XML:

```
<usuario>

  <nombre>Pedro</nombre>
  <edad>25</edad>
  <activo>true</activo>

</usuario>
```

7. Estructuras de Datos en JavaScript

Las estructuras de datos permiten organizar y manipular información de manera eficiente. JavaScript ofrece varias estructuras de datos incorporadas como arreglos y objetos, y también permite implementar estructuras más complejas.

7.1 Qué es una Estructura de Datos

Una **estructura de datos** es una manera organizada de almacenar, gestionar y acceder a datos. Las estructuras de datos eficientes son cruciales para el desarrollo de algoritmos y aplicaciones.

7.2 Arreglos de Datos

Los **arreglos** son la estructura de datos más básica en JavaScript. Son colecciones de elementos indexados, donde cada elemento tiene una posición (índice).

7.2.1 Tipos de Arreglos (Vectores, Matrices)

- **Vectores**: Arreglos unidimensionales.
- **Matrices**: Arreglos bidimensionales (o de mayor dimensión), útiles para representar tablas o estructuras más complejas.

Ejemplo:

```
let vector = [1, 2, 3];
let matriz = [[1, 2], [3, 4]];

console.log(matriz[0][1]); // 2
```

7.2.2 Operaciones sobre Arreglos

JavaScript proporciona métodos útiles para manipular arreglos, como `push()`, `pop()`, `shift()`, `map()`, `filter()`, entre otros.

Ejemplo:

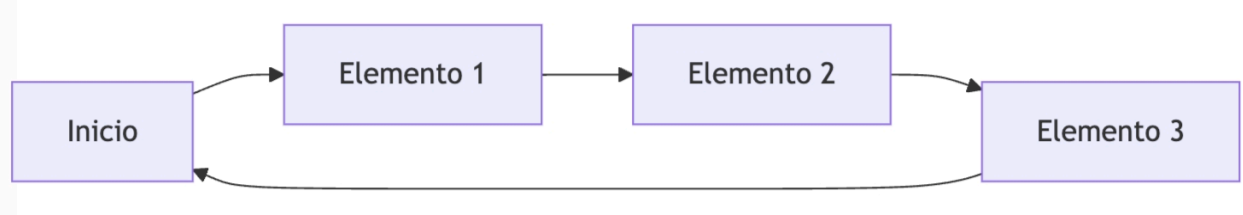
```
let numeros = [1, 2, 3, 4];

numeros.push(5); // Añade 5 al final del arreglo

numeros.shift(); // Elimina el primer elemento

let pares = numeros.filter(n => n % 2 === 0); // Filtra los números pares

console.log(pares); // [2, 4]
```

7.3 Listas

Las **listas** son estructuras de datos dinámicas que pueden almacenar un número variable de elementos. A diferencia de los arreglos, las listas no tienen un tamaño fijo.

7.3.1 Tipos de Listas (Simples, Dobles, Circulares)

- **Listas Simples**: Los elementos tienen un enlace al siguiente nodo.
- **Listas Dobles**: Cada nodo tiene enlaces al siguiente y al anterior.
- **Listas Circulares**: El último nodo se conecta al primero, formando un ciclo.



7.3.2 Operaciones sobre Listas

Las operaciones comunes incluyen insertar, eliminar, y recorrer los nodos de la lista.

7.4 Pilas

Una **pila** (stack) es una estructura de datos donde el último elemento en entrar es el primero en salir (LIFO - Last In, First Out). Se utiliza en problemas como el deshacer de acciones o la evaluación de expresiones.

Ejemplo:

```
let pila = [];

pila.push(1); // Agrega 1

pila.push(2); // Agrega 2

console.log(pila.pop()); // Elimina y devuelve 2
```

7.5 Colas

Una **cola** (queue) es una estructura de datos donde el primer elemento en entrar es el primero en salir (FIFO - First In, First Out). Se usa en sistemas de procesamiento por turnos.

Ejemplo:

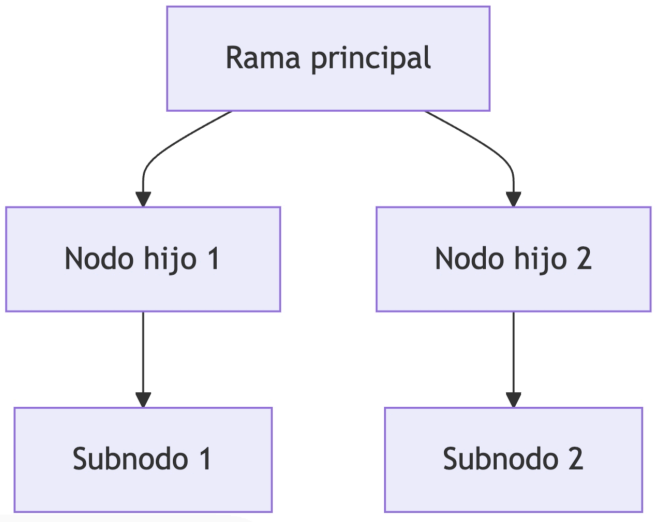
```
let cola = [];

cola.push(1); // Agrega 1

cola.push(2); // Agrega 2

console.log(cola.shift()); // Elimina y devuelve 1
```

7.6 Árboles

Los **árboles** son estructuras de datos jerárquicas donde cada nodo puede tener varios hijos.

7.6.1 Tipos de Árboles (Binarios, Ordenados, Multicamino)

- **Árbol Binario**: Cada nodo tiene como máximo dos hijos.
- **Árbol Ordenado**: Los nodos hijos se mantienen en un orden específico.
- **Árbol Multicamino**: Un nodo puede tener más de dos hijos.



7.7 Conjuntos

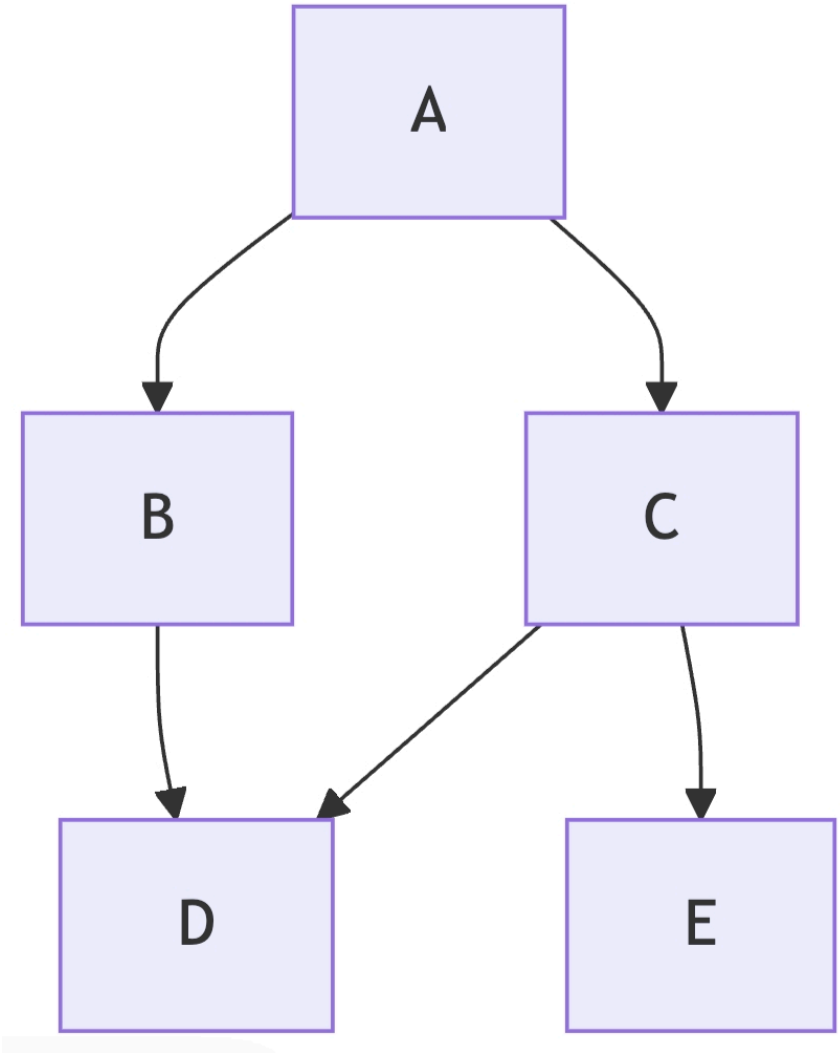
Un **conjunto** es una colección de elementos únicos. En JavaScript, los conjuntos se implementan con el tipo `Set`.

Ejemplo:

```
let conjunto = new Set([1, 2, 3, 3, 4]);

console.log(conjunto); // Set { 1, 2, 3, 4 }
```

7.8 Grafos

Un **grafo** es una estructura de datos que modela relaciones entre pares de nodos. Se utiliza en problemas como el análisis de redes sociales o la búsqueda de rutas.



8. Programación de Algoritmos

8.1 ¿Por qué Estudiar Algoritmos?

Los algoritmos son fundamentales para resolver problemas de manera eficiente. Un algoritmo es una serie de pasos que permiten alcanzar una solución específica, y comprender cómo

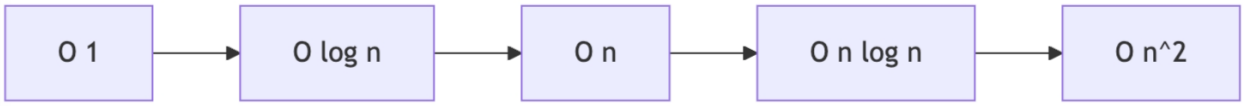
funcionan es esencial para optimizar el rendimiento de las aplicaciones. El estudio de algoritmos ayuda a los desarrolladores a:

- Optimizar el uso de recursos (tiempo y espacio).
- Mejorar la eficiencia de las soluciones.
- Solucionar problemas complejos con estrategias estructuradas.

8.2 Eficiencia y Complejidad de un Algoritmo (Complejidad Ciclomática y Big-O)

La **eficiencia de un algoritmo** se mide principalmente por dos factores: el tiempo que tarda en ejecutarse y el espacio que utiliza. La notación **Big-O** es utilizada para describir el comportamiento de un algoritmo cuando los datos de entrada crecen, definiendo el peor caso posible.

- **Big-O Notation**: O(1), O(n), O(n²), O(log n), etc.
  - **O(1)**: Tiempo constante, el tiempo de ejecución no depende del tamaño de los datos.
  - **O(n)**: Tiempo lineal, el tiempo de ejecución crece proporcionalmente con los datos.
  - **O(n²)**: Tiempo cuadrático, el tiempo de ejecución aumenta en función del cuadrado del número de datos.



La **complejidad ciclomática** mide la cantidad de caminos independientes a través del código, y ayuda a evaluar la mantenibilidad y el riesgo de errores en un algoritmo.

8.3 Algoritmos Recursivos

Un **algoritmo recursivo** es aquel que se llama a sí mismo para resolver un problema. Estos algoritmos son útiles cuando un problema puede dividirse en subproblemas más pequeños del mismo tipo.

Ejemplo de Recursión (Factorial):

```
function factorial(n) {

    if (n === 0) return 1;
    return n * factorial(n - 1);
}

console.log(factorial(5)); // 120
```

8.4 Algoritmos de Ordenamiento

Los **algoritmos de ordenamiento** son esenciales para organizar datos de manera eficiente. Algunos de los más comunes son:

- **Bubble Sort** (O(n²)): Intercambia elementos adyacentes si están en el orden incorrecto.
- **Merge Sort** (O(n log n)): Divide y conquista, dividiendo el arreglo en partes más pequeñas y luego fusionándolas.
- **Quick Sort** (O(n log n)): Selecciona un pivote y organiza los elementos en función de su relación con el pivote.

Ejemplo de Bubble Sort:

```
function bubbleSort(arr) {

    let swapped;
    do {
        swapped = false;

        for (let i = 0; i < arr.length - 1; i++) {
            if (arr[i] > arr[i + 1]) {
                [arr[i], arr[i + 1]] = [arr[i + 1], arr[i]];
                swapped = true;
            }
        }
```

```
    } while (swapped);
    return arr;
}

console.log(bubbleSort([5, 3, 8, 4])); // [3, 4, 5, 8]
```

8.5 Algoritmos de Búsqueda

Los algoritmos de búsqueda permiten encontrar elementos en una estructura de datos.

- **Búsqueda Lineal** (O(n)): Recorre cada elemento uno por uno hasta encontrar el elemento.
- **Búsqueda Binaria** (O(log n)): Requiere que los datos estén ordenados. Divide el conjunto de datos a la mitad repetidamente.

Ejemplo de Búsqueda Binaria:

```
function busquedaBinaria(arr, x) {

    let inicio = 0, fin = arr.length - 1;

    while (inicio <= fin) {
        let medio = Math.floor((inicio + fin) / 2);

        if (arr[medio] === x) return medio;
        if (arr[medio] < x) inicio = medio + 1;
        else fin = medio - 1;
    }
    return -1;
}

console.log(busquedaBinaria([1, 3, 5, 7, 9], 5)); // 2
```

8.6 Implementación en JavaScript

JavaScript permite la implementación de todos los algoritmos mencionados de manera eficiente gracias a sus capacidades de recursión, manejo de datos y optimización de memoria. Utilizar técnicas como el uso de promesas, funciones asíncronas, y manejo de estructuras de datos puede mejorar considerablemente el rendimiento.

8.7 Técnicas de Debugging

El debugging es fundamental para la identificación y resolución de errores en el código. Algunas técnicas incluyen:

- Uso de la función `console.log()` para rastrear la ejecución del código.
- Utilizar el **debugger** en el navegador o en Node.js.
- Monitorear el uso de memoria y rendimiento de los algoritmos para optimización.

8.8 Librerías que Facilitan el Trabajo con Estructuras de Datos

Algunas librerías útiles en JavaScript para trabajar con algoritmos y estructuras de datos incluyen:

- **Lodash**: Proporciona una serie de utilidades para trabajar con arreglos, objetos y más.
- **Underscore.js**: Similar a Lodash, proporciona funciones que facilitan la manipulación de estructuras de datos.
- **Moment.js**: Para la manipulación de fechas y tiempos.

9. Programación Funcional en JavaScript

9.1 El Paradigma de Programación Funcional

La **programación funcional** es un paradigma en el que las funciones son tratadas como ciudadanos de primera clase. Se centra en el uso de funciones puras, la inmutabilidad y la composición de funciones.

9.2 Enfoque Imperativo vs. Declarativo

- **Imperativo**: Define el "cómo" hacer algo paso a paso (ej. bucles).
- **Declarativo**: Define el "qué" se debe hacer, sin detallar cómo (ej. `map()`, `filter()`).

Ejemplo Imperativo vs Declarativo:

```
// Imperativo

let numeros = [1, 2, 3, 4];
let pares = [];

for (let i = 0; i < numeros.length; i++) {
    if (numeros[i] % 2 === 0) pares.push(numeros[i]);
}

console.log(pares); // [2, 4]
```

```
// Declarativo

let paresDeclarativo = numeros.filter(num => num % 2 === 0);

console.log(paresDeclarativo); // [2, 4]
```

9.3 Funciones como Objetos de Primera Clase

Las funciones en JavaScript son **ciudadanos de primera clase**, lo que significa que pueden ser asignadas a variables, pasadas como argumentos y retornadas por otras funciones.

Ejemplo:

```
let saludar = function() {
```

```
    console.log("Hola");
};

saludar(); // Hola
```

9.4 Funciones y Procedimientos

Las funciones en programación funcional no deben tener efectos colaterales, lo que significa que no deben modificar el estado global ni afectar otras variables fuera de su alcance.

9.5 La Función Flecha (`() => {}`)

Las **funciones flecha** proporcionan una sintaxis más concisa y manejan mejor el contexto de `this` en JavaScript.

Ejemplo:

```
let saludar = () => console.log("Hola");

saludar(); // Hola
```

9.6 Funciones Dentro de Objetos

JavaScript permite que los objetos contengan métodos (funciones dentro de un objeto).

Ejemplo:

```
let persona = {
    nombre: "Carlos",

    saludar: function() {
        console.log(`Hola, soy ${this.nombre}`);
    }
```

```
};

persona.saludar(); // Hola, soy Carlos
```

9.7 Recursión en Programación Funcional

La **recursión** en la programación funcional es un patrón común en el que una función se llama a sí misma para resolver problemas más pequeños.

9.8 Currying en Programación Funcional

El **currying** es una técnica que permite convertir una función con múltiples argumentos en una secuencia de funciones que reciben un argumento cada vez.

Ejemplo:

```
let suma = a => b => a + b;

console.log(suma(5)(3)); // 8
```

9.9 Composición de Funciones

La **composición de funciones** permite encadenar funciones para crear nuevas funcionalidades.

Ejemplo:

```
let sumar = x => x + 1;
let duplicar = x => x * 2;
let sumarYDuplicar = x => duplicar(sumar(x));

console.log(sumarYDuplicar(5)); // 12
```

10. Programación Orientada a Eventos y Programación Asíncrona

10.1 ¿Qué es un Evento?

Un **evento** es cualquier interacción del usuario o cambio en el sistema que puede desencadenar una acción. JavaScript es un lenguaje orientado a eventos, utilizado principalmente en el entorno web para manejar interacciones como clics, tecleos o cambios en la página.

10.2 Paradigma de la Orientación a Eventos

Este paradigma se basa en la respuesta a eventos, lo que permite a las aplicaciones ser reactivas. Los desarrolladores pueden usar **listeners** para capturar eventos y desencadenar funciones específicas.

10.3 Creación de Callbacks

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

Ejemplo:

```
function hacerAlgo(callback) {
    console.log("Haciendo algo...");
    callback();
}

hacerAlgo(() => console.log("Callback ejecutado"));
```

10.4 Creación de Funciones ASYNC/AWAIT

El uso de `async` y `await` simplifica el manejo de código asincrónico y evita el anidamiento excesivo de callbacks.

Ejemplo:

```
async function obtenerDatos() {

    let respuesta = await
fetch("https://jsonplaceholder.typicode.com/todos/1");
    let datos = await respuesta.json();
    console.log(datos);
}

obtenerDatos();
```

11. Programación Orientada a Objetos en JavaScript

11.1 ¿En Qué Consiste el Paradigma de Programación Orientada a Objetos (POO)?

La **POO** es un paradigma que organiza el código en torno a objetos, que representan entidades del mundo real. Los objetos tienen **propiedades** (datos) y **métodos** (funciones).

11.2 Principios de la POO

1. **Encapsulación**: Esconder detalles internos y exponer solo lo necesario.
2. **Abstracción**: Reducir la complejidad mostrando solo lo esencial.
3. **Herencia**: Permitir que las clases hereden características de otras.
4. **Polimorfismo**: Permitir que los métodos se comporten de manera diferente según el contexto.

11.3 Implementación de la POO en JavaScript

11.3.1 Objetos Literales

Los **objetos literales** permiten crear instancias de objetos fácilmente.

Ejemplo:

```
let persona = {
    nombre: "Ana",

    saludar: function() {
        console.log(`Hola, soy ${this.nombre}`);
    }
};

persona.saludar();
```

11.3.2 Método `Create`

`Object.create()` permite crear nuevos objetos utilizando un prototipo existente.

Ejemplo:

```
let animal = {
    sonido: "roar",

    hacerSonido: function() {
        console.log(this.sonido);
    }
};

let leon = Object.create(animal);
leon.sonido = "grrr";
leon.hacerSonido(); // "grrr"
```

11.3.3 Función Constructor

La **función constructora** es un patrón que permite crear múltiples objetos similares utilizando una plantilla común.

Ejemplo:

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

let juan = new Persona("Juan", 30);
console.log(juan.nombre); // "Juan"
```

11.3.4 Herencia por Prototipos

JavaScript utiliza un sistema de herencia basado en prototipos, donde los objetos pueden heredar propiedades y métodos de otros objetos.

11.3.5 Definición de Clases con ECMAScript 2015

Con ES6, JavaScript introdujo la sintaxis de **clases** para facilitar la programación orientada a objetos.

Ejemplo:

```
class Animal {

    constructor(nombre) {
        this.nombre = nombre;
    }

    hacerSonido() {
        console.log(`${this.nombre} hace un sonido.`);
    }
```

```
}

let perro = new Animal("Perro");
perro.hacerSonido(); // "Perro hace un sonido."
```

11.3.6 Definición de Subclases

Las subclases permiten heredar características de otra clase.

Ejemplo:

```
class Perro extends Animal {

    hacerSonido() {
        console.log(`${this.nombre} ladra.`);
    }
}

let miPerro = new Perro("Fido");
miPerro.hacerSonido(); // "Fido ladra."
```

12. Programación Orientada a Objetos en JavaScript

La **Programación Orientada a Objetos** (POO) es un paradigma de programación que organiza el software en torno a "objetos". Estos objetos son instancias de clases, que representan entidades o conceptos del mundo real con atributos (propiedades) y comportamientos (métodos).

12.1 En Qué Consiste el Paradigma de Programación Orientada a Objetos

El paradigma de la programación orientada a objetos se basa en la creación de objetos que interactúan entre sí para realizar tareas. En este paradigma, los objetos son estructuras que encapsulan datos y funciones relacionadas. Las principales características de este enfoque incluyen:

- **Abstracción**: Permite ocultar los detalles de implementación y exponer solo lo esencial.
- **Encapsulación**: Restringe el acceso directo a ciertos componentes de un objeto.
- **Herencia**: Permite que un objeto o clase derive características de otro.
- **Polimorfismo**: Permite que las clases tengan diferentes implementaciones de un método, dependiendo del contexto.

Los objetos en JavaScript se crean con propiedades (datos) y métodos (funciones), permitiendo estructurar mejor el código, reutilizar componentes y facilitar el mantenimiento.

12.2 Principios de la POO

Los principios fundamentales de la programación orientada a objetos son:

12.2.1 Encapsulación

La **encapsulación** es el mecanismo mediante el cual se ocultan los detalles internos de un objeto, exponiendo solo lo necesario. Esto protege los datos del acceso no autorizado y ayuda a mantener un código limpio y modular.

Ejemplo de Encapsulación:

```
class Persona {

    constructor(nombre, edad) {
        let _nombre = nombre; // Encapsulamos la propiedad
        this.edad = edad;
        this.getNombre = function() {
            return _nombre; // Método para acceder a la propiedad
```

```
encapsulada

        };
    }
}

const persona = new Persona("Carlos", 30);

console.log(persona.getNombre()); // Carlos
console.log(persona._nombre); // undefined, porque está encapsulada
```

En este ejemplo, el nombre está encapsulado dentro de la clase y solo se puede acceder a través del método `getNombre()`.

12.2.2 Abstracción

La **abstracción** implica la creación de modelos simplificados que representan entidades del mundo real. En POO, la abstracción permite enfocarse en los aspectos relevantes de un objeto, ignorando los detalles innecesarios.

Ejemplo de Abstracción:

```
class Vehiculo {

    constructor(marca, modelo) {
        this.marca = marca;
        this.modelo = modelo;
    }

    encender() {
        console.log(`${this.marca} ${this.modelo} está encendido.`);
    }
}

const miCarro = new Vehiculo("Toyota", "Corolla");
miCarro.encender(); // Toyota Corolla está encendido.
```

Aquí, `Vehiculo` es una abstracción que representa cualquier tipo de vehículo, y podemos añadir más detalles específicos en subclases.

12.2.3 Herencia

La **herencia** permite que una clase derive de otra, heredando sus propiedades y métodos. Esto promueve la reutilización del código y la creación de relaciones jerárquicas.

Ejemplo de Herencia:

```
class Animal {

    constructor(nombre) {
        this.nombre = nombre;
    }

    hacerSonido() {
        console.log(`${this.nombre} hace un sonido.`);
    }
}

class Perro extends Animal {

    hacerSonido() {
        console.log(`${this.nombre} ladra.`);
    }
}

const miPerro = new Perro("Fido");
miPerro.hacerSonido(); // Fido ladra.
```

En este ejemplo, `Perro` hereda de `Animal` y redefine el método `hacerSonido()` para especificar el sonido que hace un perro.

12.2.4 Polimorfismo

El **polimorfismo** permite que diferentes clases respondan a la misma acción de distintas formas. El método que se ejecuta depende de la clase del objeto.

Ejemplo de Polimorfismo:

```
class Animal {

    hacerSonido() {
        console.log("El animal hace un sonido.");
    }
}

class Gato extends Animal {

    hacerSonido() {
        console.log("El gato maúlla.");
    }
}

class Perro extends Animal {

    hacerSonido() {
        console.log("El perro ladra.");
    }
}

const animales = [new Gato(), new Perro()];
animales.forEach(animal => animal.hacerSonido());

// "El gato maúlla."
// "El perro ladra."
```

Aquí, tanto `Gato` como `Perro` sobrescriben el método `hacerSonido()` de `Animal`, mostrando el comportamiento específico para cada clase.

12.3 Implementación de la POO en JavaScript

JavaScript, aunque inicialmente no fue diseñado con POO en mente, soporta la programación orientada a objetos mediante varios enfoques. Existen diferentes formas de implementar la POO en JavaScript:

12.3.1 Objetos Literales

Los **objetos literales** son la forma más sencilla de crear objetos en JavaScript. Estos no tienen un constructor explícito ni herencia, pero permiten crear estructuras simples.

Ejemplo de Objeto Literal:

```
let persona = {
    nombre: "Juan",
    edad: 30,

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

persona.saludar(); // Hola, soy Juan
```

Los objetos literales son útiles para estructuras simples, pero carecen de las capacidades avanzadas de la POO, como la herencia.

12.3.2 Método `Object.create()`

El método **`Object.create()`** permite crear nuevos objetos utilizando un prototipo existente. Este enfoque es útil para crear objetos que compartan propiedades y métodos de un objeto base.

Ejemplo:

```
let animal = {
    hacerSonido: function() {
        console.log("El animal hace un sonido.");
    }
};

let perro = Object.create(animal);

perro.hacerSonido(); // El animal hace un sonido
```

Aquí, el objeto `perro` hereda el método `hacerSonido` del objeto `animal`.

12.3.3 Función Constructora

Las **funciones constructoras** son un patrón tradicional en JavaScript para crear múltiples objetos similares. Una función constructora actúa como una "plantilla" para crear objetos mediante el uso de la palabra clave `new`.

Ejemplo:

```
function Persona(nombre, edad) {

    this.nombre = nombre;
    this.edad = edad;

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

const juan = new Persona("Juan", 25);

juan.saludar(); // Hola, me llamo Juan y tengo 25 años.
```

En este ejemplo, `Persona` es una función constructora que crea objetos con las propiedades `nombre` y `edad`.

12.3.4 Herencia por Prototipos

La **herencia por prototipos** es el mecanismo fundamental de herencia en JavaScript. Cada objeto tiene una referencia interna hacia otro objeto llamado **prototipo**, y hereda sus propiedades y métodos.

Ejemplo:

```
function Animal(nombre) {
    this.nombre = nombre;
}

Animal.prototype.hacerSonido = function() {
    console.log(`${this.nombre} hace un sonido.`);
};

function Perro(nombre) {
    Animal.call(this, nombre); // Heredar constructor
}

Perro.prototype = Object.create(Animal.prototype); // Heredar métodos

Perro.prototype.constructor = Perro;

const fido = new Perro("Fido");

fido.hacerSonido(); // Fido hace un sonido.
```

En este ejemplo, `Perro` hereda el comportamiento de `Animal` utilizando la herencia basada en prototipos.

12.4 Implementación de la POO con ECMAScript 2015 (ES6)

A partir de **ES6**, JavaScript introdujo una nueva sintaxis más cercana a la programación orientada a objetos clásica, facilitando la creación de clases y subclases.

12.4.1 Definición de Clases

En ES6, una clase es simplemente una función, pero con una sintaxis más clara y orientada a objetos. Las clases contienen constructores y métodos.

Ejemplo:

```
class Persona {

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

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

const juan = new Persona("Juan", 30);

juan.saludar(); // Hola, me llamo Juan y tengo 30 años.
```

En este ejemplo, `Persona` es una clase que contiene un constructor y un método `saludar`.

12.4.2 Definición de Subclases

Las subclases permiten heredar características de una clase base, utilizando la palabra clave `extends`. El método `super()` es utilizado para llamar al constructor de la clase base.

Ejemplo de Subclase:

```
class Animal {

    constructor(nombre) {
        this.nombre = nombre;
    }

    hacerSonido() {
        console.log(`${this.nombre} hace un sonido.`);
    }
}

class Perro extends Animal {

    constructor(nombre, raza) {
        super(nombre); // Llama al constructor de la clase base
        this.raza = raza;
    }

    hacerSonido() {
        console.log(`${this.nombre} ladra.`);
    }
}

const miPerro = new Perro("Rex", "Labrador");

miPerro.hacerSonido(); // Rex ladra.
```

Aquí, `Perro` hereda de `Animal` y sobrescribe el método `hacerSonido()`.

12.4.3 Creación de Instancias

Para crear una instancia de una clase en JavaScript, se utiliza la palabra clave `new`. Esto llama al constructor de la clase, inicializando los valores de las propiedades.

Ejemplo:

```
let perro = new Perro("Fido", "Pastor Alemán");

console.log(perro.nombre); // Fido
console.log(perro.raza);   // Pastor Alemán
```

En este ejemplo, creamos una nueva instancia de la clase `Perro`, que hereda de `Animal`, y accedemos a las propiedades de la instancia.

Material de Referencia

Libros:

- **"JavaScript: The Good Parts"** de Douglas Crockford
  Este libro clásico proporciona una visión profunda de las características más poderosas de **JavaScript** y cómo utilizarlas de manera efectiva, evitando los aspectos más problemáticos del lenguaje.

- **"Eloquent JavaScript"** de Marijn Haverbeke
  Un recurso excelente para aprender sobre las bases y aspectos avanzados de **JavaScript**, con ejemplos interactivos y proyectos. El capítulo sobre la **programación funcional** y la **programación orientada a objetos** es particularmente útil.

- **"You Don't Know JS (Yet)"** de Kyle Simpson
  Esta serie de libros profundiza en los conceptos más complejos de **JavaScript**, como el manejo del **scope**, **closures**, y las funciones asíncronas. Es ideal para desarrolladores que desean profundizar en las peculiaridades del lenguaje.

- **"JavaScript Patterns"** de Stoyan Stefanov
  Un libro que aborda patrones de diseño en **JavaScript**, con enfoques sólidos para aplicar **POO**, encapsulación, y módulos. Es útil para quienes buscan escribir código más estructurado y mantenible.

Enlaces a Recursos Online:

- **JavaScript | MDN (mozilla.org)**
  Un recurso fundamental para cualquier desarrollador, ofrece documentación completa y ejemplos sobre todos los temas de **JavaScript**, desde los básicos hasta los avanzados. Es una referencia confiable y detallada que cubre cada aspecto del lenguaje.

- **The Modern JavaScript Tutorial**
  Un tutorial completo y detallado que cubre desde lo básico hasta conceptos avanzados, incluyendo **programación orientada a objetos** y **asíncrona**. Es ideal para aprender las características modernas del lenguaje.

- **Lodash Documentation**
  Lodash es una librería que facilita el trabajo con datos en **JavaScript**. Su documentación es una referencia útil para la manipulación de estructuras de datos como **arrays** y **objetos**, proporcionando métodos avanzados y optimizados.

Videos Recomendados:

- **JavaScript: Understanding the Weird Parts (2024 Edition) | Udemy**
  Un curso que explora los aspectos más profundos y "extraños" de **JavaScript**, incluyendo cómo funciona el motor de ejecución, el **event loop**, y el **scope**. Ideal para quienes desean una comprensión sólida del comportamiento del lenguaje.

- **YouTube: ES6 Features**
  Un video explicativo sobre las nuevas características de **ES6**, incluyendo clases, módulos, y funciones flechas. Ayuda a familiarizarse con las herramientas modernas de **JavaScript**.

- **YouTube: Callbacks, Promises, Async/Await**
  Un video claro que explica el manejo de funciones asíncronas en **JavaScript** y cómo usar **async/await** de manera efectiva. Es perfecto para aprender a gestionar el flujo de código asíncrono.

- **YouTube: Object-Oriented Programming**
  Un tutorial detallado sobre cómo implementar **POO** en **JavaScript**, desde objetos literales hasta el uso de clases en **ES6**

CORFO
BecasCapital Humano

Adalid
aspasia
LA FORMACIÓN DE TU FUTURO

MÓDULO 3

PROGRAMACIÓN AVANZADA EN JAVASCRIPT