Tema 2. Javascript ES6+. Volumen 2.

Ciclo: Desarrollo de Aplicaciones Multiplataforma. Módulo: Desarrollo de Interfaces.

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)**

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 Clases

Las clases en JavaScript son una forma especial de funciones que permiten crear objetos utilizando un modelo basado en la orientación a objetos. Fueron introducidas en ES6 (ECMAScript 2015) y proporcionan una sintaxis más clara y organizada para trabajar con funciones constructoras y objetos prototípicos.

Antes de ES6, la creación de objetos y la herencia se lograban mediante funciones constructoras y el uso de prototipos. Las clases son básicamente "azúcar sintáctico" sobre esta funcionalidad existente, es decir, no introducen un nuevo paradigma de programación, sino que hacen que el proceso sea más sencillo y legible.

Antes de ES6, se utilizaban funciones constructoras para crear objetos. Aquí un ejemplo de cómo se hacía:

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

// Añadiendo un método al prototipo
Persona.prototype.saludar = function() {
    console.log(`Hola, mi nombre es ${this.nombre} y tengo ${this.edad} años.`);
};

// Crear una instancia de Persona
const juan = new Persona('Juan', 30);
juan.saludar(); // Hola, mi nombre es Juan y tengo 30 años.
```

Con la llegada de ES6, la sintaxis de las clases se introdujo para hacer este proceso más intuitivo:

```
// Definición de una clase
class Persona {
    constructor(nombre, edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

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

// Crear una instancia de Persona
const juan = new Persona('Juan', 30);
juan.saludar(); // Hola, mi nombre es Juan y tengo 30 años.
```

Las clases en JavaScript tienen varias ventajas sobre el uso de funciones constructoras tradicionales:

- Sintaxis más limpia y clara: La sintaxis de clases es más cercana a la de otros lenguajes de programación orientados a objetos como Java, C++, etc., lo que facilita la comprensión para los

desarrolladores que provienen de esos entornos.
- Modularidad y organización: Las clases permiten organizar mejor el código, agrupando propiedades y métodos relacionados en un solo lugar.
- Soporte nativo para herencia: La herencia con clases es más directa y fácil de entender que con prototipos.

Imaginemos que queremos modelar una clase Animal que tenga una propiedad nombre y un método hacerSonido. Luego, queremos extender esa clase para crear una clase Perro que herede de Animal y tenga un método adicional ladrar.

Sin usar clases (Función constructora y prototipos)

```
// Función constructora para Animal
function Animal(nombre) {
    this.nombre = nombre;
}

Animal.prototype.hacerSonido = function() {
    console.log('El animal hace un sonido.');
};

// Función constructora para Perro que hereda de Animal
function Perro(nombre) {
    Animal.call(this, nombre); // Llamada al constructor de Animal
}

// Heredando el prototipo de Animal
Perro.prototype = Object.create(Animal.prototype);
Perro.prototype.constructor = Perro;

// Añadiendo un nuevo método a Perro
Perro.prototype.ladrar = function() {
    console.log('¡Guau guau!');
};

// Crear una instancia de Perro
const miPerro = new Perro('Firulais');
miPerro.hacerSonido(); // El animal hace un sonido.
miPerro.ladrar(); // ¡Guau guau!
```

Usando clases en ES6

```
// Definición de la clase Animal
class Animal {
    constructor(nombre) {
        this.nombre = nombre;
    }

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

// Definición de la clase Perro que hereda de Animal
class Perro extends Animal {
    ladrar() {
        console.log('¡Guau guau!');
    }
```

```
}

// Crear una instancia de Perro
const miPerro = new Perro('Firulais');
miPerro.hacerSonido(); // El animal hace un sonido.
miPerro.ladrar(); // ¡Guau guau!
```

1.1 Sintaxis Básica de una Clase

En JavaScript, una clase se define utilizando la palabra clave class, seguida del nombre de la clase. El nombre de la clase generalmente se escribe en PascalCase, es decir, cada palabra comienza con una letra mayúscula.

```
// Definición de una clase básica
class Persona {
    // Definición del constructor
    constructor(nombre, edad) {
        this.nombre = nombre; // Propiedad nombre
        this.edad = edad;     // Propiedad edad
    }
}
```

En este ejemplo, hemos declarado una clase Persona con un constructor que acepta dos parámetros (nombre y edad) y asigna esos valores a las propiedades de la instancia (this.nombre y this.edad).

Dentro de una clase, los métodos se definen como funciones normales, pero sin la palabra clave function. Estos métodos se agregan automáticamente al prototipo del objeto cuando se crea una instancia de la clase.

```
// Clase Persona con un método
class Persona {
    constructor(nombre, edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    // Método saludar
    saludar() {
        console.log(`Hola, mi nombre es ${this.nombre} y tengo ${this.edad} años.`);
    }
}

// Crear una instancia de Persona
const juan = new Persona('Juan', 30);
juan.saludar(); // Hola, mi nombre es Juan y tengo 30 años.
```

En este ejemplo, hemos añadido un método saludar() a la clase Persona, que puede ser llamado desde cualquier instancia de Persona.

- Constructor

El constructor es un método especial que se ejecuta automáticamente cuando se crea una nueva instancia de la clase. Es aquí donde se inicializan las propiedades del objeto.

```
// Clase con un constructor
class Coche {
    constructor(marca, modelo, año) {
        this.marca = marca;
        this.modelo = modelo;
        this.año = año;
    }

    // Método para describir el coche
    describir() {
        console.log(`Este coche es un ${this.marca} ${this.modelo} del año ${this.año}.`);
    }
}

// Crear una instancia de Coche
const miCoche = new Coche('Toyota', 'Corolla', 2021);
miCoche.describir(); // Este coche es un Toyota Corolla del año 2021.
```

En este ejemplo, el constructor de la clase Coche toma tres parámetros (marca, modelo, año) y los asigna a las propiedades de la instancia (this.marca, this.modelo, this.año). Luego, se define un método describir() para mostrar información sobre el coche.

Supongamos que queremos definir una clase Rectangulo que tenga métodos para calcular el área y el perímetro de un rectángulo.

```
// Definición de la clase Rectangulo
class Rectangulo {
    constructor(ancho, alto) {
        this.ancho = ancho;
        this.alto = alto;
    }

    // Método para calcular el área
    calcularArea() {
        return this.ancho * this.alto;
    }

    // Método para calcular el perímetro
    calcularPerimetro() {
        return 2 * (this.ancho + this.alto);
    }
}

// Crear una instancia de Rectangulo
const miRectangulo = new Rectangulo(10, 5);

console.log(`Área: ${miRectangulo.calcularArea()}`); // Área: 50
console.log(`Perímetro: ${miRectangulo.calcularPerimetro()}`); // Perímetro: 30
```

- Clases como expresiones:

Las clases también pueden ser definidas como expresiones, y pueden ser anónimas.

```
const Rectangulo = class {
    constructor(ancho, alto) {
        this.ancho = ancho;
        this.alto = alto;
```

```
    }

    calcularArea() {
        return this.ancho * this.alto;
    }
};

const miRectangulo = new Rectangulo(10, 5);
console.log(miRectangulo.calcularArea()); // 50
```

- Clases con nombres personalizados:

Incluso como expresión, una clase puede tener un nombre, lo que es útil para propósitos de depuración.

```
const Cuadrado = class Cuadrado {
    constructor(lado) {
        this.lado = lado;
    }

    calcularArea() {
        return this.lado * this.lado;
    }
};

const miCuadrado = new Cuadrado(4);
console.log(miCuadrado.calcularArea()); // 16
```

- Clases son “hoisted” de forma diferente a las funciones:

A diferencia de las funciones, las clases no son “hoisted”. Esto significa que no puedes instanciar una clase antes de su declaración.

```
const miCoche = new Coche('Ford', 'Mustang', 1969); // Error: Cannot access 'Coche' before
    initialization

class Coche {
    constructor(marca, modelo, año) {
        this.marca = marca;
        this.modelo = modelo;
        this.año = año;
    }
}
```

1.2 Creación de Instancias

Para crear una instancia de una clase en JavaScript, se utiliza la palabra clave new seguida del nombre de la clase y los paréntesis. Cuando se llama a new, se realiza lo siguiente:

- Se crea un nuevo objeto.
- El constructor de la clase se ejecuta.
- Se asigna el valor de this dentro del constructor al nuevo objeto creado.

- El nuevo objeto se enlaza al prototipo de la clase.
- Se devuelve el nuevo objeto.

Consideremos la siguiente clase Persona:

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

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

// Crear una instancia de la clase Persona
const juan = new Persona('Juan', 30);

// Llamar al método saludar en la instancia
juan.saludar(); // Hola, mi nombre es Juan y tengo 30 años.
```

En este ejemplo, hemos creado una instancia de Persona llamada juan. Al hacerlo, el constructor de la clase Persona se ejecuta, asignando el nombre y la edad al objeto juan. Luego, podemos llamar al método saludar en juan, que usará las propiedades definidas en el constructor.

- Diferencias entre Instancias y la Clase

Cada vez que se crea una instancia de una clase, se crea un nuevo objeto con sus propias propiedades y métodos. Sin embargo, estos objetos comparten el mismo prototipo, lo que significa que los métodos definidos en la clase se comparten entre todas las instancias, pero las propiedades definidas dentro del constructor son únicas para cada instancia.

```
class Coche {
    constructor(marca, modelo, año) {
        this.marca = marca;
        this.modelo = modelo;
        this.año = año;
    }

    describir() {
        console.log(`Este coche es un ${this.marca} ${this.modelo} del año ${this.año}.`);
    }
}

// Crear varias instancias de Coche
const coche1 = new Coche('Toyota', 'Corolla', 2021);
const coche2 = new Coche('Honda', 'Civic', 2020);

// Llamar a describir en diferentes instancias
coche1.describir(); // Este coche es un Toyota Corolla del año 2021.
coche2.describir(); // Este coche es un Honda Civic del año 2020.
```

En este caso, coche1 y coche2 son instancias independientes de la clase Coche. Aunque comparten los mismos métodos (describir), sus propiedades (marca, modelo, año) son diferentes y específicas para cada instancia.

- Acceso y Modificación de Propiedades en Instancias

Las propiedades de una instancia se pueden acceder y modificar directamente utilizando la notación de punto (.).

```
// Crear una instancia de Persona
const maria = new Persona('Maria', 28);

console.log(maria.nombre); // Maria
console.log(maria.edad);   // 28

// Modificar propiedades
maria.nombre = 'Maria Garcia';
maria.edad = 29;

console.log(maria.nombre); // Maria Garcia
console.log(maria.edad);   // 29
```

Aquí, hemos accedido y modificado directamente las propiedades nombre y edad de la instancia maria.

Supongamos que estamos modelando un sistema para una biblioteca. Definimos una clase Libro y creamos varias instancias para representar diferentes libros.

```
// Definición de la clase Libro
class Libro {
    constructor(titulo, autor, anioPublicacion) {
        this.titulo = titulo;
        this.autor = autor;
        this.anioPublicacion = anioPublicacion;
    }

    describir() {
        console.log(`${this.titulo}, escrito por ${this.autor} en ${this.anioPublicacion}.`);
    }
}

// Crear instancias de la clase Libro
const libro1 = new Libro('Cien Años de Soledad', 'Gabriel García Márquez', 1967);
const libro2 = new Libro('Don Quijote de la Mancha', 'Miguel de Cervantes', 1605);
const libro3 = new Libro('1984', 'George Orwell', 1949);

// Describir cada libro
libro1.describir(); // Cien Años de Soledad, escrito por Gabriel García Márquez en 1967.
libro2.describir(); // Don Quijote de la Mancha, escrito por Miguel de Cervantes en 1605.
libro3.describir(); // 1984, escrito por George Orwell en 1949.
```

En este ejemplo, cada instancia de Libro representa un libro diferente, con su propio título, autor y año de publicación. Aunque todas las instancias comparten el método describir, cada una trabaja con los datos específicos del libro que representa.

1.3 Propiedades de Clase

Las propiedades de instancia son aquellas que se definen dentro del constructor de una clase. Cada vez que se crea una nueva instancia de la clase, estas propiedades se inicializan de acuerdo con los va-

lores pasados al constructor o con valores predeterminados. Ejemplo de Propiedades de Instancia

Consideremos la clase Persona con propiedades como nombre y edad:

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

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

// Crear instancias de Persona
const persona1 = new Persona('Carlos', 25);
const persona2 = new Persona('Ana', 30);

persona1.saludar(); // Hola, mi nombre es Carlos y tengo 25 años.
persona2.saludar(); // Hola, mi nombre es Ana y tengo 30 años.
```

1.3.1 Propiedades Estáticas (static)

Las propiedades estáticas son propiedades que pertenecen a la clase en sí, y no a las instancias de la clase. Para definir una propiedad estática, se utiliza la palabra clave static. Estas propiedades se pueden acceder directamente desde la clase sin necesidad de crear una instancia. Ejemplo de Propiedades Estáticas

Vamos a definir una propiedad estática en la clase Coche para contar el número de coches creados:

```
class Coche {
    static contadorCoches = 0;

    constructor(marca, modelo) {
        this.marca = marca;
        this.modelo = modelo;
        Coche.contadorCoches++; // Incrementar el contador cada vez que se crea una nueva
            instancia
    }

    describir() {
        console.log(`Este coche es un ${this.marca} ${this.modelo}.`);
    }

    static obtenerNumeroCoches() {
        return `Se han creado ${Coche.contadorCoches} coches.`;
    }
}

// Crear varias instancias de Coche
const coche1 = new Coche('Toyota', 'Corolla');
const coche2 = new Coche('Honda', 'Civic');
const coche3 = new Coche('Ford', 'Mustang');

// Acceder a la propiedad estática
console.log(Coche.obtenerNumeroCoches()); // Se han creado 3 coches.
```

En este ejemplo:

- contadorCoches es una propiedad estática que pertenece a la clase Coche y se utiliza para contar cuántos coches se han creado.
- obtenerNumeroCoches() es un método estático que devuelve el número total de coches creados.
- Este método se llama directamente desde la clase Coche, no desde una instancia.
- Diferencias entre Propiedades de Instancia y Propiedades Estáticas

Característica	Propiedades de Instancia	Propiedades Estáticas
Definición	Se definen dentro del constructor (this.propiedad)	Se definen con la palabra clave static
Pertenencia	Pertenece a cada instancia individual de la clase	Pertenece a la clase en sí
Acceso	Se accede a través de la instancia (instancia.propiedad)	Se accede directamente a través de la clase (Clase.propiedad)
Uso Común	Almacenar datos específicos de una instancia	Almacenar datos o funciones comunes a todas las instancias

Supongamos que estamos modelando un sistema de bibliotecas donde necesitamos contar el número total de libros creados, independientemente de la categoría a la que pertenecen.

```
class Libro {
    static totalLibros = 0;

    constructor(titulo, autor, categoria) {
        this.titulo = titulo;
        this.autor = autor;
        this.categoria = categoria;
        Libro.totalLibros++; // Incrementar el contador estático
    }

    describir() {
        console.log(`Título: ${this.titulo}, Autor: ${this.autor}, Categoría: ${this.
            categoria}`);
    }

    static obtenerTotalLibros() {
        return `Se han registrado ${Libro.totalLibros} libros en total.`;
    }
}

// Crear instancias de Libro
const libro1 = new Libro('El Principito', 'Antoine de Saint-Exupéry', 'Ficción');
```

```
const libro2 = new Libro('Cien Años de Soledad', 'Gabriel García Márquez', 'Realismo Mágico')
    ;
const libro3 = new Libro('Sapiens', 'Yuval Noah Harari', 'Historia');

// Llamar al método describir en cada libro
libro1.describir(); // Título: El Principito, Autor: Antoine de Saint-Exupéry, Categoría:
    Ficción
libro2.describir(); // Título: Cien Años de Soledad, Autor: Gabriel García Márquez, Categoría
    : Realismo Mágico
libro3.describir(); // Título: Sapiens, Autor: Yuval Noah Harari, Categoría: Historia

// Acceder a la propiedad estática
console.log(Libro.obtenerTotalLibros()); // Se han registrado 3 libros en total.
```

En este ejemplo:

- titulo, autor, y categoria son propiedades de instancia específicas para cada libro.
- totalLibros es una propiedad estática que cuenta el número total de libros creados.
- obtenerTotalLibros() es un método estático que devuelve el número total de libros registrados.
- Consideraciones y Buenas Prácticas
  - Uso de Propiedades Estáticas: Las propiedades estáticas son útiles cuando necesitas almacenar o acceder a datos que son comunes a todas las instancias, como contadores o configuraciones compartidas.
  - Modificación de Propiedades: Las propiedades de instancia pueden modificarse para cada objeto individual, mientras que las propiedades estáticas deben ser modificadas a nivel de la clase, lo cual afectará a todas las instancias de esa clase.
  - Encapsulamiento: Es recomendable usar propiedades estáticas con cuidado, ya que pueden llevar a diseños menos encapsulados si se abusa de ellas. En muchos casos, es preferible utilizar métodos de instancia y propiedades de instancia para mantener la flexibilidad y modularidad del código.

1.4 Métodos de Clase

Los métodos de instancia son funciones que se definen dentro de una clase y se asocian con una instancia específica de esa clase. Estos métodos se pueden llamar en cualquier objeto que sea una instancia de la clase. Cuando se llama a un método de instancia, this se refiere a la instancia actual del objeto.

Consideremos una clase Coche que tiene un método para describir el coche y otro para arrancarlo:

```
class Coche {
    constructor(marca, modelo) {
        this.marca = marca;
        this.modelo = modelo;
        this.encendido = false;
```

```
    }

    describir() {
        console.log(`Este coche es un ${this.marca} ${this.modelo}.`);
    }

    arrancar() {
        if (!this.encendido) {
            this.encendido = true;
            console.log(`${this.marca} ${this.modelo} está arrancado.`);
        } else {
            console.log(`${this.marca} ${this.modelo} ya está en marcha.`);
        }
    }

    apagar() {
        if (this.encendido) {
            this.encendido = false;
            console.log(`${this.marca} ${this.modelo} está apagado.`);
        } else {
            console.log(`${this.marca} ${this.modelo} ya está apagado.`);
        }
    }
}

// Crear una instancia de Coche
const miCoche = new Coche('Toyota', 'Corolla');

// Llamar a los métodos de instancia
miCoche.describir();  // Este coche es un Toyota Corolla.
miCoche.arrancar();   // Toyota Corolla está arrancado.
miCoche.arrancar();   // Toyota Corolla ya está en marcha.
miCoche.apagar();     // Toyota Corolla está apagado.
```

1.4.1 Métodos Estáticos

Los métodos estáticos son funciones que pertenecen a la clase en sí, y no a las instancias de la clase. Se definen usando la palabra clave static. A diferencia de los métodos de instancia, los métodos estáticos no pueden acceder a las propiedades de la instancia (this dentro de un método estático se refiere a la clase, no a la instancia).

Vamos a definir un método estático en la clase Coche que cuente cuántos coches han sido creados:

```
class Coche {
    static totalCoches = 0;

    constructor(marca, modelo) {
        this.marca = marca;
        this.modelo = modelo;
        Coche.totalCoches++; // Incrementar el contador de coches creados
    }

    describir() {
        console.log(`Este coche es un ${this.marca} ${this.modelo}.`);
    }

    static obtenerTotalCoches() {
        return `Se han creado ${Coche.totalCoches} coches en total.`;
```

```
    }
}

// Crear varias instancias de Coche
const coche1 = new Coche('Honda', 'Civic');
const coche2 = new Coche('Ford', 'Mustang');
const coche3 = new Coche('Chevrolet', 'Camaro');

// Llamar al método estático sin instanciar
console.log(Coche.obtenerTotalCoches()); // Se han creado 3 coches en total.
```

1.4.2 Comparación entre Métodos de Instancia y Métodos Estáticos

Característica	Métodos de Instancia	Métodos Estáticos
Definición	Se definen dentro de la clase sin la palabra clave **`static`**	Se definen con la palabra clave **`static`**
Pertenencia	Pertenece a la instancia de la clase	Pertenece a la clase en sí
Acceso a **`this`**	**`this`** se refiere a la instancia actual del objeto	**`this`** se refiere a la clase (no puede acceder a **`this`** de la instancia)
Uso Común	Operaciones que dependen de los datos específicos de una instancia	Funciones utilitarias o datos que no dependen de una instancia específica

Supongamos que estamos modelando un sistema de pago y queremos definir una clase Pago que tenga métodos para calcular el total de una compra y aplicar un descuento.

```
class Pago {
    constructor(subtotal) {
        this.subtotal = subtotal;
    }

    // Método de instancia
    calcularTotal() {
        return this.subtotal * 1.15; // Asumiendo un 15% de impuestos
    }

    // Método estático
    static aplicarDescuento(total, porcentajeDescuento) {
        return total - (total * (porcentajeDescuento / 100));
    }
}

// Crear una instancia de Pago
const miPago = new Pago(100);

// Llamar al método de instancia para calcular el total
const totalConImpuestos = miPago.calcularTotal();
console.log(`Total con impuestos: $${totalConImpuestos}`); // Total con impuestos: $115

// Llamar al método estático para aplicar un descuento
```

```
const totalConDescuento = Pago.aplicarDescuento(totalConImpuestos, 10);
console.log(`Total con descuento: $${totalConDescuento}`); // Total con descuento: $103.5
```

En este ejemplo:

- calcularTotal es un método de instancia que calcula el total de una compra incluyendo impuestos, basado en el subtotal de esa instancia.
- aplicarDescuento es un método estático que aplica un porcentaje de descuento al total. No depende de ninguna instancia específica y puede ser llamado directamente desde la clase Pago.
- Buenas Prácticas al Definir Métodos
- Uso de Métodos Estáticos: Los métodos estáticos deben ser utilizados para funcionalidades que no dependen de los datos de una instancia específica, como funciones de utilidad o cálculo general.
- Acceso a Propiedades de Instancia: Los métodos de instancia son ideales para trabajar con los datos de la instancia y manipular el estado del objeto.
- Modularidad y Claridad: Definir métodos de instancia y estáticos en función de su responsabilidad ayuda a mantener el código organizado y fácil de entender.

1.5 Encapsulamiento

1.5.1 Propiedades y Métodos Públicos

En JavaScript, todas las propiedades y métodos de una clase son, por defecto, públicos. Esto significa que se pueden acceder y modificar directamente desde fuera de la clase. Sin embargo, en muchos casos, se desea controlar el acceso a ciertas partes de un objeto para proteger su integridad o para esconder detalles de implementación que no son relevantes para el usuario del objeto.

Considera la siguiente clase CuentaBancaria:

```
class CuentaBancaria {
    constructor(titular, saldoInicial) {
        this.titular = titular; // Propiedad pública
        this.saldo = saldoInicial; // Propiedad pública
    }

    depositar(cantidad) {
        this.saldo += cantidad; // Método público
        console.log(`Se depositaron ${cantidad} dólares. Nuevo saldo: ${this.saldo}`);
    }

    retirar(cantidad) {
        if (cantidad > this.saldo) {
            console.log('Fondos insuficientes.');
        } else {
            this.saldo -= cantidad; // Método público
```

```
            console.log(`Se retiraron ${cantidad} dólares. Nuevo saldo: ${this.saldo}`);
        }
    }

    consultarSaldo() {
        return `El saldo actual es: ${this.saldo} dólares.`;
    }
}

// Crear una instancia de CuentaBancaria
const cuenta = new CuentaBancaria('Juan Pérez', 1000);

// Acceder a métodos y propiedades públicos
cuenta.depositar(500);  // Se depositaron 500 dólares. Nuevo saldo: 1500
cuenta.retirar(200);    // Se retiraron 200 dólares. Nuevo saldo: 1300
console.log(cuenta.consultarSaldo()); // El saldo actual es: 1300 dólares.

cuenta.saldo=0;
console.log(cuenta.saldo); // Se puede modificar directamente.
```

En este ejemplo:

- titular y saldo son propiedades públicas que se pueden acceder y modificar desde cualquier lugar.
- Los métodos depositar, retirar, y consultarSaldo son públicos y pueden ser llamados desde fuera de la clase.

1.5.2 Propiedades y Métodos Privados

Para lograr el encapsulamiento, es posible definir propiedades y métodos como privados. Las propiedades y métodos privados no son accesibles ni modificables desde fuera de la clase, lo que permite controlar mejor cómo se interactúa con el objeto.

Desde la introducción de los campos privados en ES2022, puedes definir atributos o métodos privados dentro de una clase usando el prefijo #. Estos atributos son realmente privados y no pueden ser accedidos fuera de la clase.

Vamos a modificar la clase CuentaBancaria para encapsular la propiedad saldo:

```
class CuentaBancaria {
    #saldo; // Definición de una propiedad privada

    constructor(titular, saldoInicial) {
        this.titular = titular; // Propiedad pública
        this.#saldo = saldoInicial; // Propiedad privada
    }

    // Método privado para imprimir el saldo actual
    #imprimirSaldo() {
        console.log(`Saldo actual: ${this.#saldo} dólares.`);
    }

    depositar(cantidad) {
        this.#saldo += cantidad; // Método público que accede a la propiedad privada
```

```
        console.log(`Se depositaron ${cantidad} dólares.`);
        this.#imprimirSaldo(); // Llamada al método privado
    }

    retirar(cantidad) {
        if (cantidad > this.#saldo) {
            console.log('Fondos insuficientes.');
        } else {
            this.#saldo -= cantidad; // Método público que accede a la propiedad privada
            console.log(`Se retiraron ${cantidad} dólares.`);
            this.#imprimirSaldo(); // Llamada al método privado
        }
    }

    consultarSaldo() {
        return `El saldo actual es: ${this.#saldo} dólares.`; // Acceso controlado a la
            propiedad privada
    }
}

// Crear una instancia de CuentaBancaria
const cuenta = new CuentaBancaria('Juan Pérez', 1000);

// Intentar acceder a la propiedad o método privado desde fuera (genera un error)
console.log(cuenta.#saldo); // Error: Private field '#saldo' must be declared in an enclosing
     class
console.log(cuenta.#imprimirSaldo()); // Error: Private field '#imprimirSaldo' must be
    declared in an enclosing class
```

- Método Privado #imprimirSaldo: Este método es privado y solo puede ser invocado dentro de la clase CuentaBancaria. Se utiliza para imprimir el saldo después de realizar una operación de depósito o retiro, asegurando que cada vez que se modifique el saldo, se informe del estado actual al usuario.
- Acceso Controlado: Ningún método o propiedad que comience con # puede ser accedido desde fuera de la clase, lo que proporciona un nivel adicional de seguridad y encapsulamiento, garantizando que la lógica interna y el estado de la clase solo se modifiquen a través de métodos controlados.

1.5.3 Beneficios del Encapsulamiento

El encapsulamiento es una técnica fundamental en la programación orientada a objetos que ofrece varios beneficios:

- Protección de Datos: Al hacer que ciertas propiedades sean privadas, se evita que sean modificadas directamente desde fuera de la clase, lo que protege la integridad de los datos.
- Control sobre el Acceso: Permite controlar cómo y cuándo se puede acceder a ciertas partes de un objeto, lo que puede ser crucial para mantener la consistencia del estado del objeto.
- Mantenimiento y Flexibilidad: Al esconder detalles de implementación, los cambios internos en la clase no afectarán a los usuarios de la clase, siempre y cuando la interfaz pública (métodos

públicos) permanezca constante.

Supongamos que estamos desarrollando un sistema de seguridad donde queremos asegurarnos de que los intentos de acceso no autorizado sean registrados sin que los usuarios del sistema puedan manipular directamente los registros.

```
class SistemaSeguridad {
    #intentosFallidos; // Propiedad privada

    constructor() {
        this.#intentosFallidos = 0;
    }

    acceder(codigo) {
        if (codigo === '1234') {
            console.log('Acceso concedido.');
            this.#intentosFallidos = 0; // Resetear contador en caso de éxito
        } else {
            this.#intentosFallidos++; // Incrementar contador de intentos fallidos
            console.log('Acceso denegado.');
        }
    }

    obtenerIntentosFallidos() {
        return `Intentos fallidos: ${this.#intentosFallidos}`;
    }
}

// Crear una instancia de SistemaSeguridad
const sistema = new SistemaSeguridad();

// Intentos de acceso
sistema.acceder('1111'); // Acceso denegado.
sistema.acceder('2222'); // Acceso denegado.
sistema.acceder('1234'); // Acceso concedido.

// Consultar los intentos fallidos
console.log(sistema.obtenerIntentosFallidos()); // Intentos fallidos: 0
```

En este ejemplo:

- #intentosFallidos es una propiedad privada que rastrea los intentos fallidos de acceso.
- Los métodos públicos acceder y obtenerIntentosFallidos proporcionan una forma controlada de interactuar con la propiedad privada.

1.5.4 Limitaciones y Alternativas al Encapsulamiento

- Compatibilidad: El uso de propiedades privadas con # es una característica relativamente reciente, introducida en ES2022. Esto significa que en entornos más antiguos de JavaScript, es posible que esta funcionalidad no esté disponible.
- Simulación de Privacidad: Antes de la introducción de las propiedades privadas, el encapsulamiento se lograba utilizando cierres (closures) y convenciones de nomenclatura (por ejemplo, prefijando los nombres de las propiedades privadas con un guion bajo _).

Ejemplo de Simulación de Propiedades Privadas con Cierres

```
function crearCuentaBancaria(titular, saldoInicial) {
    let saldo = saldoInicial; // Variable privada mediante cierre

    return {
        depositar: function(cantidad) {
            saldo += cantidad;
            console.log(`Se depositaron ${cantidad} dólares. Nuevo saldo: ${saldo}`);
        },
        retirar: function(cantidad) {
            if (cantidad > saldo) {
                console.log('Fondos insuficientes.');
            } else {
                saldo -= cantidad;
                console.log(`Se retiraron ${cantidad} dólares. Nuevo saldo: ${saldo}`);
            }
        },
        consultarSaldo: function() {
            return `El saldo actual es: ${saldo} dólares.`;
        }
    };
}

// Crear una cuenta bancaria utilizando la función
const miCuenta = crearCuentaBancaria('Ana', 1000);
miCuenta.depositar(500); // Se depositaron 500 dólares. Nuevo saldo: 1500
miCuenta.retirar(200);   // Se retiraron 200 dólares. Nuevo saldo: 1300
console.log(miCuenta.consultarSaldo()); // El saldo actual es: 1300 dólares.
```

En este ejemplo, saldo es una variable privada que no puede ser accedida directamente desde fuera del objeto retornado por crearCuentaBancaria.

1.6 Herencia

1.6.1 Introducción a la Herencia

La herencia es un principio fundamental en la programación orientada a objetos que permite crear nuevas clases basadas en clases existentes. En JavaScript, la herencia se implementa utilizando la palabra clave extends. La clase que hereda de otra se denomina clase hija o subclase, mientras que la clase de la que se hereda se denomina clase padre o superclase.

1.6.2 Herencia con la Palabra Clave extends

Cuando una clase extiende otra, la subclase hereda todas las propiedades y métodos de la superclase. Esto permite reutilizar código y crear jerarquías de clases que reflejan relaciones “es un” (por ejemplo, un Perro es un Animal).

Considera una clase Animal y una subclase Perro que hereda de Animal:

```
// Superclase
```

```
class Animal {
    constructor(nombre) {
        this.nombre = nombre;
    }

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

// Subclase que hereda de Animal
class Perro extends Animal {
    ladrar() {
        console.log(`${this.nombre} ladra: ¡Guau guau!`);
    }
}

// Crear una instancia de Perro
const miPerro = new Perro('Firulais');

// Llamar a métodos heredados y específicos
miPerro.hacerSonido(); // Firulais hace un sonido.
miPerro.ladrar();      // Firulais ladra: ¡Guau guau!
```

1.6.3 Uso del Método super()

En una subclase, el método super() se utiliza para llamar al constructor de la superclase. Esto es necesario cuando la subclase tiene su propio constructor y necesitas inicializar las propiedades heredadas de la superclase.

Modifiquemos el ejemplo anterior para incluir más propiedades:

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

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

class Perro extends Animal {
    constructor(nombre, edad, raza) {
        super(nombre, edad); // Llamada al constructor de la superclase
        this.raza = raza;    // Propiedad específica de Perro
    }

    ladrar() {
        console.log(`${this.nombre}, un ${this.raza}, ladra: ¡Guau guau!`);
    }
}

// Crear una instancia de Perro
const miPerro = new Perro('Firulais', 3, 'Labrador');

// Llamar a métodos
miPerro.hacerSonido(); // Firulais hace un sonido.
```

```
miPerro.ladrar();      // Firulais, un Labrador, ladra: ¡Guau guau!
```

En este ejemplo:

- Perro tiene un constructor que toma tres parámetros (nombre, edad, raza).
- La llamada a super(nombre, edad) inicializa las propiedades nombre y edad heredadas de Animal.
- raza es una propiedad específica de Perro.

1.6.4 Sobrescritura de Métodos

La sobrescritura de métodos ocurre cuando una subclase define un método que ya existe en la superclase. El método de la subclase reemplaza al método de la superclase, pero todavía se puede acceder al método original de la superclase utilizando super.

Supongamos que queremos que Perro tenga un comportamiento específico cuando se llama a hacerSonido():

```
class Animal {
    constructor(nombre) {
        this.nombre = nombre;
    }

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

class Perro extends Animal {
    constructor(nombre, raza) {
        super(nombre);
        this.raza = raza;
    }

    // Sobrescribir el método hacerSonido
    hacerSonido() {
        console.log(`${this.nombre}, un ${this.raza}, ladra: ¡Guau guau!`);
    }

    hacerSonidoPadre() {
      super.hacerSonido();
    }
}

// Crear una instancia de Perro
const miPerro = new Perro('Firulais', 'Labrador');

// Llamar al método sobrescrito
miPerro.hacerSonido(); // Firulais, un Labrador, ladra: ¡Guau guau!
```

- Perro sobrescribe el método hacerSonido de Animal con su propia implementación.
- Cuando se llama a hacerSonido() en una instancia de Perro, se utiliza la versión sobrescrita.

1.6.5 Herencia Múltiple (No Soportada Directamente)

JavaScript no soporta herencia múltiple de forma nativa (una clase no puede extender más de una clase). Sin embargo, se pueden usar técnicas como mixins para combinar funcionalidades de múltiples fuentes.

Ejemplo de Uso de Mixins para Simular Herencia Múltiple

```
// Mixin para agregar la habilidad de correr
const Corredor = Base => class extends Base {
    correr() {
        console.log(`${this.nombre} está corriendo.`);
    }
};

// Mixin para agregar la habilidad de nadar
const Nadador = Base => class extends Base {
    nadar() {
        console.log(`${this.nombre} está nadando.`);
    }
};

// Clase base Animal
class Animal {
    constructor(nombre) {
        this.nombre = nombre;
    }

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

// Clase Perro que combina los mixins
class Perro extends Corredor(Nadador(Animal)) {
    ladrar() {
        console.log(`${this.nombre} ladra: ¡Guau guau!`);
    }
}

// Crear una instancia de Perro
const miPerro = new Perro('Firulais');

// Llamar a los métodos
miPerro.hacerSonido(); // Firulais hace un sonido.
miPerro.ladrar();      // Firulais ladra: ¡Guau guau!`);
miPerro.correr();      // Firulais está corriendo.
miPerro.nadar();       // Firulais está nadando.
```

En este ejemplo:

- Perro hereda de Animal, pero también incorpora funcionalidades de los mixins Corredor y Nadador.
- Esto permite que Perro tenga métodos correr y nadar, simulado una especie de herencia múltiple.

1.6.6 Buenas Prácticas en el Uso de Herencia

```
Utilizar Herencia para Relaciones "es un": La herencia debe usarse cuando una subclase
    realmente es una especialización de la superclase.

Evitar Jerarquías Profundas: Las jerarquías de herencia muy profundas pueden hacer que el có
    digo sea difícil de mantener. Es mejor mantenerlas lo más planas posible.

Prefiere Composición sobre Herencia: En muchos casos, es mejor usar la composición (agregar
    comportamiento a través de agregación de objetos o mixins) en lugar de la herencia, para
    mantener la flexibilidad del código.

Sobrescritura Cuidadosa: Al sobrescribir métodos, asegúrate de no romper el contrato de la
    superclase, a menos que tengas una buena razón para hacerlo.
```

1.7 Polimorfismo

1.7.1 Introducción al Polimorfismo

El polimorfismo es un principio fundamental en la programación orientada a objetos que permite que una función o un método tome diferentes formas. En términos simples, significa que el mismo método puede tener diferentes comportamientos según el objeto que lo implemente. En JavaScript, el polimorfismo se manifiesta principalmente a través de la sobrescritura de métodos y la herencia.

1.7.2 Polimorfismo a través de la Sobrescritura de Métodos

Cuando una subclase sobrescribe un método de la superclase, se implementa el polimorfismo. Esto permite que diferentes clases hereden de una misma clase base y reimplementen un método de manera diferente, dependiendo del tipo de objeto que lo esté usando.

Consideremos una superclase Animal con un método hacerSonido. Varias subclases (Perro, Gato, Vaca) sobrescriben este método para implementar su propio sonido.

```
// Superclase
class Animal {
    constructor(nombre) {
        this.nombre = nombre;
    }

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

// Subclase Perro
class Perro extends Animal {
    hacerSonido() {
        console.log(`${this.nombre} ladra: ¡Guau guau!`);
    }
}
```

```
// Subclase Gato
class Gato extends Animal {
    hacerSonido() {
        console.log(`${this.nombre} maúlla: ¡Miau miau!`);
    }
}

// Subclase Vaca
class Vaca extends Animal {
    hacerSonido() {
        console.log(`${this.nombre} muge: ¡Muuu!`);
    }
}

// Crear instancias de diferentes animales
const perro = new Perro('Firulais');
const gato = new Gato('Misu');
const vaca = new Vaca('Lola');

// Llamar al método hacerSonido en cada instancia
perro.hacerSonido(); // Firulais ladra: ¡Guau guau!
gato.hacerSonido();  // Misu maúlla: ¡Miau miau!
vaca.hacerSonido();  // Lola muge: ¡Muuu!
```

En este ejemplo:

- hacerSonido es un método polimórfico. Dependiendo del tipo de objeto (subclase) que lo esté utilizando, se ejecuta una versión diferente del método.
- A pesar de que perro, gato, y vaca son instancias de diferentes clases, todas pueden responder a hacerSonido de una manera específica.

1.7.3 Polimorfismo con Interfaces Implícitas

Aunque JavaScript no tiene soporte nativo para interfaces como en otros lenguajes (por ejemplo, Java o C#), se puede lograr un comportamiento similar a través de la convención. El polimorfismo con interfaces implícitas ocurre cuando diferentes clases implementan métodos con el mismo nombre y firma, permitiendo que se utilicen de manera intercambiable.

Consideremos un escenario donde diferentes clases implementan un método procesarPago, aunque no hereden de una clase común:

```
class TarjetaCredito {
    procesarPago(cantidad) {
        console.log(`Procesando pago de ${cantidad} dólares con tarjeta de crédito.`);
    }
}

class PayPal {
    procesarPago(cantidad) {
        console.log(`Procesando pago de ${cantidad} dólares con PayPal.`);
    }
}
```

```
class Bitcoin {
    procesarPago(cantidad) {
        console.log(`Procesando pago de ${cantidad} dólares con Bitcoin.`);
    }
}

// Función que procesa un pago independientemente del método de pago
function procesarPago(metodoPago, cantidad) {
    metodoPago.procesarPago(cantidad);
}

// Usar diferentes métodos de pago
const tarjeta = new TarjetaCredito();
const paypal = new PayPal();
const bitcoin = new Bitcoin();

procesarPago(tarjeta, 100); // Procesando pago de 100 dólares con tarjeta de crédito.
procesarPago(paypal, 200);  // Procesando pago de 200 dólares con PayPal.
procesarPago(bitcoin, 300); // Procesando pago de 300 dólares con Bitcoin.
```

En este ejemplo:

- TarjetaCredito, PayPal, y Bitcoin no comparten una clase base, pero todos implementan un método procesarPago con la misma firma.
- La función procesarPago puede trabajar con cualquiera de estos objetos, demostrando el polimorfismo basado en interfaces implícitas.

1.7.4 Ventajas del Polimorfismo

- Flexibilidad: Permite que el código sea más flexible y adaptable a diferentes tipos de objetos, lo que facilita la reutilización del código.
- Mantenibilidad: Hace que el código sea más fácil de mantener y extender, ya que las nuevas clases pueden integrarse sin modificar el código existente.
- Desacoplamiento: Fomenta el desacoplamiento entre diferentes partes del código, lo que mejora la modularidad y reduce la dependencia entre módulos.

1.8 Getters y Setters

1.8.1 Definición y Uso de get y set

Los getters y setters en JavaScript son métodos que permiten controlar cómo se accede y modifica el estado de un objeto. Utilizan las palabras clave get y set y se definen como métodos dentro de una clase. Los getters se utilizan para obtener el valor de una propiedad, mientras que los setters se utilizan para modificar el valor de una propiedad.

1.8.2 Ventajas de los Getters y Setters

- Encapsulamiento: Permiten controlar el acceso y la modificación de las propiedades internas de un objeto, manteniendo la integridad del estado.
- Validación: Los setters pueden incluir lógica de validación para asegurar que los valores establecidos sean correctos.
- Cálculos Automáticos: Los getters pueden realizar cálculos o manipular datos antes de devolver un valor, proporcionando más flexibilidad.

1.8.3 Ejemplo de Getters y Setters

Supongamos que estamos modelando una clase Persona donde queremos controlar cómo se accede y modifica la propiedad edad:

```
class Persona {
    constructor(nombre, edad) {
        this.nombre = nombre;
        this._edad = edad; // Propiedad "privada"
    }

    // Getter para obtener la edad
    get edad() {
        return this._edad;
    }

    // Setter para establecer la edad con validación
    set edad(nuevaEdad) {
        if (nuevaEdad >= 0) {
            this._edad = nuevaEdad;
        } else {
            console.log('La edad no puede ser negativa.');
        }
    }

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

// Crear una instancia de Persona
const juan = new Persona('Juan', 30);

// Usar el getter
console.log(juan.edad); // 30

// Usar el setter para modificar la edad
juan.edad = 35;
console.log(juan.edad); // 35

// Intentar establecer una edad inválida
juan.edad = -5; // La edad no puede ser negativa.
console.log(juan.edad); // 35 (la edad no cambió)
```

En este ejemplo:

- *edad es una propiedad "privada" que almacenamos internamente con un guion bajo* para indicar que no debería ser accedida directamente desde fuera de la clase.
- El getter edad devuelve el valor de _edad.
- El setter edad permite establecer un nuevo valor para _edad, pero incluye una validación para asegurarse de que la edad no sea negativa.

1.8.4 Getters y Setters Computados

Los getters y setters no solo se utilizan para acceder y modificar valores simples. También pueden ser utilizados para propiedades que requieren cálculo o manipulación de datos.

Consideremos una clase Rectangulo que calcule el área de un rectángulo basándose en su ancho y alto, y permita ajustar su tamaño con un setter:

```
class Rectangulo {
    constructor(ancho, alto) {
        this.ancho = ancho;
        this.alto = alto;
    }

    // Getter para el área
    get area() {
        return this.ancho * this.alto;
    }

    // Setter para cambiar el tamaño del rectángulo
    set cambiarTamaño({ nuevoAncho, nuevoAlto }) {
        this.ancho = nuevoAncho;
        this.alto = nuevoAlto;
    }
}

// Crear una instancia de Rectangulo
const miRectangulo = new Rectangulo(10, 5);

// Usar el getter para obtener el área
console.log(miRectangulo.area); // 50

// Usar el setter para cambiar el tamaño
miRectangulo.cambiarTamaño = { nuevoAncho: 20, nuevoAlto: 10 };
console.log(miRectangulo.area); // 200
```

En este ejemplo:

- El getter area calcula el área del rectángulo en función de sus propiedades ancho y alto.
- El setter cambiarTamaño permite modificar ambas dimensiones del rectángulo al mismo tiempo.

1.8.5 Usos Comunes de Getters y Setters

- Validación de Entrada: Los setters son ideales para implementar validaciones de entrada antes de modificar una propiedad.
- Cálculos en Tiempo Real: Los getters pueden ser utilizados para devolver resultados calculados dinámicamente en función de otras propiedades.
- Acceso Controlado: Getters y setters permiten un acceso controlado a las propiedades de un objeto, mejorando el encapsulamiento y la seguridad.

Imaginemos que tenemos una clase CuentaBancaria donde queremos controlar cómo se accede y modifica el saldo:

```
class CuentaBancaria {
    constructor(titular, saldoInicial) {
        this.titular = titular;
        this._saldo = saldoInicial;
    }

    // Getter para obtener el saldo
    get saldo() {
        return this._saldo;
    }

    // Setter para depositar dinero
    set depositar(cantidad) {
        if (cantidad > 0) {
            this._saldo += cantidad;
            console.log(`Depositaste ${cantidad}. Nuevo saldo: ${this._saldo}`);
        } else {
            console.log('El monto a depositar debe ser positivo.');
        }
    }

    // Setter para retirar dinero
    set retirar(cantidad) {
        if (cantidad > 0 && cantidad <= this._saldo) {
            this._saldo -= cantidad;
            console.log(`Retiraste ${cantidad}. Nuevo saldo: ${this._saldo}`);
        } else {
            console.log('Fondos insuficientes o cantidad inválida.');
        }
    }
}

// Crear una instancia de CuentaBancaria
const cuenta = new CuentaBancaria('Ana', 1000);

// Consultar el saldo
console.log(cuenta.saldo); // 1000

// Depositar dinero
cuenta.depositar = 500; // Depositaste 500. Nuevo saldo: 1500

// Retirar dinero
cuenta.retirar = 200; // Retiraste 200. Nuevo saldo: 1300

// Intentar retirar una cantidad inválida
cuenta.retirar = 1500; // Fondos insuficientes o cantidad inválida.
```

En este ejemplo:

- El getter saldo devuelve el saldo actual de la cuenta.
- Los setters depositar y retirar permiten modificar el saldo con validación, asegurando que solo se depositen montos positivos y que no se retiren más fondos de los disponibles.

1.8.6 Consideraciones y Buenas Prácticas

- Consistencia: Mantén la coherencia en la interfaz pública de la clase, utilizando getters y setters cuando sea necesario para controlar el acceso a las propiedades.
- Evitar Lógica Compleja: Aunque los getters y setters pueden realizar cálculos o validaciones, evita introducir lógica demasiado compleja que pueda dificultar el entendimiento y la depuración del código.
- Documentación: Documenta bien el comportamiento de getters y setters, especialmente cuando realizan más que un simple acceso o modificación de propiedades.

1.9 Clases Abstractas y Métodos Abstractos

1.9.1 Concepto de Clases Abstractas

En la programación orientada a objetos, una clase abstracta es una clase que no puede ser instanciada directamente. Su propósito principal es servir como una clase base para otras clases. Las clases abstractas suelen contener uno o más métodos abstractos, que son métodos declarados sin implementación y que deben ser implementados por las subclases.

JavaScript no tiene soporte nativo para clases abstractas como otros lenguajes de programación (por ejemplo, Java o C#). Sin embargo, podemos simular el comportamiento de una clase abstracta utilizando ciertas convenciones y técnicas, como lanzar errores si alguien intenta instanciar la clase abstracta directamente.

1.9.2 Implementación de Clases Abstractas en JavaScript

Para simular una clase abstracta en JavaScript, podemos definir un constructor que lance un error si se intenta crear una instancia de la clase abstracta. Además, los métodos que queramos que sean abstractos pueden lanzar un error si no son sobrescritos en las subclases.

Supongamos que queremos definir una clase abstracta FiguraGeometrica que tenga un método abstracto calcularArea:

```
class FiguraGeometrica {
    constructor() {
        if (this.constructor === FiguraGeometrica) {
            throw new Error("No se puede instanciar una clase abstracta");
        }
    }

    // Método abstracto
    calcularArea() {
        throw new Error("El método 'calcularArea' debe ser implementado");
    }
}

// Subclase que implementa el método calcularArea
class Rectangulo extends FiguraGeometrica {
    constructor(ancho, alto) {
        super();
        this.ancho = ancho;
        this.alto = alto;
    }

    calcularArea() {
        return this.ancho * this.alto;
    }
}

// Subclase que implementa el método calcularArea
class Circulo extends FiguraGeometrica {
    constructor(radio) {
        super();
        this.radio = radio;
    }

    calcularArea() {
        return Math.PI * Math.pow(this.radio, 2);
    }
}

// Intentar instanciar la clase abstracta (genera un error)
try {
    const figura = new FiguraGeometrica();
} catch (error) {
    console.error(error.message); // No se puede instanciar una clase abstracta
}

// Crear instancias de las subclases y calcular el área
const rectangulo = new Rectangulo(10, 5);
console.log(`Área del rectángulo: ${rectangulo.calcularArea()}`); // Área del rectángulo: 50

const circulo = new Circulo(7);
console.log(`Área del círculo: ${circulo.calcularArea()}`); // Área del círculo:
    153.93804002589985
```

En este ejemplo:

- FiguraGeometrica actúa como una clase abstracta, que no puede ser instanciada directamente. El constructor lanza un error si se intenta crear una instancia de ella.
- El método calcularArea en FiguraGeometrica es abstracto y lanza un error si no es sobrescrito por las subclases.
- Rectangulo y Circulo son subclases que implementan el método calcularArea, proporcionando

una implementación específica.

1.9.3 Uso Práctico de Clases Abstractas

Las clases abstractas son útiles en escenarios donde deseas definir un comportamiento común para un grupo de clases, pero dejar detalles específicos a las subclases. Esto es especialmente útil en sistemas que requieren una estructura jerárquica bien definida.

Imaginemos que estamos desarrollando un sistema de facturación que necesita manejar diferentes tipos de documentos (facturas, recibos, etc.). Podemos definir una clase abstracta Documento que obligue a las subclases a implementar un método generarPDF:

```
class Documento {
    constructor(titulo) {
        if (this.constructor === Documento) {
            throw new Error("No se puede instanciar una clase abstracta");
        }
        this.titulo = titulo;
    }

    // Método abstracto
    generarPDF() {
        throw new Error("El método 'generarPDF' debe ser implementado");
    }
}

class Factura extends Documento {
    constructor(titulo, total) {
        super(titulo);
        this.total = total;
    }

    generarPDF() {
        console.log(`Generando PDF de la factura: ${this.titulo} por ${this.total} dólares.`)
            ;
    }
}

class Recibo extends Documento {
    constructor(titulo, monto) {
        super(titulo);
        this.monto = monto;
    }

    generarPDF() {
        console.log(`Generando PDF del recibo: ${this.titulo} por ${this.monto} dólares.`);
    }
}

// Crear instancias de Factura y Recibo
const factura = new Factura('Factura #123', 500);
factura.generarPDF(); // Generando PDF de la factura: Factura #123 por 500 dólares.

const recibo = new Recibo('Recibo #456', 200);
recibo.generarPDF(); // Generando PDF del recibo: Recibo #456 por 200 dólares.
```

En este ejemplo:

- Documento es una clase abstracta que define el método generarPDF como abstracto.
- Factura y Recibo son subclases que implementan generarPDF para generar un PDF con la información específica de cada tipo de documento.

1.9.4 Resumen y Buenas Prácticas

- Uso de Clases Abstractas: Utiliza clases abstractas cuando necesitas definir una estructura base común para un conjunto de clases relacionadas, pero no quieres que la clase base sea instanciada directamente.
- Implementación Obligatoria: Los métodos abstractos deben ser implementados por todas las subclases. Si una subclase no implementa un método abstracto, se debe lanzar un error para evitar un comportamiento inesperado.
- Documentación: Es importante documentar claramente las clases y métodos abstractos, indicando que deben ser extendidos e implementados por las subclases.
- Flexibilidad: Si bien las clases abstractas proporcionan una estructura sólida, asegúrate de que las subclases tengan suficiente flexibilidad para personalizar el comportamiento sin romper el contrato establecido por la clase abstracta.

1.10 Mixins

1.10.1 Descripción

Un mixin es un patrón de diseño en la programación orientada a objetos que permite agregar funcionalidades a una clase sin utilizar herencia múltiple. En lugar de heredar de una sola clase, una clase puede “mezclar” comportamientos de diferentes mixins, combinando varias funcionalidades independientes.

En JavaScript, los mixins se implementan como funciones que toman una clase base y retornan una nueva clase con funcionalidades adicionales. Este patrón es particularmente útil en escenarios donde se desea compartir comportamiento entre clases que no comparten una jerarquía común.

- Reutilización de Código: Permiten compartir funcionalidades entre clases sin la necesidad de herencia múltiple.
- Modularidad: Facilitan la composición de clases complejas a partir de componentes más pequeños y reutilizables.
- Flexibilidad: Permiten agregar características a una clase de manera selectiva, según sea necesario, sin afectar su jerarquía de herencia.

Supongamos que queremos crear varias clases que puedan tanto correr como nadar. En lugar de duplicar el código en cada clase, utilizamos mixins para agregar estas habilidades.

```
// Mixin para agregar la habilidad de correr
const Corredor = Base => class extends Base {
    correr() {
        console.log(`${this.nombre} está corriendo.`);
    }
};

// Mixin para agregar la habilidad de nadar
const Nadador = Base => class extends Base {
    nadar() {
        console.log(`${this.nombre} está nadando.`);
    }
};

// Clase base Animal
class Animal {
    constructor(nombre) {
        this.nombre = nombre;
    }

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

// Clase Perro que usa los mixins Corredor y Nadador
class Perro extends Corredor(Nadador(Animal)) {
    ladrar() {
        console.log(`${this.nombre} ladra: ¡Guau guau!`);
    }
}

// Crear una instancia de Perro
const miPerro = new Perro('Firulais');

// Llamar a los métodos del perro
miPerro.hacerSonido(); // Firulais hace un sonido.
miPerro.ladrar();      // Firulais ladra: ¡Guau guau!
miPerro.correr();      // Firulais está corriendo.
miPerro.nadar();       // Firulais está nadando.
```

En este ejemplo:

- Corredor y Nadador son mixins que agregan habilidades específicas (correr y nadar) a cualquier clase que los utilice.
- Perro hereda de Animal y utiliza ambos mixins para obtener las funcionalidades adicionales.
- miPerro, una instancia de Perro, puede realizar todas las acciones definidas en Animal, Corredor, y Nadador.

1.10.2 Limitaciones y Consideraciones

- Conflictos de Nombre: Si dos mixins intentan agregar métodos con el mismo nombre, puede haber conflictos. Esto debe gestionarse cuidadosamente para evitar problemas.

- Complejidad: Usar demasiados mixins puede llevar a un código difícil de mantener y entender, ya que las clases pueden terminar con demasiadas responsabilidades mezcladas.
- Encapsulamiento: Al agregar métodos a través de mixins, se debe tener cuidado de no romper el encapsulamiento de la clase, especialmente si los mixins acceden directamente a propiedades internas.

Ejemplo de Manejo de Conflictos de Nombre

Supongamos que tenemos dos mixins, Volador y SuperSalto, que ambos agregan un método llamado moverse:

```
// Mixin para volar con un método moverse
const Volador = Base => class extends Base {
    moverse() {
        console.log(`${this.nombre} está volando.`);
    }
};

// Mixin para saltar con un método moverse
const SuperSalto = Base => class extends Base {
    moverse() {
        console.log(`${this.nombre} está saltando alto.`);
    }
};

// Clase Superheroe que combina los mixins
class Superheroe extends SuperSalto(Volador(Animal)) {
    moverse() {
        console.log(`${this.nombre} está usando ambos poderes.`);
    }
}

// Crear una instancia de Superheroe
const miHeroe = new Superheroe('Batman');

// Llamar al método moverse
miHeroe.moverse(); // Batman está usando ambos poderes.
```

En este ejemplo: - Volador y SuperSalto tienen un método moverse, lo que podría causar un conflicto. - En Superheroe, sobrescribimos moverse para evitar el conflicto y definir un comportamiento específico.

1.11 Tipos de Datos en Clases

1.11.1 Tipos de Datos Primitivos y No Primitivos

En JavaScript, los tipos de datos pueden clasificarse en dos categorías principales: primitivos y no primitivos.

- Primitivos: Son los tipos de datos más básicos y se incluyen string, number, boolean, null, undefined, symbol y bigint. Estos tipos se manejan por valor, lo que significa que cuando se asigna

una variable a otra, se copia el valor.

- No Primitivos: Incluyen object, array, function, y otros tipos derivados de object. Estos se manejan por referencia, lo que significa que cuando se asigna una variable a otra, ambas variables apuntan al mismo objeto en memoria.

1.12 Contexto (this) en Clases

1.12.1 Entendiendo el Contexto this

En JavaScript, el contexto de this se refiere al objeto que está ejecutando el código actual. En el contexto de las clases, this generalmente se refiere a la instancia actual de la clase. Sin embargo, el valor de this puede cambiar dependiendo de cómo se llama a una función o método, lo que puede causar confusión si no se maneja adecuadamente.

1.12.2 this en el Contexto de una Clase

Cuando utilizas this dentro de una clase, estás accediendo a las propiedades y métodos de la instancia de esa clase. Por ejemplo, en el constructor de una clase, this se refiere a la nueva instancia que se está creando. Ejemplo de this en Clases

```
class Persona {
    constructor(nombre, edad) {
        this.nombre = nombre; // `this` se refiere a la instancia de Persona
        this.edad = edad;
    }

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

// Crear una instancia de Persona
const juan = new Persona('Juan', 30);
juan.saludar(); // Hola, mi nombre es Juan y tengo 30 años.
```

En este ejemplo:

- Dentro del constructor y del método saludar, this se refiere a la instancia específica de Persona (juan en este caso).
- Esto permite acceder a las propiedades nombre y edad definidas en la instancia.

1.12.3 Cambiar el Contexto con bind, call, y apply

JavaScript proporciona tres métodos (bind, call, y apply) que permiten cambiar explícitamente el valor de this. Esto es útil cuando necesitas que una función se ejecute en un contexto diferente.

- bind: Crea una nueva función con this vinculado a un valor específico.
- call: Llama a una función con un valor específico de this y argumentos separados por comas.
- apply: Llama a una función con un valor específico de this y argumentos como un array.

Ejemplo de Uso de bind, call, y apply

```
class Saludador {
    constructor(nombre) {
        this.nombre = nombre;
    }

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

const saludador = new Saludador('Carlos');

// Usando `bind` para cambiar el contexto de `this`
const saludarComoPedro = saludador.saludar.bind({ nombre: 'Pedro' });
saludarComoPedro(); // Hola, soy Pedro

// Usando `call` para ejecutar la función con un nuevo contexto
saludador.saludar.call({ nombre: 'Luis' }); // Hola, soy Luis

// Usando `apply` para ejecutar la función con un nuevo contexto
saludador.saludar.apply({ nombre: 'Ana' }); // Hola, soy Ana
```

En este ejemplo:

- bind: Crea una nueva función saludarComoPedro donde this está vinculado a { nombre: 'Pedro' }.
- call: Llama a saludar con this establecido en { nombre: 'Luis' }.
- apply: Llama a saludar con this establecido en { nombre: 'Ana' }.

1.12.4 this en Funciones Flecha

Las funciones flecha en JavaScript tienen un comportamiento especial respecto a this: no tienen su propio contexto this, sino que heredan el this del contexto en el que se definen. Esto es diferente a las funciones normales y puede ser útil para evitar problemas de contexto.

```
class Contador {
    constructor() {
        this.contador = 0;
    }
```

```
    iniciar() {
        setInterval(() => {
            this.contador++;
            console.log(this.contador);
        }, 1000);
    }
}

const miContador = new Contador();
miContador.iniciar(); // Incrementa e imprime el contador cada segundo
```

En este ejemplo:

- La función flecha dentro de setInterval no tiene su propio this, por lo que hereda el this de iniciar, que es la instancia de Contador.
- Esto permite que el this.contador++ funcione correctamente, incrementando la propiedad contador de la instancia.

1.12.5 Problemas Comunes con this y Cómo Evitarlos

El manejo incorrecto de this puede llevar a errores, especialmente en callbacks y en funciones que son pasadas como argumentos.

Problema: Pérdida del Contexto this en Callbacks

```
class Boton {
    constructor(texto) {
        this.texto = texto;
    }

    click() {
        console.log(`Botón ${this.texto} clickeado`);
    }

    registrarClick() {
        document.getElementById('miBoton').addEventListener('click', this.click);
    }
}

const boton = new Boton('Enviar');
boton.registrarClick(); // Puede fallar porque `this` no está correctamente enlazado
```

En este ejemplo, el método click puede fallar porque this no está correctamente enlazado al objeto Boton.

Solución: Usar bind o una Función Flecha

```
class Boton {
    constructor(texto) {
        this.texto = texto;
    }

    click() {
        console.log(`Botón ${this.texto} clickeado`);
```

```
    }

    registrarClick() {
        document.getElementById('miBoton').addEventListener('click', this.click.bind(this));
    }
}

const boton = new Boton('Enviar');
boton.registrarClick(); // Ahora `this` está correctamente enlazado
```

O alternativamente, usando una función flecha:

```
class Boton {
    constructor(texto) {
        this.texto = texto;
    }

    registrarClick() {
        document.getElementById('miBoton').addEventListener('click', () => {
            console.log(`Botón ${this.texto} clickeado`);
        });
    }
}

const boton = new Boton('Enviar');
boton.registrarClick(); // `this` se hereda correctamente de la instancia
```

1.12.6 Buenas Prácticas con this en Clases

- Usar Funciones Flecha en Callbacks: Las funciones flecha son útiles para evitar problemas con el contexto de this, especialmente en callbacks y funciones asíncronas.
- Usar bind para Métodos: Cuando pases métodos de clase como callbacks, usa bind para asegurar que this se refiera al objeto correcto.
- Evitar this en Funciones Anidadas: Si necesitas acceder a this en funciones anidadas, usa una función flecha o guarda el contexto de this en una variable (const self = this; o similar).

1.13 Clases y Prototipos

1.13.1 Relación entre Clases y Prototipos en JavaScript

En JavaScript, las clases son una forma sintáctica de trabajar con la herencia basada en prototipos. Bajo el capó, cuando defines una clase, en realidad estás definiendo una función constructora y un prototipo asociado. Cada vez que creas una instancia de una clase, esa instancia hereda propiedades y métodos desde el prototipo de la clase.

1.13.2 Prototipos: Conceptos Básicos

Un prototipo es un objeto del que otros objetos heredan propiedades y métodos. En JavaScript, cada función (incluyendo las clases) tiene una propiedad prototype, que es un objeto que contiene las propiedades y métodos que deben ser compartidos entre todas las instancias creadas por esa función.

Cuando accedes a una propiedad o método de un objeto, JavaScript primero busca en el propio objeto. Si no encuentra lo que busca, sube en la cadena de prototipos hasta que encuentra la propiedad o método, o hasta que alcanza el final de la cadena (null).

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

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

const perro = new Animal('Firulais');
perro.hacerSonido(); // Firulais hace un sonido.
```

En este ejemplo:

- Animal es una función constructora, y Animal.prototype es el prototipo del que las instancias de Animal heredan.
- hacerSonido es un método definido en Animal.prototype, por lo que todas las instancias de Animal pueden acceder a él.

1.13.3 Añadir Métodos al Prototipo de una Clase

Aunque los métodos de una clase se definen generalmente dentro de la propia clase, también es posible añadir métodos al prototipo de una clase fuera de la declaración de la clase.

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

// Añadir un método al prototipo después de la declaración de la clase
Persona.prototype.saludar = function() {
    console.log(`Hola, me llamo ${this.nombre}.`);
};

const juan = new Persona('Juan');
juan.saludar(); // Hola, me llamo Juan.
```

En este ejemplo: - saludar se añade a Persona.prototype después de que la clase Persona ha sido definida. - Todas las instancias de Persona pueden acceder al método saludar.

1.13.4 Modificar el Prototipo de una Clase

Puedes modificar el prototipo de una clase para añadir o sobrescribir métodos existentes. Sin embargo, esto debe hacerse con cuidado, ya que modificar el prototipo de manera incorrecta puede tener efectos colaterales en todas las instancias de esa clase.

```
class Gato {
    constructor(nombre) {
        this.nombre = nombre;
    }

    maullar() {
        console.log(`${this.nombre} dice: Miau`);
    }
}

const gato = new Gato('Misu');
gato.maullar(); // Misu dice: Miau

// Sobrescribir el método maullar en el prototipo
Gato.prototype.maullar = function() {
    console.log(`${this.nombre} dice: Miau Miau`);
};

gato.maullar(); // Misu dice: Miau Miau
```

En este ejemplo:

- maullar se define inicialmente dentro de la clase Gato.
- Luego, se sobrescribe maullar directamente en Gato.prototype, cambiando su comportamiento para todas las instancias de Gato.

1.13.5 Buenas Prácticas al Trabajar con Clases y Prototipos

- Mantén los Métodos en el Prototipo: Definir métodos en el prototipo (a través de la sintaxis de clases) es más eficiente que definirlos dentro del constructor, ya que los métodos definidos en el prototipo son compartidos entre todas las instancias.
- Evita Modificar el Prototipo de Objetos Nativos: Modificar el prototipo de objetos nativos (como Array o Object) es generalmente desaconsejado, ya que puede causar conflictos con otros scripts o bibliotecas.
- Usa Clases para Legibilidad: Aunque puedes manipular prototipos directamente, usar la sintaxis de clases mejora la legibilidad y organización del código, especialmente en proyectos más grandes.
- Comprende la Cadena de Prototipos: Es importante entender cómo JavaScript busca propiedades y métodos en la cadena de prototipos para evitar errores sutiles y problemas de rendimiento.

1.14 Ejemplos Prácticos

En esta sección, se presentarán ejemplos prácticos para aplicar los conceptos aprendidos sobre clases en JavaScript. Estos ejemplos demostrarán cómo crear clases básicas, utilizar herencia, implementar polimorfismo, y trabajar con mixins y métodos estáticos.

1.14.1 Crear una Clase Básica

Comencemos con un ejemplo simple de una clase que representa un libro en una biblioteca.

```
class Libro {
    constructor(titulo, autor, añoPublicacion) {
        this.titulo = titulo;
        this.autor = autor;
        this.añoPublicacion = añoPublicacion;
    }

    describir() {
        console.log(`${this.titulo} fue escrito por ${this.autor} en ${this.añoPublicacion
            }.`);
    }
}

// Crear una instancia de Libro
const miLibro = new Libro('Cien Años de Soledad', 'Gabriel García Márquez', 1967);
miLibro.describir(); // Cien Años de Soledad fue escrito por Gabriel García Márquez en 1967.
```

En este ejemplo:

- La clase Libro tiene un constructor que inicializa las propiedades titulo, autor, y añoPublicacion.
- El método describir proporciona una forma de describir el libro.

1.14.2 Ejemplo de Herencia y Polimorfismo

Ahora, extendamos la clase Libro para crear una subclase Ebook que represente un libro electrónico.

```
class Libro {
    constructor(titulo, autor, añoPublicacion) {
        this.titulo = titulo;
        this.autor = autor;
        this.añoPublicacion = añoPublicacion;
    }

    describir() {
        console.log(`${this.titulo} fue escrito por ${this.autor} en ${this.añoPublicacion
            }.`);
    }
}

class Ebook extends Libro {
    constructor(titulo, autor, añoPublicacion, tamañoArchivo) {
```

```
        super(titulo, autor, añoPublicacion);
        this.tamañoArchivo = tamañoArchivo; // Propiedad específica de Ebook
    }

    describir() {
        console.log(`${this.titulo} fue escrito por ${this.autor} en ${this.añoPublicacion},
            y tiene un tamaño de archivo de ${this.tamañoArchivo} MB.`);
    }

    descargar() {
        console.log(`Descargando ${this.titulo}...`);
    }
}

// Crear una instancia de Ebook
const miEbook = new Ebook('1984', 'George Orwell', 1949, 2.5);
miEbook.describir(); // 1984 fue escrito por George Orwell en 1949, y tiene un tamaño de
    archivo de 2.5 MB.
miEbook.descargar(); // Descargando 1984...
```

En este ejemplo:

- Ebook extiende Libro, agregando una propiedad tamañoArchivo y sobrescribiendo el método describir para incluir información específica sobre el tamaño del archivo.
- Ebook también introduce un nuevo método descargar, que no está presente en Libro.

1.14.3 Aplicación de Mixins

Supongamos que queremos que ciertos objetos tengan la capacidad de ser auditados (registrar quién los creó y cuándo). Podemos crear un mixin Auditado y aplicarlo a diferentes clases.

```
const Auditado = Base => class extends Base {
    registrarAuditoria(usuario) {
        this.creadoPor = usuario;
        this.fechaCreacion = new Date();
    }

    mostrarAuditoria() {
        console.log(`Creado por: ${this.creadoPor} el ${this.fechaCreacion}`);
    }
};

class Producto {
    constructor(nombre, precio) {
        this.nombre = nombre;
        this.precio = precio;
    }

    describir() {
        console.log(`Producto: ${this.nombre}, Precio: ${this.precio} dólares.`);
    }
}

// Aplicar el mixin Auditado a la clase Producto
class ProductoAuditado extends Auditado(Producto) {}

// Crear una instancia de ProductoAuditado
```

```
const miProducto = new ProductoAuditado('Laptop', 1500);
miProducto.registrarAuditoria('Juan Pérez');
miProducto.describir(); // Producto: Laptop, Precio: 1500 dólares.
miProducto.mostrarAuditoria(); // Creado por: Juan Pérez el <fecha_actual>
```

En este ejemplo:

- Auditado es un mixin que añade métodos para registrar y mostrar información de auditoría.
- ProductoAuditado extiende Producto y mezcla la funcionalidad de Auditado, permitiendo que las instancias de ProductoAuditado sean auditadas.

1.14.4 Uso de Métodos Estáticos

Los métodos estáticos son útiles para definir funciones que no necesitan acceso a this y que están relacionadas con la clase en sí, más que con las instancias.

```
class Matematica {
    static sumar(a, b) {
        return a + b;
    }

    static restar(a, b) {
        return a - b;
    }

    static multiplicar(a, b) {
        return a * b;
    }

    static dividir(a, b) {
        if (b === 0) {
            throw new Error('No se puede dividir por cero');
        }
        return a / b;
    }
}

console.log(Matematica.sumar(10, 5)); // 15
console.log(Matematica.restar(10, 5)); // 5
console.log(Matematica.multiplicar(10, 5)); // 50
console.log(Matematica.dividir(10, 5)); // 2
```

En este ejemplo:

- Matematica es una clase que define varios métodos estáticos (sumar, restar, multiplicar, dividir).
- Estos métodos no necesitan acceso a this, por lo que son estáticos y se pueden llamar directamente desde la clase sin crear una instancia.

1.14.5 Integración de Diferentes Conceptos

Veamos un ejemplo más complejo que integra varios conceptos como herencia, mixins y métodos estáticos. Supongamos que estamos creando un sistema de gestión de usuarios y productos para un

e-commerce.

```
const Auditado = Base => class extends Base {
    registrarAuditoria(usuario) {
        this.creadoPor = usuario;
        this.fechaCreacion = new Date();
    }

    mostrarAuditoria() {
        console.log(`Creado por: ${this.creadoPor} el ${this.fechaCreacion}`);
    }
};

class Usuario {
    constructor(nombre, email) {
        this.nombre = nombre;
        this.email = email;
    }

    describir() {
        console.log(`Usuario: ${this.nombre}, Email: ${this.email}`);
    }
}

class Producto {
    constructor(nombre, precio) {
        this.nombre = nombre;
        this.precio = precio;
    }

    describir() {
        console.log(`Producto: ${this.nombre}, Precio: ${this.precio} dólares`);
    }
}

class ProductoAuditado extends Auditado(Producto) {}

class GestorEcommerce {
    static listarProductos(productos) {
        productos.forEach(producto => producto.describir());
    }

    static listarUsuarios(usuarios) {
        usuarios.forEach(usuario => usuario.describir());
    }
}

// Crear instancias de Usuario y ProductoAuditado
const usuario1 = new Usuario('Ana', 'ana@example.com');
const usuario2 = new Usuario('Carlos', 'carlos@example.com');

const producto1 = new ProductoAuditado('Laptop', 1500);
producto1.registrarAuditoria('Ana');

const producto2 = new ProductoAuditado('Smartphone', 800);
producto2.registrarAuditoria('Carlos');

// Listar usuarios y productos
GestorEcommerce.listarUsuarios([usuario1, usuario2]);
GestorEcommerce.listarProductos([producto1, producto2]);

// Mostrar auditoría
producto1.mostrarAuditoria(); // Creado por: Ana el <fecha_actual>
producto2.mostrarAuditoria(); // Creado por: Carlos el <fecha_actual>
```

En este ejemplo: - Usuario y Producto son clases base que representan entidades en el sistema. - ProductoAuditado extiende Producto y mezcla la funcionalidad del mixin Auditado. - GestorEcommerce es una clase que utiliza métodos estáticos para listar usuarios y productos. - Este ejemplo demuestra cómo integrar diferentes conceptos para construir un sistema más complejo y funcional.

2 Excepciones

Las excepciones son una parte fundamental del manejo de errores, permitiendo a los desarrolladores detectar y manejar situaciones anómalas o inesperadas durante la ejecución de un programa. Las excepciones en JavaScript se manejan principalmente mediante las declaraciones try, catch, throw, y finally.

Una excepción es un evento que ocurre durante la ejecución de un programa y que interrumpe el flujo normal de las instrucciones. En JavaScript, cuando ocurre una excepción, se crea un objeto de tipo Error (o una subclase de Error), y el control del programa se transfiere a un manejador de excepciones (si existe).

2.1 Manejo de Excepciones con try y catch

La estructura básica para manejar excepciones en JavaScript es:

```
try {
  // Bloque de código donde puede ocurrir una excepción
} catch (error) {
  // Código que se ejecuta si se lanza una excepción en el bloque try
}
```

- try: El bloque try contiene el código que podría lanzar una excepción.
- catch: Si ocurre una excepción en el bloque try, la ejecución se transfiere al bloque catch. El bloque catch recibe como argumento el objeto de la excepción lanzada.

```
function performRiskyOperation(data) {
  if (!data) {
    throw new Error("No se proporcionaron datos"); // Lanzar una excepción si no se recibe la
        información necesaria
  }

  if (typeof data !== 'string') {
    throw new TypeError("Se esperaba un string como entrada"); // Lanzar un error específico
       de tipo
  }

  // Simulación de una operación riesgosa
  if (data.length < 5) {
    throw new RangeError("El dato es demasiado corto para procesar"); // Otro tipo de error
       específico
  }
```

```
  // Si todo es correcto, retorna un resultado
  return `Procesado: ${data.toUpperCase()}`;
}

try {
  let result = performRiskyOperation("abc"); // Se proporciona un string corto, lo que
      desencadena un RangeError
  console.log(result);
} catch (error) {
  if (error instanceof TypeError) {
    console.error("Error de tipo:", error.message);
  } else if (error instanceof RangeError) {
    console.error("Error de rango:", error.message);
  } else {
    console.error("Error general:", error.message);
  }
} finally {
  console.log("La operación ha finalizado."); // Este bloque se ejecuta siempre
}
```

2.2 Lanzar Excepciones con throw

El uso de throw permite lanzar una excepción manualmente. Puedes lanzar cualquier valor como una excepción, aunque generalmente se lanza un objeto de tipo Error o una de sus subclases.

```
function divide(a, b) {
  if (b === 0) {
    throw new Error('División por cero');
  }
  return a / b;
}

try {
  let result = divide(4, 0);
} catch (error) {
  console.error('Error:', error.message);
}
```

En este ejemplo, si b es 0, se lanza una excepción personalizada con el mensaje “División por cero”.

2.3 Bloque finally

El bloque finally es opcional y se utiliza para ejecutar código que debe correr independientemente de si ocurre una excepción o no. El bloque finally se ejecuta después de que se hayan ejecutado los bloques try y catch.

```
try {
  let result = riskyOperation();
  console.log(result);
} catch (error) {
  console.error('Error:', error.message);
} finally {
  console.log('Este mensaje se mostrará siempre.');
```

```
}
```

2.4 Subclases de Error

En JavaScript, es común crear subclases de Error para representar diferentes tipos de errores específicos. Por ejemplo, puedes crear un error específico para fallos en la validación:

```
class ValidationError extends Error {
  constructor(message) {
    super(message);
    this.name = "ValidationError";
  }
}

function validateUserInput(input) {
  if (input.length < 5) {
    throw new ValidationError("El input es demasiado corto");
  }
}

try {
  validateUserInput("abc");
} catch (error) {
  if (error instanceof ValidationError) {
    console.error('Error de validación:', error.message);
  } else {
    console.error('Otro tipo de error:', error.message);
  }
}
```

2.5 Manejo de Excepciones Asíncronas

Con la popularización de async y await en JavaScript, manejar excepciones en código asíncrono es crucial. Las excepciones lanzadas en funciones asíncronas pueden ser capturadas utilizando try-catch dentro de una función asíncrona.

```
async function fetchData() {
  try {
    let response = await fetch('https://api.example.com/data');
    if (!response.ok) {
      throw new Error('Error en la red');
    }
    let data = await response.json();
    return data;
  } catch (error) {
    console.error('Error al obtener los datos:', error.message);
  }
}

fetchData();
```

2.6 Propagación de Excepciones

Las excepciones en JavaScript se propagan hacia arriba en la cadena de llamadas hasta que son capturadas por un bloque catch. Si no hay ningún bloque catch en la cadena, la excepción no capturada provocará que el programa termine su ejecución (o en el caso de un entorno de navegador, mostrará un mensaje de error en la consola).

2.7 Buenas Prácticas en el Manejo de Excepciones

- Usar try-catch solo cuando sea necesario: No es recomendable abusar del manejo de excepciones. Solo usa try-catch cuando el código dentro del bloque try pueda fallar de manera predecible.
- Lanzar errores con información útil: Cuando lances excepciones, asegúrate de proporcionar mensajes claros y contextuales para que el error sea fácil de identificar y resolver.
- Manejar excepciones específicas: Captura y maneja solo las excepciones que esperas. Evita capturar todas las excepciones si solo te interesa manejar un tipo específico de error.

3 Módulos

En JavaScript, antes de la introducción de los módulos en ES6, todas las variables, funciones y objetos definidos en un archivo .js se incorporaban al ámbito global. Esto presentaba un problema significativo, ya que podía dar lugar a conflictos de nombres al utilizar múltiples librerías en una misma aplicación. Para mitigar este riesgo, se utilizaban closures, una técnica que envolvía las declaraciones dentro de una función para aislarlas del resto del código.

Con la llegada de ES6, se introdujeron los módulos, una solución más robusta y organizada para manejar el código. Los módulos permiten encapsular las declaraciones dentro de su propio ámbito, evitando conflictos y mejorando la mantenibilidad del código. Al incluir un módulo en una página web, se utiliza la etiqueta `<script type="module"src="modulo.js"></script>`. De esta forma, todas las declaraciones de un módulo son visibles únicamente dentro de ese módulo, y si se desea que alguna de ellas sea accesible desde otros archivos, es necesario exportarla explícitamente e importarla donde se necesite.

3.1 Exportación en ES6+

En ES6, un módulo puede exportar variables, funciones, clases, etc. Hay dos formas principales de exportar elementos: exportación nombrada y exportación por defecto.

3.1.1 Exportación nombrada

Permite exportar múltiples elementos desde un módulo. Cada elemento exportado debe ser importado usando el mismo nombre en el módulo que lo importe.

```
// math.js
export const PI = 3.14159;

export function sum(a, b) {
  return a + b;
}

export class Circle {
  constructor(radius) {
    this.radius = radius;
  }

  area() {
    return PI * this.radius * this.radius;
  }
}
```

Para importar estos elementos en otro módulo:

```
// app.js
import { PI, sum, Circle } from './math.js';

console.log(PI); // 3.14159
console.log(sum(2, 3)); // 5

const myCircle = new Circle(5);
console.log(myCircle.area()); // 78.53975
```

3.1.2 Exportación por defecto

Un módulo puede tener una exportación por defecto, que es útil cuando se quiere exportar un único valor o función desde un módulo. La exportación por defecto puede ser importada usando cualquier nombre.

```
// logger.js
export default function log(message) {
  console.log(message);
}
```

Para importar la exportación por defecto:

```
// app.js
import log from './logger.js';

log('Hello, world!'); // Hello, world!
```

3.2 Importación en ES6+

Las importaciones en ES6+ pueden ser muy flexibles. Aparte de la importación directa como se mostró anteriormente, existen otras formas: Importación completa de un módulo

- Se pueden importar todos los exportados de un módulo bajo un solo objeto usando * as:

```
// app.js
import * as MathUtils from './math.js';

console.log(MathUtils.PI); // 3.14159
console.log(MathUtils.sum(2, 3)); // 5
```

- Importación combinada

Puedes combinar la importación de elementos específicos y una importación por defecto:

```
// app.js
import log, { PI, sum } from './math.js';

log(PI); // 3.14159
log(sum(2, 3)); // 5
```

3.3 Beneficios de los módulos ES6+

- Encapsulación: Los módulos permiten encapsular el código, evitando conflictos de nombres y creando un espacio de nombres específico para cada módulo.
- Reutilización: Los módulos pueden ser reutilizados en diferentes partes de la aplicación o incluso en diferentes proyectos.
- Optimización: Muchos motores de JavaScript modernos y herramientas como Webpack permiten la tree-shaking o “agitación de árbol”, eliminando el código no utilizado durante la fase de construcción, lo que resulta en archivos más pequeños y eficientes.
- Cargar módulos de manera asíncrona: En combinación con el sistema de módulos dinámicos (usando import()), se pueden cargar módulos de forma asíncrona, lo que puede mejorar el rendimiento en aplicaciones grandes.
- Mantenimiento: Al dividir el código en módulos, es más fácil mantener y actualizar la base de código. Los cambios en un módulo pueden realizarse sin afectar otras partes de la aplicación.

3.4 Módulos en Navegadores vs. Node.js

- Navegadores: Los módulos ES6 son soportados directamente por la mayoría de los navegadores modernos. Cuando se usa en navegadores, los módulos deben ser cargados desde un servidor debido a las restricciones de CORS (Cross-Origin Resource Sharing).

```
<script type="module" src="app.js"></script>
```

- Node.js: Node.js usa un sistema de módulos basado en CommonJS (require y module.exports). Sin embargo, desde la versión 12, Node.js también soporta módulos ES6 nativamente. Para usar módulos ES6 en Node.js, el archivo debe tener la extensión .mjs o el campo “type”: “module” debe estar presente en el package.json.

3.5 Módulos Dinámicos

ES6+ también introduce la capacidad de importar módulos de manera dinámica usando la función import():

```
// app.js
const loadModule = async () => {
  const { default: log, PI } = await import('./math.js');
  log(PI);
};

loadModule();
```

Esta capacidad es especialmente útil para optimizar el rendimiento en aplicaciones grandes, permitiendo cargar partes del código solo cuando son necesarias.

4 Asincronía

4.1 Ejecución de JavaScript en un Solo Hilo

El código JavaScript, tanto en una pestaña del navegador como en un entorno de Node.js, se ejecuta en un solo hilo. Esto significa que, por diseño, todo el código JavaScript se ejecuta de manera secuencial en un único hilo de procesamiento. No es posible, utilizando solo JavaScript, crear múltiples procesos o hilos paralelos como se podría hacer en otros lenguajes de programación como Java o C++.

Por naturaleza, JavaScript es un lenguaje síncrono. Cada línea de código se ejecuta de manera ordenada, y la siguiente línea no se ejecuta hasta que la anterior ha terminado su trabajo. Esto simplifica el desarrollo, ya que elimina la complejidad de manejar problemas de concurrencia que podrían surgir si múltiples hilos estuvieran modificando los mismos recursos simultáneamente. De esta manera, no hay riesgo de que dos procesos interfieran entre sí, lo que facilita la escritura de código sin errores de sincronización.

Aunque esto podría parecer una limitación, en realidad es una característica que facilita la programación. Al evitar la concurrencia nativa, JavaScript previene una serie de problemas complejos relacio-

nados con la sincronización de recursos compartidos y las condiciones de carrera, lo que resulta en un entorno de desarrollo más seguro y sencillo.

Interacción Asíncrona en las Páginas Web: A pesar de que JavaScript es síncrono, las páginas web son inherentemente asíncronas. Los usuarios interactúan con las páginas web generando eventos en tiempo real, como hacer clic en botones, enviar formularios, hacer scroll, mover el ratón, desplegar menús, entre otros. Cada una de estas acciones ocurre de forma impredecible y en cualquier momento, lo que hace que la programación de aplicaciones web requiera un manejo eficaz de la asincronía.

Un ejemplo clave de asincronía es cuando JavaScript realiza peticiones HTTP para obtener datos de un servidor. El tiempo que tarda en recibir una respuesta no es fijo; depende de varios factores como la velocidad de la red o la carga del servidor. A pesar de esta incertidumbre, el resto del código JavaScript en la página debe seguir ejecutándose sin quedarse bloqueado esperando la respuesta.

Node.js es un entorno de ejecución de JavaScript en el servidor que también opera en un solo hilo. Al igual que en las páginas web, Node.js maneja tareas asíncronas de manera eficiente, permitiendo que operaciones como la lectura de archivos, consultas a bases de datos, o solicitudes HTTP no bloqueen el hilo principal. Sin embargo, a diferencia de una página web, Node.js es usado para aplicaciones del lado del servidor, lo que puede implicar manejar un gran número de solicitudes simultáneas.

4.2 Conceptos básicos de la ejecución síncrona vs. asíncrona

- Ejecución Síncrona: En la programación síncrona, las tareas se ejecutan de manera secuencial, una después de otra. El código no avanza hasta que la tarea actual se ha completado. Este enfoque es directo y fácil de entender, pero puede ser problemático cuando se manejan tareas que requieren mucho tiempo, como operaciones de entrada/salida (I/O), ya que el programa puede quedar "bloqueado" esperando que termine una tarea, impidiendo que otras tareas se ejecuten mientras tanto.

```
console.log('Inicio');
const resultado = calcular(5, 3); // Supongamos que esta función realiza una operación pesada
console.log(`Resultado: ${resultado}`);
console.log('Fin');
```

Explicación: En este ejemplo, `calcular(5, 3)` se ejecuta y el programa espera hasta que se obtenga el resultado antes de continuar con las siguientes líneas. Si calcular tarda mucho, el programa permanecerá inactivo durante ese tiempo.

- Ejecución Asíncrona: En la programación asíncrona, las tareas pueden iniciarse y luego el control puede pasar a la siguiente tarea sin esperar que la anterior termine. Esto permite que otras tareas se ejecuten mientras la primera tarea sigue en progreso. Una vez que la tarea asíncrona termina, se activa una función de callback o se resuelve una Promise, que se encarga de manejar el resultado.

```
console.log('Inicio');

setTimeout(() => {
    console.log('Este mensaje aparece después de 2 segundos');
}, 2000);

console.log('Fin');
```

Explicación: Aquí, setTimeout es una función asíncrona que programa una tarea para ejecutarse después de 2 segundos. El código no espera a que este tiempo pase, sino que sigue ejecutando console.log('Fin') inmediatamente. Después de 2 segundos, el mensaje programado se muestra.

4.3 El event loop y su importancia

El event loop es un concepto central en la asincronía de JavaScript. Es un mecanismo que permite a JavaScript realizar operaciones no bloqueantes a pesar de que es un lenguaje de un solo hilo.

Funcionamiento básico:

- Stack: Cuando se llama a una función, esta se agrega al stack (pila) de llamadas. Si la función llama a otra función, esta última se agrega al stack, y así sucesivamente.
- Web APIs: Cuando se realiza una operación asíncrona, como un setTimeout o una solicitud HTTP (fetch), se envía la tarea a una Web API del navegador, que la maneja fuera del stack principal.
- Task Queue: Una vez que la operación asíncrona completa su trabajo (por ejemplo, el tiempo del setTimeout se agota), la callback asociada se coloca en la "cola de tareas" (task queue).
- Event Loop: El event loop verifica constantemente si el stack está vacío. Si lo está y hay tareas en la task queue, saca la primera tarea de la cola y la coloca en el stack para ejecutarla.

```
console.log('Inicio');

setTimeout(() => {
    console.log('Timeout 1');
}, 0);

Promise.resolve().then(() => {
    console.log('Promise 1');
});

console.log('Fin');
```

Salida esperada:

```
Inicio
Fin
Promise 1
Timeout 1
```

Explicación detallada:

- console.log('Inicio'): Se ejecuta inmediatamente y se muestra "Inicio".

- setTimeout(() => { console.log('Timeout 1'); }, 0);: Este setTimeout coloca su callback en la task queue, pero como tiene un tiempo de espera de 0 ms, aún debe esperar hasta que el stack esté vacío.
- Promise.resolve().then(() => { console.log('Promise 1'); });: Las promesas resueltas colocan su then en la microtask queue, que se procesa antes que la task queue.
- console.log('Fin'): Se ejecuta inmediatamente y se muestra "Fin".
- Event Loop: Después de que el stack esté vacío, el event loop procesa primero las tareas en la microtask queue (en este caso, la promesa resuelta), mostrando "Promise 1". Luego, procesa la task queue, mostrando "Timeout 1".

4.4 Callbacks

4.4.1 Definición y usos

Un callback es una función que se pasa como argumento a otra función, y que se ejecuta después de que la función receptora haya completado su tarea. Este patrón es una forma común de manejar operaciones asíncronas en JavaScript, especialmente antes de la introducción de Promises y async/a-wait.

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

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

saludar('Juan', despedirse);
```

Explicación:

- saludar: Es una función que toma dos argumentos: nombre y callback.
- callback: Es una función (en este caso, despedirse) que se ejecuta después de que saludar haya mostrado el mensaje "Hola, Juan".
- Salida: El código primero imprime "Hola, Juan", luego ejecuta el callback que imprime "Adiós".

Usos comunes de callbacks:

- Manipulación del DOM: A menudo se usan en eventos como onclick, onload, etc.

```
<!-- index.html -->
<html>
<head>
</head>
<body>
    <button onclick="pulsame()">Púlsame</button>
```

```
    <button id="boton">Botón</button>
    <script type="text/javascript" src="app.js"></script>
</body>
</html>
```

```
// app.js
function pulsame() {
    console.log("¡Pulsado!");
}

document.querySelector("#boton").addEventListener("click", () => {
    console.log("¡Prueba!");
});

window.addEventListener('load', () => console.log("Página cargada"));

const callback = () => {
    console.log("Función que no hace nada");
};

setTimeout(callback, 2000);

const callbackAddP = () => {
    let p = document.createElement("p");
    p.append("¡Hola mundo!");
    document.body.appendChild(p);
};

let interval = setInterval(callbackAddP, 2000);
```

Ejemplo boredapi:

```
<!-- index.html -->
<html>
<head>
</head>
<body>
    <div id="texto"></div>
    <script>
    let request = new XMLHttpRequest();
    request.open("GET", "https://www.boredapi.com/api/activity");
    request.send();

    request.onload = function() {
      if (request.status === 200) {
        const respuesta = JSON.parse(request.responseText);
        document.getElementById('texto').innerHTML = respuesta.activity;
      } else {
        console.log(request.statusText);
      }
    };

    request.onerror = request.ontimeout = function(e) {
      console.log(e.type);
    };
  </script>
</body>
</html>
```

- Operaciones de entrada/salida (I/O): En operaciones como la lectura de archivos o peticiones HTTP.

```
fs.readFile('/archivo.txt', 'utf8', function(err, data) {
    if (err) {
        console.error('Error al leer el archivo');
    } else {
        console.log(data);
    }
});
```

4.5 Problemas de los callbacks: el Callback Hell

¿Qué es el Callback Hell? El Callback Hell ocurre cuando hay múltiples operaciones asíncronas encadenadas, y cada una depende del resultado de la anterior. Esto puede llevar a una estructura de código profundamente anidada y difícil de leer y mantener. Este problema se agrava a medida que aumenta el número de callbacks.

```
doSomething(function(result) {
    doSomethingElse(result, function(newResult) {
        doThirdThing(newResult, function(finalResult) {
            console.log('Resultado final: ' + finalResult);
        });
    });
});
```

Explicación:

- Cada función depende de la anterior, lo que provoca una anidación excesiva.
- Si alguna de las funciones falla, el manejo de errores se vuelve complicado.
- La estructura es difícil de leer, entender y depurar, especialmente en aplicaciones más grandes.

Problemas del Callback Hell:

- Legibilidad: El código se vuelve difícil de leer y seguir.
- Mantenimiento: A medida que el código crece, se hace complicado mantenerlo y refactorizarlo.
- Manejo de errores: Capturar y manejar errores en un entorno tan anidado puede ser engorroso.

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Ejemplo de Callback Hell</title>
</head>
<body>
    <div id="content">
        <h1>Información del Usuario</h1>
        <div id="user"></div>
        <h2>Posts del Usuario</h2>
        <div id="posts"></div>
        <h3>Comentarios del Primer Post</h3>
        <div id="comments"></div>
    </div>
```

```
    <script>
        // Función para realizar solicitudes HTTP (simulando XMLHttpRequest)
        function fetchData(url, callback) {
            const xhr = new XMLHttpRequest();
            xhr.open('GET', url);
            xhr.onload = function() {
                if (xhr.status === 200) {
                    callback(null, JSON.parse(xhr.responseText));
                } else {
                    callback(`Error ${xhr.status}: ${xhr.statusText}`, null);
                }
            };
            xhr.onerror = function() {
                callback('Error de conexión', null);
            };
            xhr.send();
        }

        // Iniciamos la secuencia de callbacks
        fetchData('https://jsonplaceholder.typicode.com/users/1', function(error, user) {
            if (error) {
                return console.error('Error al obtener el usuario:', error);
            }
            // Actualizar el DOM con la información del usuario
            const userDiv = document.getElementById('user');
            userDiv.innerHTML = `<p><strong>Nombre:</strong> ${user.name}</p>
                                 <p><strong>Email:</strong> ${user.email}</p>`;

            fetchData(`https://jsonplaceholder.typicode.com/users/${user.id}/posts`, function
                (error, posts) {
                if (error) {
                    return console.error('Error al obtener los posts:', error);
                }
                // Actualizar el DOM con los posts del usuario
                const postsDiv = document.getElementById('posts');
                postsDiv.innerHTML = posts.map(post => `<p><strong>${post.title}</strong>: ${
                    post.body}</p>`).join('');

                if (posts.length > 0) {
                    fetchData(`https://jsonplaceholder.typicode.com/posts/${posts[0].id}/
                        comments`, function(error, comments) {
                        if (error) {
                            return console.error('Error al obtener los comentarios:', error);
                        }
                        // Actualizar el DOM con los comentarios del primer post
                        const commentsDiv = document.getElementById('comments');
                        commentsDiv.innerHTML = comments.map(comment => `<p><strong>${comment
                            .name}</strong>: ${comment.body}</p>`).join('');
                    });
                } else {
                    console.log('El usuario no tiene posts.');
                }
            });
        });
    </script>
</body>
</html>
```

Cómo manejar el Callback Hell:

- Modularización: Separar cada función en módulos más pequeños y manejables.
- Promises: Convertir funciones basadas en callbacks en Promises, lo que permite un código más

limpio y manejable.

- Async/Await: Es la forma moderna y más legible de manejar la asincronía, eliminando la necesidad de anidación profunda.

4.6 Excepciones

Uno de los desafíos al usar callbacks en JavaScript es la gestión de excepciones, especialmente cuando se trata de operaciones asíncronas como solicitudes HTTP. Para entender mejor este problema, consideremos el siguiente escenario:

Supongamos que hacemos una solicitud HTTP a una URL que no existe. Si la solicitud falla, se ejecuta el callback request.onerror, que es responsable de manejar los errores. En un intento por gestionar adecuadamente estos errores, podríamos intentar lanzar una excepción dentro de este callback y atraparla con un bloque try/catch. Un ejemplo de este código sería:

```
let request = new XMLHttpRequest();
request.open("GET", "https://www.apiquenoexiste.com/api/activity");

try {
    request.onerror = request.ontimeout = function(e) {
        console.log(e.type);
        throw new Error('No hay conexión con el backend.');
    };
} catch (error) {
    document.getElementById('texto').innerHTML = error;
}

request.send();
```

¿Por qué no se atrapa la excepción?

Cuando ejecutamos este código, observamos que la excepción lanzada dentro del callback request.onerror no es atrapada por el bloque try/catch. La razón de esto es que el try/catch en JavaScript solo puede capturar excepciones que ocurren de forma síncrona dentro de su bloque de código. En otras palabras, cuando el try/catch se ejecuta, registra la función onerror como un callback, pero este callback no se ejecuta inmediatamente. En cambio, se ejecuta de manera asíncrona después de que la solicitud falla.

Esto significa que para cuando el callback onerror se ejecuta y lanza la excepción, el bloque try/catch ya ha finalizado su ejecución, y por lo tanto, no puede capturar la excepción. Este comportamiento es una característica inherente de cómo JavaScript maneja la asincronía: los callbacks asíncronos se ejecutan en el futuro, fuera del contexto del bloque try/catch que los registró.

- Vamos a refactorizar el siguiente código para tener una función que retorne la respuesta del servidor.

```
<!-- index.html -->
```

```
<html>
<head>
</head>
<body>
    <div id="texto"></div>
    <script>
    let request = new XMLHttpRequest();
    request.open("GET", "https://www.boredapi.com/api/activity");

    request.onload = function() {
      if (request.status === 200) {
        const respuesta = JSON.parse(request.responseText);
        document.getElementById('texto').innerHTML = respuesta.activity;
      } else {
        console.log(request.statusText);
      }
    };
    request.onerror = request.ontimeout = function(e) {
      console.log(e.type);
    };
    request.send();
  </script>
</body>
</html>
```

```
<!-- index.html -->
<html>
<head>
</head>
<body>
    <div id="texto"></div>
    <script>
    function boredActivity() {
      let request = new XMLHttpRequest();
      request.open("GET", "https://www.boredapi.com/api/activity");
      request.onload = function() {
        if (request.status === 200) {
          const respuesta = JSON.parse(request.responseText);
          return respuesta.activity;
      } else { return 'Error'; }
      }
      request.onerror = request.ontimeout = function(e) { return 'Error';}
      request.send();
    }
    const activity = boredActivity();
    console.log(activity);
  </script>
</body>
</html>
```

- Al ejecutar el código el valor de activity es undefined.
- En la función boredActivity se registran los callbacks pero cuando se ejecutan y hacen return, la función boredActivity ya finalizó su ejecución.
- Lo que vemos es el return implícito de la función boredActivity que se ejecuta de forma síncrona antes de los callbacks.

Como vemos no podemos hacer un return dentro de un callback. La función que la registra ya no existe cuando se ejecuta. La solución consiste en refactorizar usando más callbacks pasados como

parámetros a la función boredActivity.

```
<!-- index.html -->
<html>
<head>
</head>
<body>
    <div id="texto"></div>
    <script>
    function boredActivity(done, error) {
      let request = new XMLHttpRequest();
      request.open("GET", "https://www.boredapi.com/api/activity");
      request.onload = function() {
        if (request.status === 200) {
          const respuesta = JSON.parse(request.responseText);
          done(respuesta.activity);
      } else { error('Error'); }
      }
      request.onerror = request.ontimeout = function(e) { error‘(’Error);}
      request.send();
    }
    const activity = boredActivity((mensaje)=>{console.log(mensaje);}, (error)=>{console.log(
        error);});
    console.log(activity);
  </script>
</body>
</html>
```

- Esta última solución es mucho mejor que la anterior. La función boredActivity se encarga solo de hacer la petición http y delega a las funciones callback done y error qué hacer con la respuesta.
- Es una función menos acoplada.

Ahora incluso podemos moverla a otro módulo para reutilizar código. Usamos parcel para el despliegue.

```
<!-- index.html -->
<html>
<head>
</head>
<body>
    <div id="texto"></div>
    <script type="module" src="app.js"></script>
</body>
</html>
```

```
//boredapi.js
function boredActivity(done, error) {
      let request = new XMLHttpRequest();
      request.open("GET", "https://www.boredapi.com/api/activity");
      request.onload = function() {
        if (request.status === 200) {
          const respuesta = JSON.parse(request.responseText);
          done(respuesta.activity);
      } else { error('Error'); }
      }
      request.onerror = request.ontimeout = function(e) { error('Error');}
      request.send();
    }
export default boredActivity;
```

```
//app.js
import boredActivity from './boredapi';

boredActivity(
    (mensaje) => {
        document.getElementById('texto').innerHTML = mensaje;
},
(error) => {
    alert(error);
}
);
```

4.7 Promesas

Las Promises (promesas) son un concepto fundamental en JavaScript moderno para manejar operaciones asíncronas de una manera más limpia y manejable que con los tradicionales callbacks. Introducidas en ES6 (ECMAScript 2015), las Promises permiten trabajar con código asíncrono de forma que se evite el Callback Hell y se mejore la legibilidad y el mantenimiento del código.

Una Promise es un objeto que representa la eventual finalización (o fallo) de una operación asíncrona y su valor resultante. Una Promise puede estar en uno de estos tres estados:

- Pending (Pendiente): La operación asíncrona aún no ha finalizado.
- Fulfilled (Cumplida): La operación asíncrona se ha completado con éxito, y la Promise tiene un valor resultante.
- Rejected (Rechazada): La operación asíncrona ha fallado, y la Promise tiene un motivo de fallo.

```
const promesa = new Promise((resolve, reject) => {
    let exito = true;
    if (exito) {
        resolve('La operación fue exitosa.');
    } else {
        reject('Hubo un error.');
    }
});
```

Explicación:

- resolve: Es una función que se llama cuando la operación asíncrona se completa con éxito. El argumento que se pasa a resolve se convierte en el valor de la Promise cuando esta se cumple (fulfilled).
- reject: Es una función que se llama cuando la operación asíncrona falla. El argumento que se pasa a reject se convierte en el motivo de rechazo (reason) cuando la Promise es rechazada (rejected).

4.7.1 Estados de una Promise

Estados de las Promises:

- Pending (Pendiente): Es el estado inicial cuando se crea una Promise. En este estado, la Promise no ha sido ni cumplida ni rechazada.
- Fulfilled (Cumplida): La Promise ha sido resuelta con éxito. Una vez que se llama a resolve, la Promise entra en este estado.
- Rejected (Rechazada): La Promise ha sido rechazada debido a un error o fallo en la operación. Esto ocurre cuando se llama a reject.

```
const promesa = new Promise((resolve, reject) => {
    console.log('Promise iniciada');
    setTimeout(() => {
        resolve('Éxito');
        // O usar reject('Error') para simular un fallo
    }, 2000);
});

promesa
    .then(result => console.log(result))
    .catch(error => console.error(error));
```

Explicación:

- El código imprime “Promise iniciada” inmediatamente.
- Después de 2 segundos, la Promise se resuelve (estado Fulfilled), y se imprime “Éxito”.
- Si hubiéramos llamado a reject en lugar de resolve, se habría imprimido “Error”.

4.7.2 Creación de Promises

Las Promises se crean utilizando el constructor Promise, que acepta una función ejecutora (executor function) como argumento. Esta función ejecutora toma dos parámetros: resolve y reject, que se utilizan para cambiar el estado de la Promise.

```
function esperar(ms) {
    return new Promise((resolve) => {
        setTimeout(resolve, ms);
    });
}

esperar(2000).then(() => console.log('Pasaron 2 segundos'));
```

Explicación:

- esperar(ms): Esta función devuelve una Promise que se cumple después de que hayan pasado ms milisegundos.

- El método setTimeout se utiliza para simular una operación asíncrona. Cuando el temporizador finaliza, se llama a resolve, cumpliendo la Promise.

4.7.3 Métodos then(), catch() y finally()

Método then(): El método then() se utiliza para manejar el valor de una Promise una vez que se ha cumplido. Acepta dos argumentos opcionales:

- Una función que se ejecuta si la Promise se cumple (onFulfilled).
- Una función que se ejecuta si la Promise se rechaza (onRejected).

Ejemplo del Uso de then():

```
promesa.then(result => {
    console.log('Operación exitosa:', result);
});
```

Método catch(): El método catch() se utiliza para manejar errores en una Promise. Es un atajo para then(null, onRejected), y captura cualquier error que ocurra en la cadena de Promises.

Ejemplo del Uso de catch():

```
promesa
    .then(result => {
        throw new Error('Algo salió mal');
    })
    .catch(error => {
        console.error('Error atrapado:', error);
    });
```

Método finally(): El método finally() se ejecuta después de que la Promise se haya cumplido o rechazado, sin importar el resultado. Se utiliza para realizar limpieza o tareas finales que deben ejecutarse independientemente del éxito o fracaso de la operación.

Ejemplo del Uso de finally():

```
promesa
    .then(result => console.log('Resultado:', result))
    .catch(error => console.error('Error:', error))
    .finally(() => console.log('Operación finalizada'));
```

```
function obtenerTemperatura(ciudad) {
    return new Promise((resolve, reject) => {
        console.log(`Buscando la temperatura de ${ciudad}...`);
        setTimeout(() => {
            const datos = {
                "Nueva York": 22,
                "Londres": 15,
                "Tokio": 30
            };

            if (datos[ciudad] !== undefined) {
```

```
                resolve(`La temperatura en ${ciudad} es de ${datos[ciudad]}°C`);
            } else {
                reject('No se pudo encontrar la temperatura para la ciudad solicitada.');
            }
        }, 2000); // Simula un retraso en la respuesta
    });
}

// Uso de la función con Promises
obtenerTemperatura('Londres')
    .then(temperatura => console.log(temperatura))
    .catch(error => console.error(error))
    .finally(() => console.log('Consulta completada.'));
```

Explicación:

- obtenerTemperatura(ciudad): Esta función devuelve una Promise que simula la obtención de la temperatura de una ciudad. Después de 2 segundos, la Promise se cumple (resolve) con el valor de la temperatura si la ciudad es válida, o se rechaza (reject) si la ciudad no se encuentra en los datos simulados.
- then(temperatura): Se ejecuta cuando la Promise se cumple y maneja la temperatura obtenida.
- catch(error): Se ejecuta si la Promise es rechazada y maneja el error.
- finally(): Este bloque se ejecuta siempre, sin importar si la Promise fue cumplida o rechazada.

4.7.4 Encadenamiento de Promises

Encadenamiento de Promises: Las Promises se pueden encadenar, lo que significa que puedes realizar múltiples operaciones asíncronas en secuencia, cada una dependiendo del resultado de la anterior. Cada then() devuelve una nueva Promise, lo que permite que se realice un encadenamiento limpio y fácil de leer.

```
function paso1() {
    return new Promise((resolve) => {
        setTimeout(() => resolve('Paso 1 completado'), 1000);
    });
}

function paso2(mensajeAnterior) {
    return new Promise((resolve) => {
        setTimeout(() => resolve(mensajeAnterior + ', Paso 2 completado'), 1000);
    });
}

function paso3(mensajeAnterior) {
    return new Promise((resolve) => {
        setTimeout(() => resolve(mensajeAnterior + ', Paso 3 completado'), 1000);
    });
}

paso1()
    .then(result => paso2(result))
    .then(result => paso3(result))
    .then(result => console.log('Todos los pasos completados:', result));
```

Explicación:

```
paso1(): Se ejecuta primero y se resuelve después de 1 segundo.
paso2(): Se ejecuta después de que paso1() se haya completado.
paso3(): Se ejecuta después de que paso2() se haya completado.
Finalmente, se imprime el mensaje concatenado de todos los pasos.
```

4.7.5 Manejo de Errores en Promises

Manejo de Errores con catch(): Cuando trabajas con Promises, es importante manejar los posibles errores que pueden ocurrir durante la operación asíncrona. Si en cualquier parte de la cadena de Promises ocurre un error, este se propagará por la cadena hasta que encuentre un método catch() que lo maneje.

Ejemplo de Manejo de Errores:

```
function puedeFallar() {
    return new Promise((resolve, reject) => {
        const exito = Math.random() > 0.5;
        if (exito) {
            resolve('Operación exitosa');
        } else {
            reject('Operación fallida');
        }
    });
}

puedeFallar()
    .then(result => {
        console.log(result);
    })
    .catch(error => {
        console.error('Error atrapado:', error);
    });
```

Explicación:

- puedeFallar(): Esta función tiene un 50% de probabilidad de fallar (dependiendo del valor aleatorio).
- Si falla, el error es capturado por el catch().

Encadenamiento y Errores: Un catch() en la cadena de Promises no solo captura errores de la Promise original, sino también cualquier error que ocurra en los métodos then() anteriores.

```
paso1()
    .then(result => {
        throw new Error('Error en paso 2');
        return paso2(result);
    })
    .then(result => paso3(result))
    .catch(error => {
        console.error('Error atrapado en la cadena:', error);
    });
```

Explicación:

- Si ocurre un error en cualquier parte del encadenamiento, el catch() lo atrapará, asegurando que el error no pase desapercibido.

4.8 Fetch

La API fetch permite realizar peticiones HTTP a un servidor backend para obtener información de una API. Una de las características principales de fetch es que devuelve una promesa que representa la respuesta de la petición.

```
fetch('https://www.boredapi.com/api/activity')
  .then(response => {
    console.log(response);
  })
  .catch(error => {
    console.error('Hubo un problema con la petición:', error);
  });
```

En este ejemplo, realizamos una petición a la API y capturamos la respuesta. El objeto response tiene muchas propiedades que describen la respuesta HTTP, como status, statusText, headers, y body. Es importante notar que la propiedad body aún no contiene el JSON que esperamos; en su lugar, contiene un stream que necesitamos procesar.

Para acceder al contenido del cuerpo de la respuesta, debemos manejar otra promesa. Una de las formas de hacerlo es utilizando el método text() de la respuesta, que nos da el contenido en formato de texto plano.

```
fetch('https://www.boredapi.com/api/activity')
  .then(response => response.text())
  .then(data => {
    const parsedData = JSON.parse(data);
    console.log(parsedData.activity);
  })
  .catch(error => {
    console.error('Hubo un problema con la conversión del texto:', error);
  });
```

En este caso, data es un string, por lo que necesitamos convertirlo en un objeto JavaScript utilizando JSON.parse.

Para simplificar el proceso, fetch ofrece un método json() que retorna directamente una promesa que se resuelve con el objeto JSON ya parseado:

```
fetch('https://www.boredapi.com/api/activity')
  .then(response => response.json())
  .then(data => {
    console.log(data.activity);
  })
  .catch(error => {
    console.error('Hubo un problema con la conversión a JSON:', error);
```

```
  });
```

Podemos encadenar múltiples promesas para realizar operaciones más complejas. A continuación, mostramos un ejemplo que ilustra cómo encadenar peticiones para obtener datos de un usuario desde un archivo JSON local y luego hacer una segunda petición para obtener datos adicionales desde la API de GitHub:

Archivo user.json

```
{
  "name": "lmol"
}
```

Archivo app.js

```
fetch('user.json')
  .then(response => response.json())
  .then(user => {
    return fetch(`https://api.github.com/users/${user.name}`);
  })
  .then(response => response.json())
  .then(githubUser => {
    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    document.body.append(img);

    // Eliminar la imagen después de 3 segundos
    setTimeout(() => img.remove(), 3000);
  })
  .catch(error => {
    console.error('Hubo un problema en alguna de las peticiones:', error);
  });
```

En este ejemplo:

- Realizamos una petición para obtener un archivo user.json que contiene el nombre de usuario.
- Utilizamos el nombre de usuario para realizar una segunda petición a la API de GitHub y obtener información del usuario.
- Creamos un elemento img y asignamos la URL del avatar del usuario de GitHub como su fuente.
- Finalmente, el img se añade al documento y se elimina después de 3 segundos.

Es importante manejar errores que pueden ocurrir durante cualquiera de las promesas encadenadas. El bloque catch al final de la cadena captura cualquier error ocurrido durante la ejecución de cualquiera de las promesas previas, ya sea debido a un problema de red, una respuesta malformada, o cualquier otro tipo de excepción.

4.8.1 Ejemplos

Vamos a refactorizar el ejercicio realizado con callbacks para la función boredActivity utilizando promesas. El uso de promesas ofrece varias ventajas significativas que hacen que el código sea más limpio, legible y fácil de mantener:

- Múltiples Callbacks: Con las promesas, podemos registrar varios callbacks que se ejecutarán cuando se obtenga el dato. Esto permite manejar diferentes escenarios o acciones que dependen del resultado de la promesa.
- Gestión de Errores: Las promesas proporcionan un mecanismo para capturar y gestionar errores de manera centralizada. Esto es especialmente útil en aplicaciones asíncronas, donde pueden ocurrir errores en distintas etapas del flujo de datos.
- Encadenamiento de Acciones Asíncronas: Las promesas permiten encadenar diferentes acciones asíncronas de manera secuencial. Esto simplifica la lógica y evita el "callback hell", que es común cuando se trabaja con funciones asíncronas anidadas.

La función boredActivity consulta una API que devuelve una actividad para hacer cuando estás aburrido. Vamos a convertirla para que utilice promesas:

```
function boredActivity() {
    return new Promise((resolve, reject) => {
        let request = new XMLHttpRequest();
        request.open("GET", "https://www.boredapi.com/api/activity");
        request.onload = function() {
            if (request.status === 200) {
                const respuesta = JSON.parse(request.responseText);
                resolve(respuesta.activity);  // Se resuelve la promesa con la actividad
                    obtenida
            } else {
                reject(`Error: ${request.statusText}`);  // Se rechaza la promesa con el
                    mensaje de error
            }
        };
        request.onerror = request.ontimeout = function() {
            reject("Error de red o tiempo de espera agotado");  // Se rechaza en caso de
                error de red o timeout
        };
        request.send();  // Se envía la solicitud a la API
    });
}
```

Una vez que hemos refactorizado la función para que utilice promesas, podemos usarla de la siguiente manera:

```
import boredActivity from './boredapi.js';

boredActivity()
    .then((mensaje) => {
        document.getElementById("texto").innerHTML = mensaje;
        // Ejemplo adicional: puedes realizar otra acción con el mensaje
        console.log(`Actividad sugerida: ${mensaje}`);
    })
    .catch((error) => {
```

```
        alert(`Ocurrió un error: ${error}`);
        // Ejemplo adicional: loguear el error para análisis
        console.error(`Error detallado: ${error}`);
    });
```

- Promesa en boredActivity:
  - resolve(respuesta.activity): Si la solicitud es exitosa, la promesa se resuelve con la actividad obtenida.
  - reject(request.statusText): Si la solicitud falla, la promesa se rechaza con el mensaje de error correspondiente.
  - Manejo de errores de red y tiempo de espera: Mediante onerror y ontimeout, capturamos posibles fallos de red o si la solicitud tarda demasiado tiempo en completarse.
- Encadenamiento y Manejo de la Promesa:
  - then((mensaje) => {…}): El bloque then se ejecuta cuando la promesa se resuelve correctamente, permitiendo manipular el resultado. En este caso, actualizamos el contenido de un elemento HTML.
  - catch((error) => {…}): El bloque catch maneja cualquier error que ocurra durante la ejecución de la promesa, proporcionando un lugar centralizado para manejar errores y alertar al usuario.

Encadenamiento de Promesas: Si quisieras realizar otra solicitud basada en el resultado de la primera, podrías encadenar promesas:

```
boredActivity()
    .then((mensaje) => {
        document.getElementById("texto").innerHTML = mensaje;
        return fetchAdditionalInfo(mensaje);  // Supongamos que esta función devuelve otra
            promesa
    })
    .then((infoAdicional) => {
        console.log("Información adicional:", infoAdicional);
    })
    .catch((error) => {
        console.error("Se produjo un error:", error);
    });
```

Múltiples Callbacks con Promise.all: Si necesitas ejecutar múltiples solicitudes de forma concurrente y esperar a que todas se completen:

```
    Promise.all([boredActivity(), anotherAsyncFunction()])
        .then(([mensaje, otroResultado]) => {
            console.log("Ambas operaciones completadas:", mensaje, otroResultado);
        })
        .catch((error) => {
            console.error("Error en una de las operaciones:", error);
        });
```

Esta refactorización no solo mejora la estructura y legibilidad del código, sino que también lo hace

más robusto y fácil de escalar, especialmente en aplicaciones que dependen en gran medida de operaciones asíncronas.

4.9 Async y await

En la especificación de ES2017, se introdujo una nueva sintaxis que facilita el manejo de la asincronía en JavaScript: async/await. Esta sintaxis simplifica la forma en que trabajamos con promesas, permitiéndonos escribir código asincrónico de una manera que se asemeja más a la programación síncrona. Ejemplo con Promesas (then/catch)

Consideremos el siguiente ejemplo que utiliza la sintaxis clásica de promesas:

```
fetch("https://www.boredapi.com/api/activity")
  .then(response => response.json())
  .then(data => {
    console.log(data.activity);
  })
  .catch(err => {
    console.log(err);
  });
```

Este código realiza una solicitud a la API y maneja la respuesta utilizando then para procesar los datos y catch para manejar posibles errores.

Transformación a async/await

Podemos transformar este código utilizando async/await, lo que nos permite escribirlo de manera más clara y estructurada:

```
async function getActivity() {
  try {
    const response = await fetch("https://www.boredapi.com/api/activity");
    const data = await response.json();
    console.log(data.activity);
  } catch (error) {
    console.log(error);
  }
}

getActivity();
```

Con async/await, podemos escribir nuestro código asincrónico como si fuera síncrono. La palabra clave await hace que el código “espere” a que la promesa se resuelva antes de continuar con la siguiente línea. Además, cualquier función que use await debe declararse como asíncrona utilizando la palabra clave async antes de function.

Es importante notar que async/await es solo “syntactic sugar” sobre las promesas. En el fondo, todo sigue siendo promesas. Retorno de Valores en Funciones Asíncronas

Cuando una función asíncrona retorna un valor, en realidad está retornando una promesa que resolverá ese valor en el futuro:

```
async function getActivity() {
  const response = await fetch("https://www.boredapi.com/api/activity");
  const data = await response.json();
  return data.activity;
}

const activityPromise = getActivity();
console.log(activityPromise);  // Esto imprimirá una promesa
```

Si queremos obtener el valor retornado, debemos utilizar then para manejar la promesa o utilizar await dentro de otra función asíncrona:

```
getActivity().then(activity => console.log(activity));

// O dentro de otra función asíncrona
async function displayActivity() {
  const activity = await getActivity();
  console.log(activity);
}

displayActivity();
```

Manejo de Excepciones

Otra ventaja de async/await es la posibilidad de manejar errores de manera más intuitiva utilizando bloques try/catch:

```
async function getActivity() {
  try {
    const response = await fetch("https://www.boredapi.com/api/activity");
    const data = await response.json();
    return { data: data.activity, error: null };
  } catch (error) {
    return { data: null, error: error };
  }
}

getActivity().then(result => {
  if (result.error === null) {
    console.log(result.data);
  } else {
    console.log("Error:", result.error);
  }
});
```

En este ejemplo, encapsulamos el valor y el error en un objeto que retornamos. Esto nos permite manejar tanto los datos como los errores de manera más flexible cuando consumimos la función.

5 DOM

5.1 Introducción al DOM

El Document Object Model, comúnmente conocido como DOM, es una de las piezas fundamentales en la construcción y manipulación de páginas web. Para entender su importancia y cómo se relaciona con otros aspectos del desarrollo web, es crucial tener claro qué es el DOM, cómo funciona y cuál es su papel en la interacción entre HTML, CSS y JavaScript.

El DOM es una interfaz de programación para documentos HTML y XML. Define la estructura lógica de los documentos y la forma en que se accede y manipula dicha estructura. En esencia, el DOM es una representación en forma de árbol de los elementos en una página web, donde cada nodo del árbol representa una parte del documento (ya sea un elemento, un atributo o un texto).

Cada nodo del DOM puede ser manipulado utilizando JavaScript, permitiendo a los desarrolladores cambiar dinámicamente el contenido, estructura y estilo de una página web sin necesidad de recargarla. Esta capacidad de modificar el documento sobre la marcha es lo que permite la creación de aplicaciones web interactivas y dinámicas.

El DOM es crucial porque actúa como puente entre el contenido de un documento y la manera en que los usuarios interactúan con él. Antes de la existencia del DOM, los desarrolladores estaban limitados a crear páginas web estáticas. Con el DOM, es posible actualizar cualquier parte del documento en respuesta a eventos del usuario, como clics de ratón, entradas de teclado, o acciones similares.

Por ejemplo, cuando se llena un formulario en una página web y se envía sin recargar la página, esto es posible gracias al DOM. JavaScript puede acceder al DOM, recoger los datos del formulario, enviar esos datos al servidor y, basado en la respuesta del servidor, actualizar el contenido de la página instantáneamente.

Además, el DOM es independiente del lenguaje de programación. Aunque JavaScript es el lenguaje más comúnmente utilizado para interactuar con el DOM en los navegadores web, otros lenguajes también pueden manipular el DOM en otros contextos, como el desarrollo de aplicaciones móviles híbridas o scripts de automatización.

- HTML y el DOM: Cuando se carga una página web, el navegador toma el código HTML y lo convierte en un modelo de objetos, que es el DOM. El DOM representa la estructura del documento en forma de un árbol de nodos. Cada elemento HTML se convierte en un nodo que puede ser manipulado usando JavaScript.
- CSS y el DOM: El DOM también interactúa con CSS. Aunque CSS define la presentación visual de un documento, los estilos se aplican a los elementos representados en el DOM. JavaScript puede acceder y modificar estos estilos directamente a través del DOM, lo que permite cambiar dinámicamente la apariencia de la página.

- JavaScript y el DOM: JavaScript es el lenguaje que permite a los desarrolladores interactuar con el DOM. A través de diferentes métodos y propiedades, JavaScript puede seleccionar elementos específicos del DOM, modificar su contenido, cambiar sus estilos, añadir o quitar nodos, y responder a eventos del usuario. Esta interacción es la base de las páginas web dinámicas.

El DOM, por tanto, es esencial para el desarrollo web moderno, ya que permite que los desarrolladores creen experiencias interactivas y adaptativas para los usuarios. Entender cómo funciona el DOM y cómo manipularlo eficientemente es una habilidad crucial para cualquier desarrollador web.

5.2 Estructura del DOM

El DOM se puede entender como una estructura jerárquica o en forma de árbol, donde cada parte del documento HTML (elementos, atributos, textos, etc.) se representa como un nodo dentro de este árbol. Comprender la estructura del DOM es fundamental para poder manipularlo de manera efectiva con JavaScript.

5.2.1 Nodos y Árbol del DOM

El DOM está compuesto por diferentes tipos de nodos, cada uno representando una parte específica del documento HTML. Estos nodos están organizados en una estructura de árbol, donde cada nodo puede tener uno o varios nodos hijos, un nodo padre, y nodos hermanos.

5.2.1.1 Tipos de nodos En el DOM, existen varios tipos de nodos que representan las distintas partes del documento:

- Nodos de Elemento: Representan los elementos HTML, como `<div>`, `<p>`, `<a>`, etc. Estos son los nodos más comunes y contienen otros nodos, como atributos y texto.
- Nodos de Atributo: Estos nodos representan los atributos de los elementos HTML, como class, id, href, etc. Aunque son importantes, en la mayoría de las manipulaciones DOM, los atributos se manejan a través de las propiedades del nodo de elemento en lugar de acceder directamente a los nodos de atributo.
- Nodos de Texto: Estos nodos contienen el texto dentro de los elementos HTML. Por ejemplo, en `<p>Hola Mundo</p>`, “Hola Mundo” es un nodo de texto.
- Nodos de Comentario: Representan los comentarios en el HTML. Por ejemplo, `<!-- Este es un comentario -->`.
- Nodo de Documento: Es el nodo raíz del documento, el punto de entrada al DOM, representado por el objeto document en JavaScript.

5.2.2 Raíz del documento (document)

El nodo raíz de cualquier documento HTML es el nodo document. Este nodo actúa como la entrada principal al DOM y permite acceder y manipular todos los otros nodos dentro del documento. Por ejemplo, para seleccionar un elemento específico en la página, se comienza desde el nodo document y se utiliza un método como getElementById o querySelector.

5.3 Relaciones entre nodos

El DOM establece una relación jerárquica entre los nodos, similar a una estructura familiar, donde los nodos pueden ser padres, hijos o hermanos.

5.3.1 Nodos padres e hijos

En la estructura del DOM, un nodo que contiene otros nodos se conoce como nodo padre, y los nodos dentro de él se llaman nodos hijos. Por ejemplo, en el siguiente código HTML:

```
<div>
  <p>Este es un párrafo</p>
</div>
```

El elemento es el nodo padre del elemento <p>, y <p> es un nodo hijo de .

5.3.1.1 Nodos hermanos Los nodos hermanos son aquellos que comparten el mismo nodo padre. Siguiendo el ejemplo anterior, si añadimos otro párrafo:

```
<div>
    <p>Este es un párrafo</p>
    <p>Este es otro párrafo</p>
</div>
```

Ambos elementos <p> son nodos hermanos, ya que comparten el mismo nodo padre .

5.3.2 Nodo raíz

El nodo raíz de cualquier documento HTML es el elemento <html>. Dentro de este nodo se encuentran los elementos <head> y <body>, que a su vez contienen otros nodos hijos. Toda la estructura del documento se organiza a partir de este nodo raíz.

```
<html>
  <head>
    <title>Mi Página Web</title>
  </head>
```

```
  <body>
    <div id="contenedor">
      <h1>Bienvenidos</h1>
      <p>Este es un párrafo de introducción.</p>
    </div>
  </body>
</html>
```

5.4 Selección de Elementos en el DOM

Una vez que comprendes la estructura del DOM, el siguiente paso es aprender cómo seleccionar elementos específicos para manipularlos usando JavaScript. Seleccionar elementos es uno de los aspectos más fundamentales al trabajar con el DOM, ya que es el primer paso para modificar o interactuar con ellos.

JavaScript proporciona varios métodos nativos para seleccionar elementos del DOM. Estos métodos permiten localizar y acceder a uno o más elementos de una página web según diferentes criterios, como su ID, clase o nombre de etiqueta.

- getElementById: Este es uno de los métodos más comunes y sencillos para seleccionar un elemento del DOM. getElementById permite seleccionar un único elemento basado en su atributo id.

```
var elemento = document.getElementById("miId");
```

- Ventajas: Es rápido y eficiente, ya que los IDs son únicos en el documento.
- Limitación: Solo se puede seleccionar un elemento a la vez, ya que los IDs deben ser únicos.
- getElementsByClassName: Este método selecciona todos los elementos que comparten un mismo valor de la clase (class). Retorna una colección de nodos (una lista de elementos).

```
var elementos = document.getElementsByClassName("miClase");
```

- Ventajas: Es útil para seleccionar múltiples elementos que comparten el mismo estilo o comportamiento.
- Limitación: Retorna una colección de elementos, lo que significa que para manipular un elemento específico dentro de esta colección, debes acceder a él por su índice.
- getElementsByTagName: getElementsByTagName selecciona todos los elementos que coinciden con un nombre de etiqueta específico, como , <p>, , etc.

```
var elementos = document.getElementsByTagName("p");
```

- Ventajas: Es útil para seleccionar todos los elementos de un tipo específico en la página.

- Limitación: Similar a getElementsByClassName, retorna una colección de nodos, por lo que es necesario acceder a los elementos individualmente.
- Selección moderna con Selectores CSS

Con la evolución de JavaScript y la necesidad de métodos de selección más versátiles, se introdujeron métodos que permiten seleccionar elementos utilizando selectores CSS, lo que proporciona un control más granular y flexible.

- querySelector: Este método permite seleccionar el primer elemento que coincide con un selector CSS. Es uno de los métodos más poderosos y flexibles, ya que puedes utilizar cualquier selector CSS válido.

```
var elemento = document.querySelector(".miClase");
```

- Ventajas: Muy flexible, permite seleccionar por ID, clase, atributo, combinaciones complejas de selectores, y más.
- Limitación: Solo selecciona el primer elemento que coincide con el selector.
- querySelectorAll: Similar a querySelector, este método selecciona todos los elementos que coinciden con un selector CSS y retorna una NodeList (una lista de nodos similar a un array).

```
var elementos = document.querySelectorAll(".miClase");
```

- Ventajas: Permite seleccionar múltiples elementos con selectores CSS complejos.
- Limitación: Retorna una NodeList que debe ser recorrida para manipular los elementos individualmente.

5.4.1 Comparación entre métodos de selección

Elegir el método de selección adecuado depende del contexto y de las necesidades específicas de la tarea. Aquí hay algunas consideraciones:

- Uso de IDs (getElementById): Cuando se sabe que un elemento tiene un ID único y se necesita seleccionarlo directamente, getElementById es la opción más rápida y eficiente.
- Selección por clases o etiquetas (getElementsByClassName, getElementsByTagName): Estos métodos son útiles cuando se necesita seleccionar múltiples elementos que comparten la misma clase o el mismo tipo de etiqueta. Sin embargo, es importante recordar que estos métodos retornan colecciones, por lo que se requiere un bucle para manipular cada elemento.

- Uso de selectores CSS (querySelector, querySelectorAll): Estos métodos son extremadamente flexibles y permiten seleccionar elementos con mayor precisión utilizando la sintaxis de selectores CSS. Son ideales cuando se necesitan combinaciones complejas de selectores o cuando no se conoce de antemano la estructura exacta del HTML.

Ejemplos prácticos

Aquí tienes algunos ejemplos de cómo utilizar estos métodos:

- Seleccionar un elemento por su ID:

```
var titulo = document.getElementById("titulo-principal");
titulo.textContent = "Nuevo Título";
```

Seleccionar todos los elementos con una clase específica y cambiar su estilo:

```
var items = document.getElementsByClassName("item-lista");
for (var i = 0; i < items.length; i++) {
  items[i].style.color = "blue";
}
```

Seleccionar todos los párrafos y añadir una clase:

```
var parrafos = document.getElementsByTagName("p");
for (var i = 0; i < parrafos.length; i++) {
  parrafos[i].classList.add("nuevo-estilo");
}
```

Seleccionar el primer elemento con una clase específica utilizando querySelector:

```
var primerItem = document.querySelector(".item-lista");
primerItem.style.fontWeight = "bold";
```

Seleccionar todos los elementos de una clase específica utilizando querySelectorAll y cambiar su texto:

```
var todosItems = document.querySelectorAll(".item-lista");
todosItems.forEach(function(item) {
  item.textContent = "Texto Modificado";
});
```

5.4.2 Ejemplos

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Ejemplo 1: getElementById</title>
</head>
<body>
    <h1 id="titulo-principal">Título Original</h1>
    <button onclick="cambiarTitulo()">Cambiar Título</button>
```

```
    <script>
        function cambiarTitulo() {
            var titulo = document.getElementById("titulo-principal");
            titulo.textContent = "Nuevo Título Modificado";
        }
    </script>
</body>
</html>
```

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Ejemplo 2: getElementsByClassName</title>
</head>
<body>
    <ul>
        <li class="item-lista">Elemento 1</li>
        <li class="item-lista">Elemento 2</li>
        <li class="item-lista">Elemento 3</li>
    </ul>
    <button onclick="cambiarColor()">Cambiar Color</button>

    <script>
        function cambiarColor() {
            var items = document.getElementsByClassName("item-lista");
            for (var i = 0; i < items.length; i++) {
                items[i].style.color = "blue";
            }
        }
    </script>
</body>
</html>
```

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Ejemplo 3: getElementsByTagName</title>
    <style>
        .nuevo-estilo {
            font-style: italic;
            color: green;
        }
    </style>
</head>
<body>
    <p>Este es el primer párrafo.</p>
    <p>Este es el segundo párrafo.</p>
    <p>Este es el tercer párrafo.</p>
    <button onclick="añadirClase()">Añadir Clase</button>

    <script>
        function añadirClase() {
            var parrafos = document.getElementsByTagName("p");
            for (var i = 0; i < parrafos.length; i++) {
                parrafos[i].classList.add("nuevo-estilo");
            }
        }
    </script>
```

```
</body>
</html>
```

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Ejemplo 4: querySelector</title>
</head>
<body>
    <div class="contenedor">
        <h2>Subtítulo 1</h2>
        <h2>Subtítulo 2</h2>
    </div>
    <button onclick="cambiarSubtitulo()">Cambiar Subtítulo</button>

    <script>
        function cambiarSubtitulo() {
            var primerSubtitulo = document.querySelector(".contenedor h2");
            primerSubtitulo.textContent = "Subtítulo Modificado";
        }
    </script>
</body>
</html>
```

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Ejemplo 5: querySelectorAll</title>
</head>
<body>
    <ul>
        <li class="item">Elemento A</li>
        <li class="item">Elemento B</li>
        <li class="item">Elemento C</li>
    </ul>
    <button onclick="modificarElementos()">Modificar Elementos</button>

    <script>
        function modificarElementos() {
            var items = document.querySelectorAll(".item");
            items.forEach(function(item) {
                item.textContent = "Elemento Modificado";
            });
        }
    </script>
</body>
</html>
```

5.5 Manipulación del DOM

Una vez que hayas seleccionado los elementos en el DOM, el siguiente paso es manipularlos según sea necesario. La manipulación del DOM es fundamental para crear páginas web dinámicas e interactivas. Esto incluye cambiar el contenido de los elementos, modificar sus atributos, aplicar estilos, o incluso crear y eliminar nodos en el DOM.

5.5.1 Modificación de contenido de elementos

Modificar el contenido de los elementos es una de las tareas más comunes al trabajar con el DOM. JavaScript proporciona varias formas de hacerlo.

- Propiedad innerHTML: La propiedad innerHTML permite acceder o modificar el contenido HTML de un elemento. Esto incluye no solo el texto, sino también cualquier elemento HTML anidado dentro del elemento seleccionado.

```
var contenedor = document.getElementById("miDiv");
contenedor.innerHTML = "<h2>Nuevo Contenido</h2><p>Este es un párrafo dentro de un div.</p>";
```

- Ventaja: Permite establecer contenido HTML complejo, incluyendo etiquetas y estructuras anidadas.
- Desventaja: Sobrescribe todo el contenido del elemento, lo que puede resultar en pérdida de event listeners asociados a los elementos hijos.
- Propiedad textContent: La propiedad textContent permite establecer o recuperar el contenido de texto de un elemento, sin afectar la estructura HTML anidada. Todo el contenido HTML se tratará como texto plano.

```
var parrafo = document.getElementById("miParrafo");
parrafo.textContent = "Este es el nuevo texto del párrafo.";
```

- Ventaja: Ideal para insertar texto plano, ya que no interpreta las etiquetas HTML.
- Desventaja: No se puede utilizar para insertar contenido HTML, solo texto.

5.5.2 Creación y eliminación de nodos

JavaScript permite la creación de nuevos nodos en el DOM, así como la eliminación de los existentes.

- createElement: El método createElement se utiliza para crear nuevos elementos HTML que luego pueden ser añadidos al DOM.

```
var nuevoElemento = document.createElement("div");
nuevoElemento.textContent = "Este es un nuevo div";
document.body.appendChild(nuevoElemento);
```

- Ventaja: Puedes crear cualquier tipo de elemento HTML de forma dinámica y añadirlo al DOM.
- Uso Común: Crear listas dinámicas, formularios o cualquier otro contenido que necesite generarse en tiempo de ejecución.

- appendChild: Este método permite añadir un nodo creado a otro nodo existente en el DOM como su último hijo.

```
var lista = document.getElementById("miLista");
var nuevoItem = document.createElement("li");
nuevoItem.textContent = "Nuevo ítem de la lista";
lista.appendChild(nuevoItem);
```

- Ventaja: Es fácil añadir nodos al final de la lista de hijos de un elemento.
- removeChild: removeChild se utiliza para eliminar un nodo hijo de un elemento.

```
var lista = document.getElementById("miLista");
var primerItem = lista.getElementsByTagName("li")[0];
lista.removeChild(primerItem);
```

- Ventaja: Permite eliminar elementos específicos del DOM, liberando memoria y recursos.

5.5.3 Modificación de atributos

Puedes modificar los atributos de los elementos en el DOM para cambiar su comportamiento o apariencia.

- setAttribute y getAttribute: Estos métodos permiten establecer o recuperar los atributos de un elemento.

```
var enlace = document.getElementById("miEnlace");
enlace.setAttribute("href", "https://www.nuevo-enlace.com");
var href = enlace.getAttribute("href");
console.log(href); // Muestra "https://www.nuevo-enlace.com"
```

- Ventaja: Puedes manejar cualquier atributo, incluidos los atributos personalizados.

Muchos atributos HTML comunes pueden ser manipulados directamente a través de propiedades del elemento.

```
var div = document.getElementById("miDiv");
div.id = "nuevoId";
div.className = "nuevaClase";
```

- Ventaja: Es más directo y a menudo más legible que setAttribute.

5.5.4 Manipulación de estilos

Puedes cambiar los estilos de un elemento utilizando la propiedad style o mediante la manipulación de clases.

- Modificación mediante style

La propiedad style permite cambiar los estilos CSS directamente en línea en un elemento.

```
var elemento = document.getElementById("miDiv");
elemento.style.backgroundColor = "blue";
elemento.style.color = "white";
```

- Ventaja: Rápido y útil para aplicar cambios de estilo específicos a un solo elemento.
- Desventaja: La manipulación directa del style puede resultar en CSS en línea, lo cual no es ideal para mantener el código CSS separado y limpio.
- Añadir y quitar clases con classList

classList es una propiedad que permite manipular las clases de un elemento de manera eficiente.

```
- add: Añade una o más clases a un elemento.
```

```
var elemento = document.getElementById("miDiv");
elemento.classList.add("activo", "resaltado");
```

```
- remove: Elimina una o más clases de un elemento.
```

```
elemento.classList.remove("resaltado");
```

```
- toggle: Añade la clase si no está presente, o la elimina si ya lo está.
```

```
elemento.classList.toggle("activo");
```

- Ventaja: Mantiene el estilo de tu sitio web organizado, ya que promueve el uso de clases CSS predefinidas.

5.5.5 Ejemplos Completos

Ejemplo 1: Cambiar el contenido HTML de un elemento

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Modificar innerHTML</title>
</head>
<body>
    <div id="miDiv">Contenido Original</div>
    <button onclick="modificarContenido()">Modificar Contenido</button>

    <script>
        function modificarContenido() {
            var div = document.getElementById("miDiv");
            div.innerHTML = "<h2>Nuevo Contenido</h2><p>Contenido modificado utilizando
                innerHTML.</p>";
```

```
        }
    </script>
</body>
</html>
```

Ejemplo 2: Crear y añadir un nuevo nodo

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Crear y añadir nodo</title>
</head>
<body>
    <ul id="miLista">
        <li>Ítem 1</li>
        <li>Ítem 2</li>
    </ul>
    <button onclick="añadirElemento()">Añadir Ítem</button>

    <script>
        function añadirElemento() {
            var lista = document.getElementById("miLista");
            var nuevoItem = document.createElement("li");
            nuevoItem.textContent = "Nuevo Ítem";
            lista.appendChild(nuevoItem);
        }
    </script>
</body>
</html>
```

Ejemplo 3: Modificar atributos de un elemento

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Modificar Atributos</title>
</head>
<body>
    <a id="miEnlace" href="https://www.ejemplo.com">Enlace Original</a>
    <button onclick="modificarAtributo()">Modificar Enlace</button>

    <script>
        function modificarAtributo() {
            var enlace = document.getElementById("miEnlace");
            enlace.setAttribute("href", "https://www.nuevo-enlace.com");
            enlace.textContent = "Enlace Modificado";
        }
    </script>
</body>
</html>
```

Ejemplo 4: Cambiar estilos de un elemento

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

```
    <title>Cambiar Estilos</title>
</head>
<body>
    <div id="miDiv" style="width: 100px; height: 100px; background-color: red;">Div Original<
        /div>
    <button onclick="cambiarEstilo()">Cambiar Estilo</button>

    <script>
        function cambiarEstilo() {
            var div = document.getElementById("miDiv");
            div.style.backgroundColor = "blue";
            div.style.color = "white";
            div.style.padding = "20px";
        }
    </script>
</body>
</html>
```

Ejemplo 5: Manipulación de clases con classList

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Manipulación de Clases</title>
    <style>
        .activo {
            background-color: yellow;
            font-weight: bold;
        }
        .resaltado {
            border: 2px solid black;
        }
    </style>
</head>
<body>
    <div id="miDiv" class="activo">Div con clases</div>
    <button onclick="toggleResaltado()">Toggle Resaltado</button>

    <script>
        function toggleResaltado() {
            var div = document.getElementById("miDiv");
            div.classList.toggle("resaltado");
        }
    </script>
</body>
</html>
```

5.6 Eventos en el DOM

Los eventos son un aspecto fundamental de la programación web interactiva. Permiten que los desarrolladores respondan a las acciones del usuario, como clics, desplazamientos, pulsaciones de teclas y más, para crear una experiencia de usuario dinámica. En este apartado, exploraremos cómo manejar eventos en el DOM utilizando JavaScript.

Un evento es cualquier interacción del usuario o del navegador con la página web que puede ser

capturada y gestionada mediante JavaScript. Estos eventos pueden estar asociados a elementos específicos del DOM, como botones, enlaces, formularios, y se utilizan para desencadenar funciones o acciones.

- Ejemplos de eventos comunes:
  - click: Ocurre cuando un usuario hace clic en un elemento.
  - mouseover: Se activa cuando el puntero del ratón se coloca sobre un elemento.
  - keydown: Se dispara cuando una tecla es presionada.
  - submit: Ocurre cuando un formulario es enviado.
- Propagación de eventos:
  - Burbuja: Los eventos en el DOM "burbujean" hacia arriba desde el elemento que lo disparó hasta el elemento raíz (document), lo que permite capturar eventos en un nivel superior.
  - Captura: La captura es lo contrario de la burbuja; el evento se captura primero en el elemento más externo (como document) y luego se propaga hacia abajo al elemento objetivo.

5.6.1 Asignación de manejadores de eventos

Para responder a eventos, se utilizan los manejadores de eventos, que son funciones que se ejecutan cuando ocurre un evento específico en un elemento del DOM.

- addEventListener: El método addEventListener es la forma más versátil y recomendada para asignar manejadores de eventos en JavaScript. Permite registrar múltiples manejadores para el mismo evento en el mismo elemento y ofrece control sobre la fase de captura o burbuja del evento.

```
var boton = document.getElementById("miBoton");
boton.addEventListener("click", function() {
    alert("¡Botón clicado!");
});
```

- Ventaja: Puedes añadir múltiples eventos a un mismo elemento y controlar la fase de propagación.
- Uso Común: Ideal para casi todas las situaciones en las que se necesita gestionar eventos.
- Remover eventos con removeEventListener: removeEventListener permite eliminar un manejador de eventos previamente añadido con addEventListener. Es importante que la función que se pasa a removeEventListener sea la misma que la utilizada en addEventListener.

```
function manejarClick() {
    alert("¡Botón clicado!");
}

var boton = document.getElementById("miBoton");
boton.addEventListener("click", manejarClick);

// Más tarde en el código
boton.removeEventListener("click", manejarClick);
```

- Ventaja: Es útil cuando necesitas gestionar eventos temporalmente, o limpiar eventos para mejorar el rendimiento.

5.6.2 Eventos comunes

Exploraremos algunos eventos comunes y cómo manejarlos.

- Eventos de ratón (click, mouseover, mouseout): Los eventos de ratón son algunos de los más utilizados en la interacción del usuario.

```
var boton = document.getElementById("miBoton");
boton.addEventListener("click", function() {
    alert("¡Hiciste clic en el botón!");
});

boton.addEventListener("mouseover", function() {
    boton.style.backgroundColor = "yellow";
});

boton.addEventListener("mouseout", function() {
    boton.style.backgroundColor = "";
});
```

```
- click: Se dispara cuando un usuario hace clic en un elemento.
- mouseover y mouseout: Se disparan cuando el ratón entra o sale del área de un elemento.
```

- Eventos de teclado (keydown, keyup)

Los eventos de teclado permiten reaccionar a la entrada del usuario a través del teclado.

```
document.addEventListener("keydown", function(event) {
    console.log("Tecla presionada: " + event.key);
});

document.addEventListener("keyup", function(event) {
    console.log("Tecla liberada: " + event.key);
});
```

- keydown: Se dispara cuando se presiona una tecla.
- keyup: Se dispara cuando se libera una tecla.
- Eventos de formulario (submit, change)

Los formularios en HTML generan varios eventos útiles para la validación y el manejo de datos.

```
var formulario = document.getElementById("miFormulario");

formulario.addEventListener("submit", function(event) {
    event.preventDefault(); // Previene el envío del formulario
    alert("Formulario enviado");
});

var campoTexto = document.getElementById("miCampo");

campoTexto.addEventListener("change", function() {
    console.log("El valor ha cambiado a: " + campoTexto.value);
});
```

- submit: Se dispara cuando un formulario es enviado.
- change: Se dispara cuando el valor de un campo de formulario cambia.

5.6.3 Ejemplos Completos

Ejemplo 1: Manejo de evento click

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Evento Click</title>
</head>
<body>
    <button id="miBoton">Haz clic aquí</button>

    <script>
        var boton = document.getElementById("miBoton");
        boton.addEventListener("click", function() {
            alert("¡Botón clicado!");
        });
    </script>
</body>
</html>
```

Ejemplo 2: Eventos de ratón (mouseover, mouseout)

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Eventos de Ratón</title>
</head>
<body>
    <button id="miBoton">Pasa el ratón aquí</button>

    <script>
        var boton = document.getElementById("miBoton");

        boton.addEventListener("mouseover", function() {
            boton.style.backgroundColor = "yellow";
```

```
        });

        boton.addEventListener("mouseout", function() {
            boton.style.backgroundColor = "";
        });
    </script>
</body>
</html>
```

Ejemplo 3: Manejo de eventos de teclado (keydown, keyup)

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Eventos de Teclado</title>
</head>
<body>
    <input type="text" id="miCampo" placeholder="Escribe algo...">

    <script>
        var campoTexto = document.getElementById("miCampo");

        document.addEventListener("keydown", function(event) {
            console.log("Tecla presionada: " + event.key);
        });

        document.addEventListener("keyup", function(event) {
            console.log("Tecla liberada: " + event.key);
        });
    </script>
</body>
</html>
```

Ejemplo 4: Manejo de eventos de formulario (submit, change)

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Eventos de Formulario</title>
</head>
<body>
    <form id="miFormulario">
        <input type="text" id="miCampo" placeholder="Escribe tu nombre...">
        <button type="submit">Enviar</button>
    </form>

    <script>
        var formulario = document.getElementById("miFormulario");

        formulario.addEventListener("submit", function(event) {
            event.preventDefault(); // Previene el envío del formulario
            alert("Formulario enviado");
        });

        var campoTexto = document.getElementById("miCampo");

        campoTexto.addEventListener("change", function() {
            console.log("El valor ha cambiado a: " + campoTexto.value);
        });
```

```
    </script>
</body>
</html>
```

5.7 Recorrido y Manipulación Avanzada del DOM

Hasta ahora hemos cubierto la selección básica y la manipulación de elementos del DOM. En este apartado, exploraremos técnicas más avanzadas para navegar y manipular el DOM de manera eficiente. Esto incluye cómo recorrer los nodos del DOM, cómo clonar elementos, y cómo usar DocumentFragment para realizar manipulaciones más rápidas y optimizadas.

El DOM permite navegar entre los nodos de una manera jerárquica, accediendo a elementos padres, hijos y hermanos. Esto es útil cuando necesitas manipular o analizar el contenido del DOM de manera más precisa.

- parentNode, childNodes
  - parentNode: Este atributo devuelve el nodo padre del nodo actual. Es útil cuando necesitas moverte hacia arriba en la jerarquía del DOM.

```
var elemento = document.getElementById("miElemento");
var padre = elemento.parentNode;
padre.style.backgroundColor = "lightblue";
```

```
- childNodes: Este atributo devuelve una colección de todos los nodos hijos de un elemento,
    incluyendo nodos de texto y comentarios. Es importante recordar que childNodes devuelve
    todos los tipos de nodos, no solo elementos.
```

```
var lista = document.getElementById("miLista");
var hijos = lista.childNodes;

for (var i = 0; i < hijos.length; i++) {
    if (hijos[i].nodeType === 1) { // nodeType === 1 significa que es un elemento
        hijos[i].style.color = "red";
    }
}
```

- Uso Común: Estos métodos son útiles para recorrer el DOM y manipular elementos que están en una relación específica, como seleccionar todos los hijos de un contenedor y aplicarles un estilo particular.
- nextSibling, previousSibling
  - nextSibling: Este atributo devuelve el nodo inmediatamente siguiente al nodo actual en la lista de hijos del padre.

```
var primerItem = document.getElementById("primerItem");
var siguienteItem = primerItem.nextSibling;
siguienteItem.style.fontWeight = "bold";
```

- previousSibling: Este atributo devuelve el nodo inmediatamente anterior al nodo actual en la lista de hijos del padre.

```
var segundoItem = document.getElementById("segundoItem");
var anteriorItem = segundoItem.previousSibling;
anteriorItem.style.fontWeight = "bold";
```

- Uso Común: Estos atributos son útiles para moverte hacia adelante o hacia atrás entre nodos hermanos, especialmente en situaciones donde los elementos no tienen una clase o ID única.

5.7.1 Clonar nodos

En situaciones donde necesitas duplicar un elemento existente en el DOM, puedes usar el método cloneNode.

- cloneNode: El método cloneNode permite crear una copia exacta de un nodo existente en el DOM. Este método puede realizar una clonación superficial (solo clona el nodo en sí mismo) o una clonación profunda (clona el nodo y toda su descendencia).

Clonación superficial:

```
var original = document.getElementById("miElemento");
var clon = original.cloneNode(false);
document.body.appendChild(clon);
```

Clonación profunda:

```
var original = document.getElementById("miElemento");
var clon = original.cloneNode(true); // Clona todo, incluyendo nodos hijos
document.body.appendChild(clon);
```

- Uso Común: cloneNode es útil cuando necesitas replicar estructuras del DOM de manera dinámica, como duplicar filas en una tabla o crear múltiples copias de un componente de interfaz de usuario.

Clonación profunda vs. superficial - Superficial: Solo se clona el elemento en sí, sin su contenido o hijos. Esto es útil cuando solo necesitas replicar la estructura básica del nodo, pero no sus contenidos internos. - Profunda: Clona todo el contenido, incluidos todos los elementos hijos. Es más común en situaciones donde necesitas duplicar todo el contenido de un elemento.

5.7.2 DocumentFragment: Manipulación eficiente del DOM

Cuando necesitas añadir varios elementos al DOM de manera eficiente, es recomendable utilizar un DocumentFragment. Este es un nodo especial que actúa como un contenedor temporal para los elementos, permitiendo realizar manipulaciones sin afectar inmediatamente el DOM.

Un DocumentFragment es un contenedor ligero que no forma parte del DOM activo. Los elementos añadidos a un DocumentFragment no desencadenan reflujo ni repintado hasta que el fragmento es añadido al DOM, lo que lo hace mucho más eficiente para operaciones que implican múltiples cambios en la estructura del DOM.

```
var fragmento = document.createDocumentFragment();

for (var i = 0; i < 5; i++) {
    var nuevoElemento = document.createElement("li");
    nuevoElemento.textContent = "Ítem " + (i + 1);
    fragmento.appendChild(nuevoElemento);
}

var lista = document.getElementById("miLista");
lista.appendChild(fragmento);
```

- Ventaja: DocumentFragment permite realizar múltiples manipulaciones en memoria antes de hacer una sola inserción en el DOM, lo que mejora el rendimiento.
- Uso Común: Ideal para añadir grandes cantidades de elementos al DOM, como cuando se genera contenido dinámico basado en datos de una API.

5.7.3 Ejemplos Completos

Ejemplo 1: Navegación por nodos con parentNode y childNodes

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Navegación por Nodos</title>
</head>
<body>
    <ul id="miLista">
        <li>Ítem 1</li>
        <li>Ítem 2</li>
        <li>Ítem 3</li>
    </ul>
    <button onclick="resaltarPadre()">Resaltar Padre</button>
    <button onclick="resaltarHijos()">Resaltar Hijos</button>

    <script>
        function resaltarPadre() {
            var primerItem = document.getElementById("miLista").childNodes[1]; // El primer
                hijo real
            var padre = primerItem.parentNode;
            padre.style.backgroundColor = "lightgreen";
        }

        function resaltarHijos() {
            var lista = document.getElementById("miLista");
            var hijos = lista.childNodes;
            for (var i = 0; i < hijos.length; i++) {
                if (hijos[i].nodeType === 1) { // nodeType 1 asegura que sea un elemento
```

```
                    hijos[i].style.color = "red";
                }
            }
        }
    </script>
</body>
</html>
```

Ejemplo 2: Clonación de nodos

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Clonación de Nodos</title>
</head>
<body>
    <div id="miElemento">
        <p>Este es un párrafo dentro de un div.</p>
    </div>
    <button onclick="clonarElemento()">Clonar Elemento</button>

    <script>
        function clonarElemento() {
            var original = document.getElementById("miElemento");
            var clon = original.cloneNode(true); // Clonación profunda
            document.body.appendChild(clon);
        }
    </script>
</body>
</html>
```

Ejemplo 3: Uso de DocumentFragment para añadir múltiples elementos

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>DocumentFragment</title>
</head>
<body>
    <ul id="miLista"></ul>
    <button onclick="añadirElementos()">Añadir Elementos</button>

    <script>
        function añadirElementos() {
            var fragmento = document.createDocumentFragment();

            for (var i = 0; i < 5; i++) {
                var nuevoElemento = document.createElement("li");
                nuevoElemento.textContent = "Ítem " + (i + 1);
                fragmento.appendChild(nuevoElemento);
            }

            var lista = document.getElementById("miLista");
            lista.appendChild(fragmento);
        }
    </script>
</body>
</html>
```

5.8 Interacción entre el DOM y el BOM (Browser Object Model)

El DOM (Document Object Model) y el BOM (Browser Object Model) son dos conceptos fundamentales en la programación web, y aunque están relacionados, cumplen funciones diferentes. El DOM se encarga de la estructura y el contenido del documento HTML, mientras que el BOM maneja la interacción con el navegador en sí, incluyendo ventanas, historial, y la barra de direcciones. En este apartado, exploraremos cómo el DOM y el BOM interactúan y cómo puedes utilizar el BOM para manipular aspectos del navegador a través de JavaScript.

El BOM es una interfaz que permite a los desarrolladores interactuar con el navegador web. A diferencia del DOM, que se centra en el contenido del documento, el BOM proporciona métodos y propiedades para manejar elementos como las ventanas del navegador, el historial de navegación, la URL actual, y mucho más.

- Componentes principales del BOM:
  - window: El objeto global que representa la ventana del navegador. Casi todos los métodos y propiedades del BOM están disponibles a través de window.
  - navigator: Proporciona información sobre el navegador del usuario, como el nombre, la versión, el idioma, y el estado de la conexión.
  - screen: Ofrece información sobre la pantalla del dispositivo, incluyendo la resolución y el tamaño disponible.
  - location: Permite obtener y modificar la URL actual del documento.
  - history: Permite la manipulación del historial de navegación del navegador.

5.8.1 Relación entre DOM y BOM

El DOM y el BOM están íntimamente relacionados porque ambos son accesibles a través del objeto window. Aunque se ocupan de diferentes aspectos, a menudo se utilizan juntos para crear experiencias web completas e interactivas.

El objeto window como vínculo:

Acceso al DOM: Puedes acceder al DOM a través de window.document.

```
var titulo = window.document.getElementById("titulo");
```

Acceso al BOM: Puedes acceder a elementos del BOM como location, navigator, screen, y history directamente desde window.

```
console.log(window.location.href);
```

- Manipulación del DOM utilizando el BOM: A través del BOM, puedes realizar cambios en la ventana del navegador que afecten directamente al DOM, como redireccionar la página o manipular el historial.

5.8.2 Ejemplos de interacción (manipulación de ventanas, historial del navegador)

A continuación, exploraremos algunos ejemplos de cómo puedes utilizar el BOM para interactuar con el navegador y, por extensión, con el DOM.

```
Manipulación de la ventana del navegador (window)

    Abrir una nueva ventana:
```

```
var nuevaVentana = window.open("https://www.google.com", "_blank", "width=800,height=600");
```

Cerrar una ventana:

```
nuevaVentana.close();
```

Redimensionar la ventana actual:

```
window.resizeTo(1024, 768);
```

Mover la ventana actual:

```
window.moveTo(100, 100);
```

Desplazamiento en la ventana:

```
window.scrollTo(0, 500); // Desplaza la ventana 500 píxeles hacia abajo
```

Manipulación de la URL y redireccionamiento (location)

```
Obtener la URL actual:
```

```
console.log(window.location.href); // Imprime la URL actual
```

Redireccionar a una nueva URL:

```
window.location.href = "https://www.example.com";
```

Recargar la página:

```
window.location.reload(); // Recarga la página actual
```

Manipulación del historial del navegador (history)

Ir hacia adelante o atrás en el historial:

```
window.history.back(); // Va una página hacia atrás en el historial
window.history.forward(); // Va una página hacia adelante en el historial
```

Navegar a un punto específico en el historial:

```
window.history.go(-2); // Va dos páginas atrás en el historial
```

Añadir un nuevo estado al historial:

```
window.history.pushState({page: 1}, "title 1", "?page=1");
```

Obtener información sobre el navegador (navigator)

Información sobre el navegador:

```
console.log(window.navigator.userAgent); // Información sobre el navegador del usuario
```

Comprobar si el navegador está en línea:

```
if (navigator.onLine) {
    console.log("Estás conectado a Internet.");
} else {
    console.log("Estás desconectado de Internet.");
}
```

Detectar la plataforma del usuario:

```
console.log(navigator.platform); // Imprime la plataforma del sistema operativo
```

Información sobre la pantalla (screen)

Obtener la resolución de la pantalla:

```
console.log("Resolución de la pantalla: " + screen.width + "x" + screen.height);
```

Obtener la profundidad de color de la pantalla:

```
console.log("Profundidad de color: " + screen.colorDepth);
```

5.8.3 Ejemplos Completos

Ejemplo 1: Redireccionamiento y manipulación de la URL

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Manipulación de la URL</title>
</head>
<body>
    <button onclick="redireccionar()">Ir a Google</button>
    <button onclick="recargarPagina()">Recargar Página</button>

    <script>
        function redireccionar() {
            window.location.href = "https://www.google.com";
        }
```

```
        function recargarPagina() {
            window.location.reload();
        }
    </script>
</body>
</html>
```

Ejemplo 2: Manipulación del historial del navegador

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Manipulación del Historial</title>
</head>
<body>
    <button onclick="irAtras()">Ir Atrás</button>
    <button onclick="irAdelante()">Ir Adelante</button>
    <button onclick="irAPagina()">Ir a Página Específica</button>

    <script>
        function irAtras() {
            window.history.back();
        }

        function irAdelante() {
            window.history.forward();
        }

        function irAPagina() {
            window.history.go(-2); // Navega dos páginas atrás
        }
    </script>
</body>
</html>
```

Ejemplo 3: Obtener información del navegador y la pantalla

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Información del Navegador y Pantalla</title>
</head>
<body>
    <h1>Información del Navegador</h1>
    <p id="infoNavegador"></p>
    <p id="infoPantalla"></p>

    <script>
        var infoNavegador = "Navegador: " + window.navigator.userAgent + "<br>" +
                            "Plataforma: " + window.navigator.platform + "<br>" +
                            "Conectado a Internet: " + (navigator.onLine ? "Sí" : "No");
        document.getElementById("infoNavegador").innerHTML = infoNavegador;

        var infoPantalla = "Resolución de la pantalla: " + screen.width + "x" + screen.height
             + "<br>" +
                           "Profundidad de color: " + screen.colorDepth;
        document.getElementById("infoPantalla").innerHTML = infoPantalla;
    </script>
```

```
</body>
</html>
```

6 Otras APIS

Las APIs (Application Programming Interfaces) del navegador son un conjunto de herramientas y funcionalidades que los navegadores web ofrecen a los desarrolladores para interactuar y manipular varios aspectos del entorno del navegador y la página web. Estas APIs permiten acceder y modificar el contenido, la estructura y el comportamiento de una página web de manera programática usando JavaScript.

Las APIs del navegador no son parte del lenguaje JavaScript en sí, sino que son extensiones proporcionadas por el entorno de ejecución (el navegador) que permiten que el código JavaScript realice acciones que de otro modo no serían posibles, como manipular el contenido del DOM, realizar solicitudes HTTP, almacenar datos en el dispositivo del usuario, acceder a la cámara, y mucho más.

6.1 APIs de Almacenamiento

6.1.1 LocalStorage

LocalStorage es una API del navegador que permite almacenar datos de manera persistente en el dispositivo del usuario. A diferencia de las cookies, los datos almacenados en localStorage no se envían con cada solicitud HTTP, lo que mejora el rendimiento. Además, localStorage permite almacenar hasta 5 MB de datos, lo que es significativamente más que los 4 KB típicos de las cookies.

Las características principales de localStorage son:

- Persistencia: Los datos se almacenan indefinidamente hasta que el usuario los elimine o se borren desde el código JavaScript.
- Clave-valor: Los datos se almacenan en pares clave-valor. Tanto las claves como los valores son cadenas de texto.
- Accesible desde el mismo origen: Los datos en localStorage solo pueden ser accedidos por páginas del mismo origen (misma URL, protocolo y puerto).

Guardar un valor en localStorage:

```
localStorage.setItem("nombreUsuario", "Juan");
```

Recuperar un valor de localStorage:

```
const nombreUsuario = localStorage.getItem("nombreUsuario");
console.log(nombreUsuario);  // "Juan"
```

Eliminar un valor de localStorage:

```
localStorage.removeItem("nombreUsuario");
```

Eliminar todos los valores almacenados:

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

6.1.2 SessionStorage

SessionStorage es similar a localStorage en su funcionalidad, pero tiene una diferencia clave en su persistencia: los datos almacenados en sessionStorage solo persisten durante la sesión de la página. Esto significa que los datos se eliminan automáticamente cuando la pestaña o la ventana del navegador se cierra.

Características principales de sessionStorage:

- Temporalidad: Los datos se mantienen solo durante la sesión de navegación. Si el usuario recarga la página, los datos permanecen; sin embargo, si cierra la pestaña o el navegador, los datos se eliminan.
- Mismo modelo de clave-valor: Funciona igual que localStorage en términos de almacenamiento y recuperación de datos.

Guardar un valor en sessionStorage:

```
sessionStorage.setItem("estadoSesion", "activo");
```

Recuperar un valor de sessionStorage:

```
const estadoSesion = sessionStorage.getItem("estadoSesion");
console.log(estadoSesion);  // "activo"
```

Eliminar un valor de sessionStorage:

```
sessionStorage.removeItem("estadoSesion");
```

Eliminar todos los valores almacenados:

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

6.1.3 IndexedDB

IndexedDB es una API de almacenamiento mucho más poderosa que localStorage y sessionStorage. Está diseñada para almacenar grandes cantidades de datos estructurados, lo que la hace adecuada

para aplicaciones web complejas como bases de datos locales, aplicaciones de gestión de datos, y más.

Características principales de IndexedDB:

- Almacenamiento estructurado: Permite almacenar datos en una base de datos con transacciones, índices, y consultas.
- Soporte para tipos de datos complejos: A diferencia de localStorage, que solo maneja cadenas de texto, IndexedDB puede almacenar tipos de datos complejos como objetos JavaScript, archivos binarios, y más.
- Asíncrono: Las operaciones en IndexedDB son asíncronas, lo que significa que no bloquean la ejecución del código mientras se realizan.

Para trabajar con IndexedDB, necesitas seguir una serie de pasos:

- Abrir una base de datos:

```
const request = indexedDB.open("miBaseDeDatos", 1);

request.onupgradeneeded = function(evento) {
    const db = evento.target.result;
    const store = db.createObjectStore("miObjectStore", { keyPath: "id" });
};
```

Agregar datos a la base de datos:

```
const db = request.result;
const transaccion = db.transaction("miObjectStore", "readwrite");
const store = transaccion.objectStore("miObjectStore");

store.add({ id: 1, nombre: "Juan", edad: 30 });
```

Recuperar datos de la base de datos:

```
const transaccion = db.transaction("miObjectStore", "readonly");
const store = transaccion.objectStore("miObjectStore");
const requestGet = store.get(1);

requestGet.onsuccess = function() {
    console.log(requestGet.result);  // { id: 1, nombre: "Juan", edad: 30 }
};
```

Actualizar datos en la base de datos:

```
const transaccion = db.transaction("miObjectStore", "readwrite");
const store = transaccion.objectStore("miObjectStore");

const data = { id: 1, nombre: "Juan", edad: 31 };  // Actualizando la edad
store.put(data);
```

Eliminar datos de la base de datos:

```
const transaccion = db.transaction("miObjectStore", "readwrite");
```

```
const store = transaccion.objectStore("miObjectStore");

store.delete(1);
```

Ejemplos Prácticos

Ejemplo de uso:

Supongamos que estás creando una aplicación de gestión de tareas. Podrías utilizar IndexedDB para almacenar las tareas localmente:

```
const request = indexedDB.open("TareasDB", 1);

request.onupgradeneeded = function(evento) {
    const db = evento.target.result;
    const store = db.createObjectStore("tareas", { keyPath: "id", autoIncrement: true });
    store.createIndex("porFecha", "fecha");
};

request.onsuccess = function(evento) {
    const db = evento.target.result;

    const transaccion = db.transaction("tareas", "readwrite");
    const store = transaccion.objectStore("tareas");

    store.add({ titulo: "Aprender IndexedDB", fecha: new Date() });
};

request.onerror = function(evento) {
    console.error("Error al abrir IndexedDB:", evento.target.errorCode);
};
```

En este ejemplo, estamos creando una base de datos llamada “TareasDB” con un almacén de objetos (object store) llamado “tareas”. Luego, se añade una nueva tarea a la base de datos.

6.1.3.1 Ejemplo IndexedDB

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Aplicación de Tareas con IndexedDB</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            background-color: #f2f2f2;
            margin: 0;
            padding: 20px;
        }

        h1 {
            text-align: center;
        }

        .container {
            max-width: 600px;
            margin: 0 auto;
            background-color: #fff;
            padding: 30px;
```

```
            box-shadow: 0 2px 8px rgba(0,0,0,0.2);
            border-radius: 8px;
        }

        .input-group {
            display: flex;
            margin-bottom: 20px;
        }

        .input-group input {
            flex: 1;
            padding: 10px;
            font-size: 16px;
            border: 1px solid #ccc;
            border-radius: 4px;
        }

        .input-group button {
            padding: 10px 20px;
            margin-left: 10px;
            font-size: 16px;
            background-color: #28a745;
            color: #fff;
            border: none;
            border-radius: 4px;
            cursor: pointer;
        }

        .input-group button:hover {
            background-color: #218838;
        }

        .task-list {
            list-style: none;
            padding: 0;
        }

        .task-item {
            display: flex;
            justify-content: space-between;
            align-items: center;
            background-color: #fafafa;
            padding: 15px;
            margin-bottom: 10px;
            border: 1px solid #ddd;
            border-radius: 4px;
        }

        .task-item span {
            flex: 1;
            margin-right: 10px;
            word-break: break-word;
        }

        .task-item button {
            padding: 5px 10px;
            font-size: 14px;
            border: none;
            border-radius: 4px;
            cursor: pointer;
            color: #fff;
        }

        .edit-button {
```

```
            background-color: #007bff;
            margin-right: 5px;
        }

        .edit-button:hover {
            background-color: #0069d9;
        }

        .delete-button {
            background-color: #dc3545;
        }

        .delete-button:hover {
            background-color: #c82333;
        }

        .no-tasks {
            text-align: center;
            color: #777;
            margin-top: 20px;
        }
    </style>
</head>
<body>

    <div class="container">
        <h1>Aplicación de Tareas</h1>
        <div class="input-group">
            <input type="text" id="taskInput" placeholder="Escribe una nueva tarea...">
            <button id="addTaskButton">Agregar Tarea</button>
        </div>
        <ul id="taskList" class="task-list"></ul>
        <p id="noTasksMessage" class="no-tasks">No hay tareas pendientes.</p>
    </div>

    <script>
        // Verificamos si el navegador soporta IndexedDB
        if (!window.indexedDB) {
            alert("Tu navegador no soporta una versión estable de IndexedDB. Algunas caracter
                ísticas no estarán disponibles.");
        }

        const dbName = "tasksDB";
        let db;

        // Abriendo o creando la base de datos
        const request = indexedDB.open(dbName, 1);

        request.onerror = (event) => {
            console.error("Error al abrir la base de datos:", event.target.errorCode);
        };

        request.onsuccess = (event) => {
            db = event.target.result;
            console.log("Base de datos abierta con éxito");
            displayTasks();
        };

        request.onupgradeneeded = (event) => {
            db = event.target.result;
            const objectStore = db.createObjectStore("tasks", { keyPath: "id", autoIncrement:
                 true });
            objectStore.createIndex("task", "task", { unique: false });
            console.log("Object store creado o actualizado");
```

```
        };

        // Referencias a elementos del DOM
        const taskInput = document.getElementById("taskInput");
        const addTaskButton = document.getElementById("addTaskButton");
        const taskList = document.getElementById("taskList");
        const noTasksMessage = document.getElementById("noTasksMessage");

        // Event Listener para agregar una nueva tarea
        addTaskButton.addEventListener("click", addTask);

        // Función para agregar una nueva tarea
        function addTask() {
            const task = taskInput.value.trim();

            if (task === "") {
                alert("Por favor, escribe una tarea.");
                return;
            }

            const transaction = db.transaction(["tasks"], "readwrite");
            const objectStore = transaction.objectStore("tasks");

            const request = objectStore.add({ task });

            request.onsuccess = () => {
                console.log("Tarea añadida con éxito");
                taskInput.value = "";
                displayTasks();
            };

            request.onerror = (event) => {
                console.error("Error al añadir la tarea:", event.target.error);
            };
        }

        // Función para mostrar todas las tareas
        function displayTasks() {
            taskList.innerHTML = "";

            const transaction = db.transaction(["tasks"], "readonly");
            const objectStore = transaction.objectStore("tasks");

            const request = objectStore.openCursor();
            let hasTasks = false;

            request.onsuccess = (event) => {
                const cursor = event.target.result;

                if (cursor) {
                    hasTasks = true;
                    const listItem = document.createElement("li");
                    listItem.classList.add("task-item");

                    const taskSpan = document.createElement("span");
                    taskSpan.textContent = cursor.value.task;

                    const editButton = document.createElement("button");
                    editButton.textContent = "Editar";
                    editButton.classList.add("edit-button");
                    editButton.onclick = () => editTask(cursor.value);

                    const deleteButton = document.createElement("button");
                    deleteButton.textContent = "Eliminar";
```

```
                    deleteButton.classList.add("delete-button");
                    deleteButton.onclick = () => deleteTask(cursor.value.id);

                    listItem.appendChild(taskSpan);
                    listItem.appendChild(editButton);
                    listItem.appendChild(deleteButton);

                    taskList.appendChild(listItem);

                    cursor.continue();
                } else {
                    noTasksMessage.style.display = hasTasks ? "none" : "block";
                }
            };

            request.onerror = (event) => {
                console.error("Error al mostrar las tareas:", event.target.error);
            };
        }

        // Función para editar una tarea
        function editTask(task) {
            const newTask = prompt("Editar tarea:", task.task);

            if (newTask !== null) {
                const transaction = db.transaction(["tasks"], "readwrite");
                const objectStore = transaction.objectStore("tasks");

                const request = objectStore.put({ id: task.id, task: newTask.trim() });

                request.onsuccess = () => {
                    console.log("Tarea actualizada con éxito");
                    displayTasks();
                };

                request.onerror = (event) => {
                    console.error("Error al actualizar la tarea:", event.target.error);
                };
            }
        }

        // Función para eliminar una tarea
        function deleteTask(id) {
            const confirmation = confirm("¿Estás seguro de que deseas eliminar esta tarea?");

            if (confirmation) {
                const transaction = db.transaction(["tasks"], "readwrite");
                const objectStore = transaction.objectStore("tasks");

                const request = objectStore.delete(id);

                request.onsuccess = () => {
                    console.log("Tarea eliminada con éxito");
                    displayTasks();
                };

                request.onerror = (event) => {
                    console.error("Error al eliminar la tarea:", event.target.error);
                };
            }
        }
    </script>
</body>
</html>
```

6.2 APIs

Existe una gran variedad de APIs implementadas en los navegadores webs. Algunas de ellas son:

- API de geolocalización
- APIs Multimedia
- API de notificaciones
- API de Service Workers y PWA (Progressive Web Apps)
- API de Websockets
- API Drag and Drop
- API de Ficheros
- API de clipboard
- API de Web Animations
- API de Web Bluetooth

7 Ejercicios VI sobre módulos ES6

1. Crea un archivo llamado mathOperations.js que exporte dos funciones: `sum(a, b)` y `multiply(a, b)` para sumar y multiplicar dos números. Luego, crea un archivo llamado `main.js` que importe estas funciones y las utilice para calcular la suma de 5 y 3, y el producto de 4 y 7.
2. Crea un archivo llamado `greeting.js` que exporte una función por defecto llamada `greet(name)` que devuelva un saludo como "Hola, [nombre]". Luego, crea un archivo llamado `app.js` que importe esta función y la utilice para saludar a "Carlos".
3. Crea un archivo llamado `mathTools.js` que exporte una función por defecto llamada `calculate(operation, a, b)` que acepte tres parámetros: una operación ("sum", "multiply", etc.) y dos números. Dependiendo del valor de operation, deberá llamar a las funciones `sum(a, b)`, `subtract(a, b)`, o `multiply(a, b)`, que también deben ser exportadas de manera normal (no por defecto). En otro archivo `app.js`, importa la función por defecto y las funciones normales, y realiza varias operaciones como suma, resta y multiplicación.
4. Crea dos archivos, `mathOperations.js` y `app.js`. En `mathOperations.js`, exporta funciones llamadas `add(a, b)`, `subtract(a, b)` y `multiply(a, b)`. En `app.js`, crea una función `loadOperations()` que importe dinámicamente `mathOperations.js` y, al resolverse la promesa, use las funciones para realizar varias operaciones matemáticas. Luego, usa la función en `app.js` para importar el módulo dinámicamente y mostrar los resultados.

8 Ejercicios VII sobre Asincronía

1. Implementa una función que simule la consulta a una base de datos para obtener información de un usuario por su ID. Usa setTimeout para simular la demora de la operación y utiliza un callback para devolver el resultado.
2. Toma la función del ejercicio anterior y reescríbela utilizando Promesas en lugar de callbacks.
3. Usando la función del ejercicio anterior, obtén un usuario y luego realiza otra operación asincrónica basada en los datos del usuario. Por ejemplo, obtener los pedidos de un usuario después de obtener su información.
4. Reescribe el ejercicio anterior usando async/await en lugar de then/catch.
5. Escribe una función que haga una petición a una API pública utilizando fetch y maneje la respuesta. Por ejemplo, obtén los datos de un usuario de la API de GitHub.
6. Escribe una función que realice múltiples peticiones HTTP usando fetch para obtener datos de varios usuarios de GitHub a la vez. Usa Promise.all para esperar a que todas las peticiones se completen.
7. Escribe una función que intente hacer una petición con fetch, pero que reintente la operación hasta 3 veces si falla. Implementa un pequeño retardo entre cada intento.

9 Ejercicios VIII sobre asincronía y DOM

1. Crea una función arrow asíncrona con el parámetro edad, mediante async/await, y carga los datos de https://randomuser.me/api/?results=100, después mediante funciones de los arrays:
    a. Filtra sólo las mujeres.
    b. Genera un array nuevo con los campos username, full_name, email, age.
    c. Busca al primer usuario, mujer, que tenga menos edad que el parámetro dado y retorna el dato. El parámetro edad tiene valor por defecto 18.
    d. Llama a la función en el ámbito global e imprime el resultado en consola.
    e. Si ocurre un error imprimelo en consola.
2. Implementa los siguientes apartados.
    a. Crea un objeto Javascript con las propiedades dni, nombre, apellidos y edad con valores inventados. Serializa el objeto a json y después deserializalo.
    b. Recorre el objeto con un bucle imprimiendo sus propiedades y valores.
    c. Crea una función que desestructure la propiedad nombre y dni y los imprima en consola.

   d. Crea una función arrow con el parámetro tam que genere un array de números aleatorios de 0 a 100 de tamaño tam. Llama a la función y desestructura el primer y segundo elemento y el resto en constantes

3. Crea una página web para escribir la carta a los reyes magos.

   a. El usuario puede escribir el regalo que desea pedir en una caja de texto. Cuando pulsa el botón “Pedir” se añade a la lista.
   b. Crea un botón para borrar la lista completa.
   c. Muestra un mensaje que indique el total de regalos pedidos en la lista.
   d. Cada elemento de la lista tiene un botón o enlace que al pulsarlo lo eliminará

4. Implementa una galería de imágenes:

   a. Genera con un bucle un array de 20 urls con el siguiente patrón: const imagenes = [“https://randomfox.ca/images/1.jpg”, “https://randomfox.ca/images/2.jpg”, “https://randomfox.ca/images/3.jpg”,]
   b. El layout está formado por 4 elementos en el siguiente orden: Un botón Anterior, 2 imágenes y un botón Siguiente.
   c. En un primer momento se mostrará la imagen de la posición 0 y 1 del array. Al pulsar el botón siguiente se mostrarán las imágenes 1 y 2 y así sucesivamente.
   d. De igual forma implementa el comportamiento del botón Anterior.
   e. Controla el botón anterior y siguiente en las posiciones extremas.

5. Implementa un programa Javascript que muestre los posts de un foro mediante promesas:

- Listado de posts: https://jsonplaceholder.typicode.com/posts
- Listado de usuarios: https://jsonplaceholder.typicode.com/users/1, siendo 1 el id del usuario

   a. Genera dinámicamente el listado de posts. Se mostrará el title y body.
   b. Para cada post consulta los datos del usuario que lo ha escrito. Debajo de cada post se mostrará el nombre y correo del usuario correspondiente.
   c. Usa CSS para dar formato.
   d. Añade una caja de texto a modo de buscador en la parte superior. Cuando está vacía se muestran todos los posts. Si el usuario escribe la palabra “aut” se filtrará el listado mostrando aquellos posts que contienen dicha palabra en el campo título.

6. Crea una aplicación simple SPA (Single Page Application) que muestre al usuario varias páginas al hacer click en distintos enlaces. Cada “página” debe ser cargada dinámicamente sin recargar la página completa.

   a. Crea un menú de enlaces que siempre será visible: Home, DAM, Empresas. El atributo src tiene el valor =”#”. Crea un manejador de eventos que cambie la página que corresponda.

   b. Al pinchar en el enlace Home se cargará en el DOM el texto h1 “IES Virgen del Carmen”.
   c. Al pinchar en el enlace DAM, se cargará una lista numerada con los módulos de 2º DAM.
   d. Al pinchar en Empresas, se cargará una lista con varios nombres de empresas.

7. Crea un editor y renderizador de markdown. En la parte superior tenemos un textarea donde el usuario escribe el texto en markdown simplificado. Debajo se encuentra el botón “Renderizar”. Al pulsarlo se debe generar, en la parte inferior, el DOM correspondiente con el formato adecuado. Para ello, separa el texto que ha insertado el usuario en líneas e implementa el siguiente formato:

   a. Si la línea comienza por # genera un H1
   b. Si la línea comienza por ## genera un H2
   c. Si la línea comienza por ### genera un H3
   d. Si comienza y termina en ** genera una etiqueta strong
   e. Si comienza por * y termina por * genera una etiqueta i (itálica)
   f. En caso contrario genera un párrafo.

8. Crea un buscador de libros en Javascript usando la API pública openlibrary.org. Usaremos estos endpoints:

- Búsqueda de libros: https://openlibrary.org/search.json?q=javascript
- El parámetro q (query) permite realizar la búsqueda.
- Portada: https://covers.openlibrary.org/b/isbn/$value-S.jpg
- $value es el valor del campo ISBN presente en el json de cada objeto libro. Es posible que algunos libros no tengan este campo. Implementa la siguiente funcionalidad:

   a. Crea una caja de texto y un botón “Buscar”. Cuando se pulse el botón se buscará el contenido de la caja de texto en la API de openlibrary.org usando el endpoint indicado anteriormente.
   b. Muestra el mensaje “No se ha encontrado ningún libro” si el número de libros encontrado es cero.
   c. En el caso de que se encuentren libros genera un bloque div con el texto “X libros encontrados” sustituyendo X por el número de libros de la respuesta.
   d. Finalmente genera dinámicamente un bloque div por cada libro con el título, subtítulo, portada y año de publicación.

10 Ejercicios IX sobre asincronía y DOM sin soluciones

1. Crea una página web que permita a los usuarios geolocalizar su IP. Para ello, se usarán estas dos APIs públicas:

- https://api.ipify.org/?format=json Retorna la IP del ordenador que realiza la consulta a la API
- http://demo.ip-api.com/json/IP?fields=66842623&lang=es. Retorna los datos de geolocalización de la IP. Sustituye IP, en la URL, por la IP a geolocalizar.

a. Crea una función asíncrona que consulte la IP del usuario.
b. Después, consulta el segundo endpoint con la IP obtenida.
c. Renderiza dinámicamente los datos de geolocalización. País, región, ciudad, código postal y proveedor de Internet (ISP).
d. Añade una caja de texto que permita al usuario geolocalizar cualquier IP deseada.
e. Inserta un mapa de Google Maps con la posición GPS obtenida (latitud y longitud).

```
<iframe
    width="600"
    height="400"
    frameborder="0"
    scrolling="no"
    marginheight="0"
    marginwidth="0"
    src="https://maps.google.com/maps?q='+LATITUD+','+LONGITUD+'&hl=es&z=14&amp;output=
        embed"></iframe>
```

2. La API https://api.imgflip.com/get_memes permite obtener las imágenes más usadas para crear memes en Internet.

   a. Consulta la API y genera una página web de forma dinámica en la que se muestren las imágenes de los memes y sus nombres.
   b. Crea otra versión en la que se muestran 4 memes por cada fila.

3. El endpoint https://dog.ceo/api/breeds/image/random retorna un objeto javascript con una URL de una imagen aleatoria de un perro.

   a. Crea una página web con 4 radio buttons con los valores 2, 4, 6, 8, 10.
   b. Al seleccionar un radio button se llama a una función que consulta el endpoint el número de veces que corresponda.
   c. Con las urls aleatorias obtenidas, muestra las imágenes de los perros en una lista.

4. Crea un juego para aprender a reconocer banderas de países. Para ello puedes usar la web https://flagsapi.com/#quick

   - Para mostrar la bandera de España simplemente usa la URL:

```
    <img src="https://flagsapi.com/ES/flat/64.png">
```

   - Crea un array con una lista de países.

```
const paises = [
    {
        code: 'ES',
        country: 'España'
    },
    {
        code: 'FR',
        country: 'Francia'
    }
]
```

a. Muestra una bandera al azar y genera una lista de 4 países posibles, por supuesto incluyendo el correcto. Puedes usar radio buttons para cada país.
b. Cuando el usuaro selecciona un país, muestra un mensaje informando si ha acertado.
c. Crea un botón que permita al usuario jugar de nuevo.
d. Crea dos contadores globales que permitan saber el número de aciertos y de rondas desde que comenzó el juego.

5. La API pública https://pokeapi.co/api/v2/pokemon/ permite consultar datos de Pokemon. Genera una página web con los datos de los pokemon que ofrece la API.

   a. Consulta el endpoint https://pokeapi.co/api/v2/pokemon para obtener una lista con los nombres y URL de detalle de cada pokemon.
   b. Para cada pokemon, consulta el endpoint de detalle y genera de forma dinámica un bloque div con el nombre, varias imágenes, las habilidades y el peso del Pokemon.

6. Crea un juego de Trivial con la siguiente API gratuita: https://opentdb.com/api_config.php. Los parámetros permiten obtener distintos tipos de preguntas.

   a. Consulta la API obteniendo 10 preguntas de tipo Verdadero/Falso de Informática: https://opentdb.com/api.php?amount=10&category=18&type=boolean
   b. Genera dinámicamente una página web con las preguntas. Para cada pregunta se muestran dos botones “Verdadero” y “Falso”.
   c. Al pulsar un botón la pregunta queda contestada y no se podrá cambiar la respuesta.
   d. Cuando se respondan todas las preguntas se mostrará la puntuación obtenida.
   e. Una vez terminado el juego se podrá comenzar de nuevo al pulsar un botón.
   f. Permite al jugador elegir el número de preguntas mediante un desplegable.
   g. Permite que el jugador seleccione la categoría de las preguntas.
   h. Permite al jugador seleccionar la dificultad de las preguntas.
   i. Usa localstorage para almacenar el estado del juego. Si se recarga la página se debe cargar el estado actual.

   - Ayuda:
     - En este ejercicio debes almacenar el estado actual del juego para almacenar la respuesta a cada pregunta, bloquear la respuesta y calcular la puntuación.

- Para ello puedes usar un objeto global que se inicializa al principio de la partida.

```
const preguntas = [/*Cargado de la API, no cambia*/]
const estadoPartida = {
    puntuacion: 0,
    respuestas: [null, null, null, null, null, null, null, null, null, null]
}
```

- El jugador responde la segunda pregunta que es Falsa y acierta, entonces el estado cambia a:

```
const estadoPartida = {
    puntuacion: 1,
    respuestas: [null, False, null, null, null, null, null, null, null, null]
}
```

- En la página vamos a tener muchos botones, dos por cada pregunta. Para manejar el evento de pulsación del botón y saber a qué pregunta corresponde hay distintas alternativas. Una de ellas es que cada botón tenga un id del tipo “pregunta-5-true”. Al pulsar el botón se lee el id y se parsea, de esta forma se obtiene la información necesaria para marcar la respuesta. Otra alternativa más sencilla es usar atributos “data-”. Estos atributos pertenecen a la especificación HTML5 y permiten almacenar información personalizada en un elemento HTML sin afectar a la apariencia o funcionalidad. Cuando generes los botones puedes usar:

```
pregunta.innerHTML = `
    <p>${pregunta}</p>
    <button data-posicion="${posicion}" data-respuesta="True">Verdadero</button>
    <button data-posicion="${posicion}" data-respuesta="False">Falso</button>
`;
```

- Para crear el manejador de eventos de todos los botones:

```
    const botones = document.querySelectorAll('button');
    botones.forEach(boton => {
        boton.addEventListener('click', () => {
            const posicion = parseInt(event.target.dataset.posicion);
            const respuesta = event.target.dataset.respuesta;
        });
    });
```

7. La API http://universities.hipolabs.com/search permite consultar una lista de universidades de todo el mundo. Crea una página web que muestre todas las universidades de forma paginada.

    a. Consulta el endpoint para obtener todas las universidades.
    b. En principio se mostrarán 100 universidades por página. Mediante una variable global controla la página actual. Renderiza las universidades correspondientes a la página actual.
    c. Crea dos botones que permitan moverte a la página anterior y siguiente. Controla los límites de páginas desactivando los botones cuando sea necesario.
    d. Genera un pie de página con botones o enlaces que permitan moverte directamente a cada página.

e. Añade un desplegable con los valores 20,50,100,200 que permita establecer cuantas universidades se muestran por página.
f. Genera dinámicamente un desplegable (select) con todos los países que aparecen en las universidades. Al seleccionar un país se mostrarán solo las universidades de dicho país.
g. Añade una caja de texto que permita buscar la universidad por su nombre.