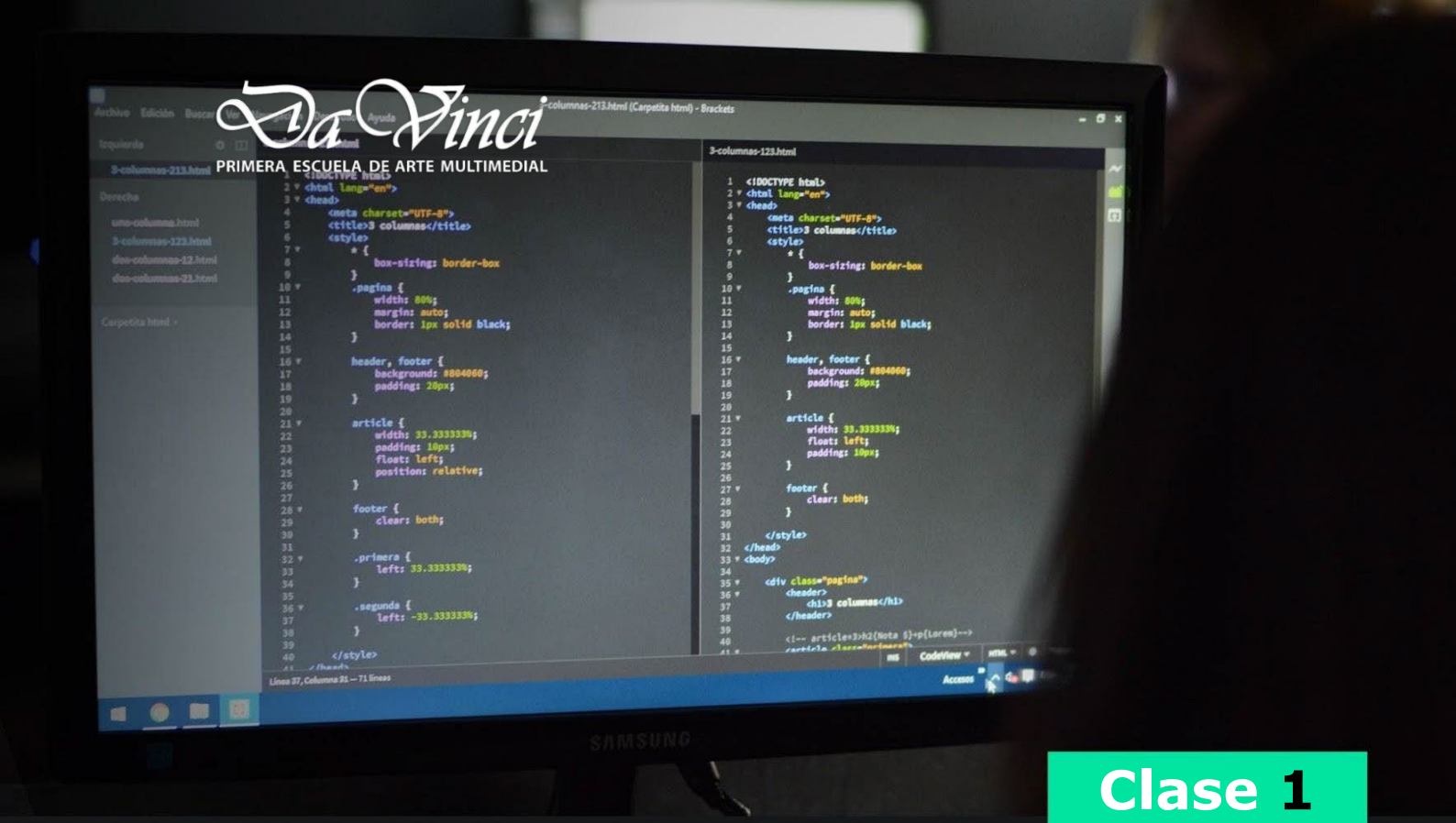


Clase 1

Diseño y Programación Web

Materia:
Aplicaciones Híbridas

Docente contenidista: MARCOS GALBÁN, Camila Belén

Javascript Asincrónico

Node.js

Node.js es un entorno de ejecución de JavaScript que permite a los desarrolladores ejecutar código JavaScript del lado del servidor. Está construido sobre el motor V8 de Chrome, que convierte el código JavaScript en código máquina, lo que lo hace muy rápido.

Puntos importantes:

- Single-Threaded Event Loop: Utiliza un bucle de eventos de un solo hilo para manejar operaciones de I/O (Input/Output) asíncronas.
- Non-Blocking I/O: Las operaciones de entrada/salida no bloquean el hilo principal, permitiendo una alta concurrencia.
- JavaScript Everywhere: Permite a los desarrolladores usar JavaScript tanto en el frontend como en el backend.
- Gran Ecosistema de Módulos: A través de npm (Node Package Manager), se puede acceder a miles de bibliotecas y paquetes.

¿Por qué usar Node.js?

- **Escalabilidad:** Ideal para aplicaciones que requieren manejar muchas conexiones simultáneamente, como chats en tiempo real, APIs y servidores web.
- **Rendimiento:** Gracias a su I/O no bloqueante y al motor V8, Node.js puede manejar un gran número de conexiones concurrentes con baja latencia.
- **Rapidez en el Desarrollo:** Permite a los desarrolladores usar el mismo lenguaje de programación en el frontend y backend, acelerando el desarrollo y reduciendo la curva de aprendizaje.

- **Comunidad Activa:** Una gran cantidad de módulos y paquetes disponibles en npm facilita la implementación rápida de nuevas funcionalidades.

Callbacks

Los callbacks son funciones que se pasan como argumentos a otras funciones y se ejecutan después de que se completa una operación. Los callbacks son fundamentales en JavaScript para manejar operaciones asíncronas, como las llamadas a APIs, la lectura de archivos y las operaciones de red.

Características:

- **Asincronía:** Permiten que el código continúe ejecutándose mientras se espera la finalización de una operación asíncrona.
- **Flexibilidad:** Pueden definir diferentes comportamientos para las operaciones según el callback pasado.
- **Encadenamiento:** Se pueden encadenar múltiples callbacks para realizar operaciones secuenciales.

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

function despedirse() {
  console.log('Chau!');
}

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

Callback Hell

El término "Callback Hell" se refiere a una situación en la que múltiples callbacks anidados resultan en un código difícil de leer y mantener. Esta situación ocurre cuando se utiliza un gran número de callbacks en operaciones asíncronas, especialmente cuando las operaciones dependen una de otra, creando una estructura de código profundamente anidada.

Características del Callback Hell:

- **Anidamiento Profundo:** Los callbacks anidados crean una estructura de código en forma de pirámide, también conocida como "Pyramid of Doom".
- **Dificultad de Mantenimiento:** El código se vuelve difícil de seguir y mantener debido a la anidación profunda y la complejidad creciente.
- **Gestión de Errores:** Manejar errores se vuelve complicado, ya que cada nivel de anidamiento puede requerir su propio manejo de errores.

```
function primeraFuncion(callback) {
    setTimeout(() => {
        console.log('Primera función completada');
        callback();
    }, 1000);
}

function segundaFuncion(callback) {
    setTimeout(() => {
        console.log('Segunda función completada');
        callback();
    }, 1000);
}

function terceraFuncion(callback) {
    setTimeout(() => {
        console.log('Tercera función completada');
        callback();
    }, 1000);
}

primeraFuncion(() => {
    segundaFuncion(() => {
        terceraFuncion(() => {
            console.log('Todas las funciones completadas');
        });
    });
});
```

Los callbacks son esenciales para manejar la asincronía en JavaScript, pero su uso excesivo puede llevar a un código complicado y difícil de mantener. En los siguientes temas, veremos cómo las promesas y async/await pueden ayudar a solucionar estos problemas y mejorar la legibilidad del código.

Promesas

Las promesas son un mecanismo en JavaScript para manejar operaciones asíncronas de una manera más limpia y manejable que los callbacks. Una promesa representa un valor que puede estar disponible ahora, en el futuro, o nunca. Permiten encadenar operaciones asíncronas y manejar errores de manera más clara.

Estados de una Promesa:

- Pendiente (Pending): Estado inicial, la promesa aún no se ha cumplido ni rechazado.
- Cumplida (Fulfilled): La operación asíncrona se completó con éxito.
- Rechazada (Rejected): La operación asíncrona falló.

Métodos de Promesas

Las promesas en JavaScript vienen con una serie de métodos incorporados que facilitan el manejo de operaciones asíncronas.

then()

El método then() se utiliza para manejar la resolución de una promesa. Recibe dos argumentos opcionales: una función callback para el caso de éxito y otra para el caso de error (aunque normalmente se maneja el error con catch).

catch()

El método `catch()` se utiliza para manejar el rechazo de una promesa.

finally()

El método finally() se utiliza para ejecutar código una vez que la promesa ha sido resuelta o rechazada, sin importar el resultado. No recibe ningún argumento.

```
const promesa = new Promise((resolve, reject) => {
```

```
    setTimeout(() => {
        const exito = true;
        if (exito) {
            resolve('Operación exitosa');
        } else {
            reject('Error en la operación');
        }
    }, 1000);
});

promesa
    .then(resultado => {
        console.log(resultado);
    })
    .catch(error => {
        console.error(error);
    })
    .finally(() => {
        console.log('Operación finalizada');
    });
```

Promise.all()

El método Promise.all() toma un array de promesas y devuelve una sola promesa que se resuelve cuando todas las promesas en el array se han resuelto, o se rechaza si alguna de las promesas se rechaza.

```
const promesa1 = new Promise((resolve) => setTimeout(resolve, 1000,
'Promesa 1 completada'));
const promesa2 = new Promise((resolve) => setTimeout(resolve, 2000,
'Promesa 2 completada'));

Promise.all([promesa1, promesa2])
    .then(resultados => {
        console.log(resultados); // ["Promesa 1 completada", "Promesa 2
completada"]
    })
    .catch(error => {
        console.error(error);
    });
```

Promise.race()

El método Promise.race() toma un array de promesas y devuelve una sola promesa que se resuelve o se rechaza tan pronto como una de las promesas en el array se resuelva o se rechace.

```
const promesa3 = new Promise((resolve) => setTimeout(resolve, 1000,
'Promesa 3 completada'));
const promesa4 = new Promise((resolve) => setTimeout(resolve, 2000,
'Promesa 4 completada'));

Promise.race([promesa3, promesa4])
    .then(resultado => {
        console.log(resultado); // "Promesa 3 completada"
    })
    .catch(error => {
        console.error(error);
    });
```

Promise.allSettled()

El método Promise.allSettled() toma un array de promesas y devuelve una promesa que se resuelve cuando todas las promesas en el array se han resuelto o rechazado. Proporciona una forma de saber cuándo todas las promesas han finalizado, sin importar su resultado.

```
const promesa5 = new Promise((resolve) => setTimeout(resolve, 1000,
'Promesa 5 completada'));
const promesa6 = new Promise((_, reject) => setTimeout(reject, 2000,
'Error en la promesa 6'));

Promise.allSettled([promesa5, promesa6])
    .then(resultados => {
        console.log(resultados);
        // [
        //   { status: "fulfilled", value: "Promesa 5 completada" },
        //   { status: "rejected", reason: "Error en la promesa 6" }
        // ]
    });
```

Promise.any()

El método Promise.any() toma un array de promesas y devuelve una promesa que se resuelve tan pronto como una de las promesas se resuelva, o se rechaza si todas las promesas se rechazan.

```
const promesa7 = new Promise((resolve) => setTimeout(resolve, 1000,
'Promesa 7 completada'));

const promesa8 = new Promise((_, reject) => setTimeout(reject, 2000,
'Error en la promesa 8'));
const promesa9 = new Promise((resolve) => setTimeout(resolve, 3000,
'Promesa 9 completada'));
```

```
Promise.any([promesa8, promesa9, promesa7])
    .then(resultado => {
        console.log(resultado); // "Promesa 7 completada"
    })
    .catch(error => {
        console.error('Ninguna promesa fue completada exitosamente');
    });
```

Funciones Asincrónicas

Las funciones asincrónicas (async) en JavaScript son funciones que permiten trabajar con promesas de manera más sencilla y manejable. La palabra clave async se utiliza para definir una función asíncrona, y dentro de ella, await se utiliza para esperar la resolución de una promesa. Esto hace que el código asíncrono se vea y se comporte de manera más similar al código sincrónico.

Puntos Clave:

- **async:** Define una función asincrónica que devuelve una promesa.
- **await:** Pausa la ejecución de la función asincrónica hasta que la promesa se resuelva o se rechace.

Ventajas:

- **Legibilidad:** El código es más fácil de leer y entender en comparación con el uso de promesas y callbacks anidados.
- **Simplicidad:** Hace que el manejo de operaciones asíncronas sea más directo y menos propenso a errores.

```
// Función que devuelve una promesa
function esperar(ms) {
    return new Promise(resolve => setTimeout(resolve, ms));
}

// Función asincrónica que usa await
async function ejecutar() {
    console.log('Inicio');

    // Esperar 1 segundo
    await esperar(1000);
    console.log('Esperó 1 segundo');
```

```
    // Esperar 2 segundos
    await esperar(2000);
    console.log('Esperó 2 segundos');

    console.log('Fin');
}

// Llamar a la función asincrónica
ejecutar();
```

El bloque try...catch en JavaScript se utiliza para manejar errores de manera controlada. Permite a los desarrolladores capturar excepciones (errores) que ocurren durante la ejecución de un bloque de código y ejecutar un código específico en respuesta a esas excepciones.

```
try {
    // Código que puede lanzar una excepción
} catch (error) {
    // Código para manejar la excepción
}
```

```
async function ejemplo() {
    const promesa = new Promise((resolve) => {
        setTimeout(() => {
            resolve('Operación exitosa');
        }, 1000);
    });

    try {
        const resultado = await promesa;
        console.log(resultado); // "Operación exitosa"
    } catch (error) {
        console.error(error);
    } finally {
        console.log('Operación finalizada');
    }
}

ejemplo();
```

El Event Loop en JavaScript

El Event Loop es un mecanismo fundamental en JavaScript que permite la ejecución de operaciones asíncronas sin bloquear el hilo principal del programa. Es responsable de manejar la concurrencia y de ejecutar el

código, recolectar y procesar eventos y ejecutar sub-tareas en un bucle continuo.

¿Cómo funciona?

Call Stack (Pila de Llamadas):

- Es una estructura de datos que sigue el principio LIFO (Last In, First Out).
- Cuando se llama a una función, se añade a la cima de la pila. Cuando una función termina, se elimina de la cima.

Heap (Montón):

- Es un área de memoria donde se almacenan objetos y datos dinámicos.

Queue (Cola de Mensajes):

- Es una estructura de datos que sigue el principio FIFO (First In, First Out).
- Los mensajes o eventos se añaden a la cola y se procesan secuencialmente.

Event Loop:

- Es un lazo que observa continuamente la pila de llamadas y la cola de mensajes.
- Si la pila de llamadas está vacía, el Event Loop toma el primer mensaje de la cola y lo coloca en la pila de llamadas para ser procesado.

Proceso del Event Loop

Iniciar el Programa:

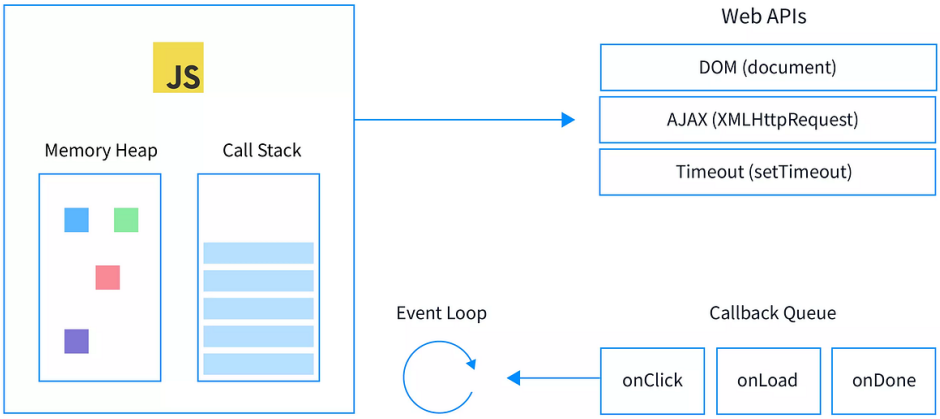
- El programa comienza ejecutando el código global. Las funciones se añaden a la pila de llamadas según se invocan.

Manejo de Operaciones Asíncronas:

- Operaciones como setTimeout, fetch, y eventos del DOM no se ejecutan inmediatamente. En su lugar, registran callbacks en el API del entorno (por ejemplo, el navegador).
- Una vez que una operación asíncrona se completa, su callback se añade a la cola de mensajes.

Procesamiento de la Cola de Mensajes:

- El Event Loop verifica continuamente si la pila de llamadas está vacía.
- Si está vacía, toma el primer mensaje de la cola y lo coloca en la pila de llamadas para ser ejecutado.



Conceptos importantes:

Operaciones Bloqueantes:

- Operaciones que bloquean la ejecución del código hasta que se completan.
- Ejemplo: while(true) {} bloqueará la ejecución indefinidamente.

Operaciones No Bloqueantes:

- Operaciones que permiten que el programa continúe ejecutándose mientras se completan en segundo plano.

- Ejemplo: setTimeout, fetch, operaciones de I/O en Node.js.

Conclusión

El Event Loop es esencial para la naturaleza asíncrona de JavaScript, permitiendo que las operaciones asíncronas se ejecuten de manera eficiente sin bloquear el hilo principal. Entender el Event Loop es crucial para escribir código asíncrono efectivo y evitar problemas comunes como el bloqueo del hilo principal y el Callback Hell.