Promesas y Asincronía

CÁTEDRA: CONSTRUCCIÓN DE INTERFACES DE USUARIO

Diferencias entre Código Síncrono y Asíncrono

Código Síncrono

El código síncrono se ejecuta en un orden secuencial. Cada línea de código debe completarse antes de que la siguiente pueda comenzar.

Esto significa que el flujo del programa se detiene hasta que una operación se completa, lo que puede ser ineficiente si la operación en cuestión es lenta o depende de recursos externos (por ejemplo, una solicitud a un servidor).

Características:

- **Bloqueo:** El hilo principal se detiene hasta que se completa una operación.
- **Fácil de leer:** El flujo de ejecución es lineal y predecible.
- **Rendimiento limitado:** Las operaciones que toman mucho tiempo pueden hacer que la aplicación se sienta lenta o que deje de responder.

Código Síncrono

En JavaScript, **Date.now()** es un método estático del objeto Date que devuelve el número de milisegundos transcurridos desde el 1 de enero de 1970 00:00:00 UTC.

Este método es útil para medir el tiempo transcurrido o para realizar cálculos de tiempo.

Código Síncrono

Ejemplo:

```
function tareaLenta() {
    let start = Date.now();
    while (Date.now() - start < 2000) {
      // Simulando una tarea lenta de 2 segundos
    }
    console.log('Tarea lenta completada');
}

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

En este ejemplo, el programa se bloquea durante 2 segundos mientras se ejecuta **tareaLenta**, y no se ejecuta la línea **console.log('Fin')** hasta que se completa **tareaLenta**.

Salida:

Inicio
Tarea lenta completada
Fin

Código Asíncrono

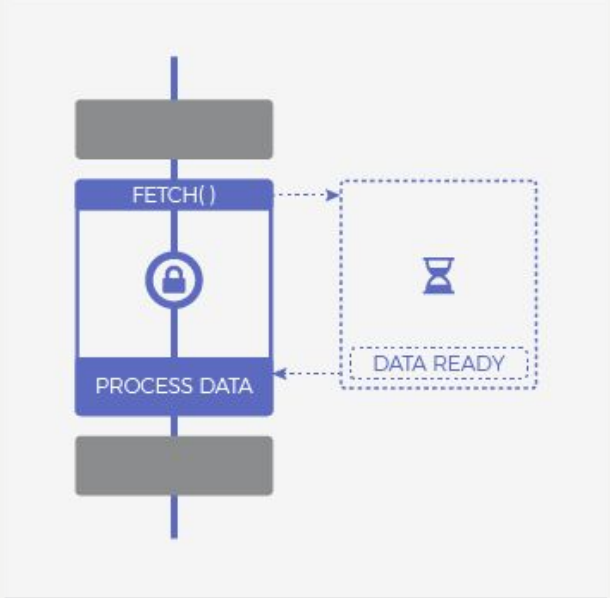
El código asíncrono permite que el programa continúe ejecutándose mientras se completan operaciones que pueden tardar un tiempo indeterminado.

Esto mejora la eficiencia y la capacidad de respuesta de la aplicación al no bloquear el hilo principal durante la espera de estas operaciones.

Características:

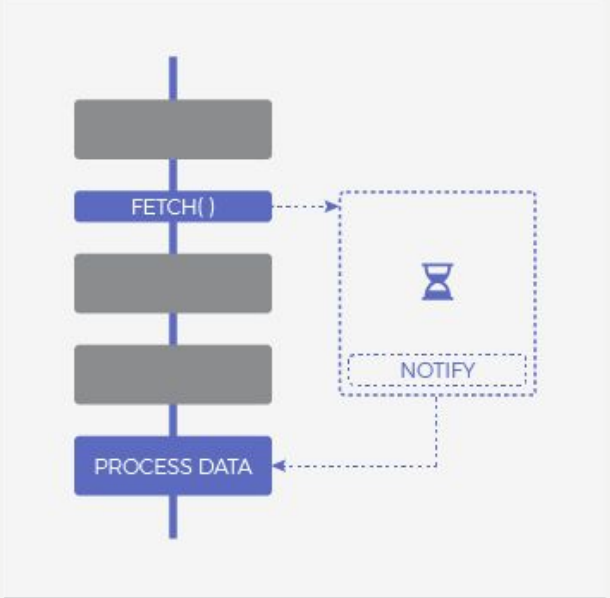
- **No bloqueo:** El hilo principal no se detiene mientras se realizan operaciones asíncronas.
- **Más difícil de leer:** El flujo de ejecución puede ser más complicado de seguir debido a los callbacks o promesas.
- **Mejor rendimiento:** Permite manejar múltiples operaciones al mismo tiempo sin bloquear la aplicación.

Comparación Directa



SÍNCRONO

Ejecución secuencial. Retorna cuando la operación ha sido completada en su totalidad.



ASÍNCRONO

La finalización de la operación es notificada al programa principal. El procesado de la respuesta se hará en algún momento futuro.

Comparación Directa

Aspecto	Código Síncrono	Código Asíncrono
Ejecución	Secuencial	Paralela/Concurrente
Bloqueo	Bloquea el hilo principal	No bloquea el hilo principal
Facilidad	Más fácil de leer y entender	Puede ser más complejo debido a callbacks/promesas/async-await
Rendimiento	Menos eficiente en operaciones de larga duración	Más eficiente en operaciones que requieren tiempo
Uso Común	Operaciones rápidas y sencillas	Operaciones de E/S, solicitudes a servidores, temporizadores

CallBack

Callback

Un **callback** es una función que se pasa como argumento a otra función y que no se ejecuta en el momento en que se pasa. Se ejecuta en otro momento de forma sincrónica o asincrónica.

Ejemplo:

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

hacerAlgo(() => {
    console.log('Terminado');
});
```

En este ejemplo, la función **hacerAlgo** acepta una función **callback** como argumento. Después de realizar su operación principal (imprimir "Haciendo algo..."), llama a la función **callback**, que imprime "Terminado".

Callback

Los callbacks son una forma de manejar operaciones asíncronas en JavaScript.

¿Este ejemplo te suena?

```
let boton= document.getElementById("miBoton");

function MostrarMsj(){
    alert("Boton apretado")
};

boton.addEventListener("click", MostrarMsj);
```

Callback

Otro ejemplo:

```
const mensaje = function() {
    console.log("El mensaje se imprime después de 2 segundos");
}
setTimeout(mensaje, 2000);
```

Hay un método incorporado en JavaScript llamado **setTimeout**, que llama a una función o evalúa una expresión después de un período de tiempo determinado (en milisegundos). Así que aquí, la función **mensaje** está siendo llamada después de que hayan pasado 2 segundos. (1 segundo = 1000 milisegundos).

En otras palabras, la función de mensaje está siendo llamada después de que algo sucedió (después de que pasaron 2 segundos para este ejemplo), pero no antes. Así que la función de mensaje es un ejemplo de una función callback.

Callbacks Asíncronos

Los callbacks son particularmente útiles para manejar operaciones asíncronas, como la lectura de archivos, la realización de solicitudes HTTP, o el manejo de temporizadores.

Ejemplo:

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

setTimeout(() => {
  console.log('Mitad');
}, 1000);

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

- **Inicio** se imprime inmediatamente.
- **Fin** se imprime inmediatamente después.
- **Mitad** se imprime después de 1 segundo, cuando el callback del setTimeout se ejecuta.

Callbacks - Tener en cuenta

- No ejecutar la función al pasarla:

```
hacerAlgo(tareaTerminada()); //ejecuta en el momento
hacerAlgo(tareaTerminada);   //la pasa como referencia
```

- No anidar tantos callbacks dentro de otros:

```
hacerAlgo(() => {
  hacerOtro(() => {
    hacerFinal(() => {
      // ...
    });
  });
});
```

Problemas con Callbacks

Uno de los principales problemas con los callbacks es el llamado "callback hell" o infierno de callbacks, qué ocurre cuando hay múltiples niveles de anidamiento de funciones callback, lo que hace que el código sea difícil de leer y mantener.

Problemas con Callbacks (Ejemplo)

```
function pasoUno(callback) {
  setTimeout(() => {
    console.log('Paso Uno');
    callback();
  }, 1000);
}
```

```
function pasoDos(callback) {
  setTimeout(() => {
    console.log('Paso Dos');
    callback();
  }, 1000);
}
```

```
function pasoTres(callback) {
  setTimeout(() => {
    console.log('Paso Tres');
    callback();
  }, 1000);
}
```

```
pasoUno(() => {
  pasoDos(() => {
    pasoTres(() => {
      console.log('Completados');
    });
  });
});
```

Promesas

Promesas

Una **promesa** en JavaScript es una forma de manejar tareas que **toman tiempo en completarse**, como cargar datos de un servidor.

Sirve para decir: "*Esto va a terminar más adelante, y cuando termine, quiero que hagas algo con el resultado o que manejes un posible error*".

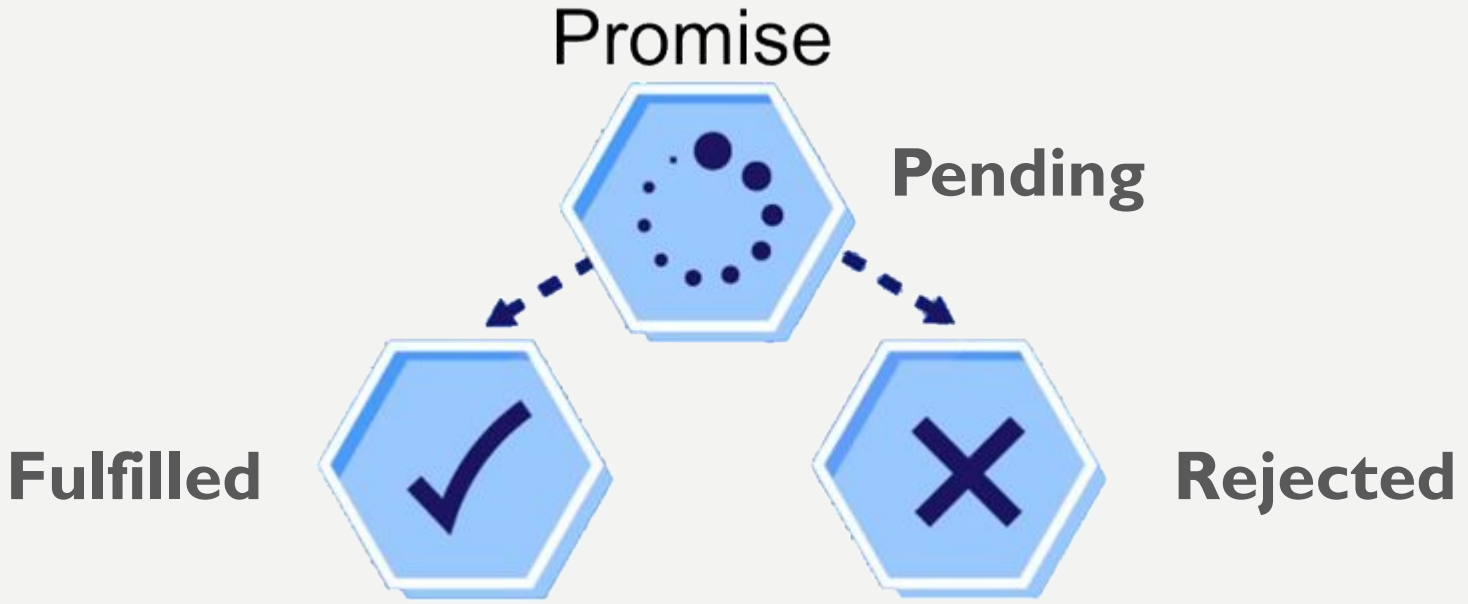
Entonces, una **promesa es un objeto** que representa la **eventual finalización** (o falla) de una operación asíncrona y su valor resultante.

Las promesas permiten encadenar operaciones asíncronas de manera más legible y manejable que los callbacks, evitando el "callback hell".

Promesas

Una promesa puede estar en uno de tres estados:

- **Pendiente (pending):** La operación asíncrona aún no ha finalizado.
- **Cumplida (fulfilled):** La operación asíncrona se completó con éxito.
- **Rechazada (rejected):** La operación asíncrona falló.



Promesas

Creación de una Promesa:

```
let promesa = new Promise((resolve, reject) => {
  let exito = true; // simulamos un resultado
  if (exito) {
    resolve('¡Operación exitosa!');
  } else {
    reject('Operación fallida.');
  }
});
```

En este ejemplo, **new Promise** crea una nueva promesa. La función pasada como argumento recibe dos parámetros (**resolve** y **reject**) que podemos invocar en nuestro código para indicar si la promesa finalizó correctamente (resolve) o falló (reject)

Métodos de una Promesa

Las promesas tienen varios métodos importantes para manejar su resultado. Estos métodos se usan para indicar lo que queremos que pase cuando la promesa finalice de manera exitosa o fallida

.then(onFulfilled):

- Se ejecuta cuando la promesa se cumple.
- onFulfilled es una función que se llama cuando la promesa se cumple.

.catch(onRejected):

- Se ejecuta cuando la promesa es rechazada.

Ejemplo: Creación y uso de una promesa

Creación la promesa:

```
let promesa = new Promise((resolve, reject) => {
  let exito = true;
  setTimeout(() => {
    if (exito) {
      resolve('¡Operación exitosa!');
    } else {
      reject('Operación fallida.');
    }
  }, 2000);
});
```

Le asigno a sus métodos .then y .catch las funciones callback que quiero que ejecute cuando finalice:

```
promesa
  .then((mensaje) => {
    console.log(mensaje);
  })
  .catch((error) => {
    console.error(error);
  })
```

Métodos de una Promesa

Explicación:

- La promesa se crea con un tiempo de espera de 2 segundos simulando una operación asíncrona.
- Si **exito** es **true**, la promesa se resuelve con el mensaje "¡Operación exitosa!".
- Si **exito** es **false**, la promesa se rechaza con el mensaje "Operación fallida.".
- **.then** maneja la promesa cumplida, **.catch** maneja la promesa rechazada.

Promesas

Encadenamiento de Promesas

Las promesas permiten encadenar múltiples operaciones asíncronas de manera secuencial, lo que facilita la lectura y mantenimiento del código.

Ejemplo con Promesas:

```
function pasoUno() {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log('Paso Uno');
      resolve();
    }, 1000);
  });
}
function pasoDos() {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log('Paso Dos');
      resolve();
    }, 1000);
  });
}
```

```
function pasoTres() {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log('Paso Tres');
      resolve();
    }, 1000);
  });
}
pasoUno()
  .then(pasoDos)
  .then(pasoTres)
  .then(() => {
    console.log('Todos los pasos
completados');
  });
```

Promesas

Manejo de Errores

Las promesas proporcionan una forma estructurada de manejar errores en operaciones asíncronas.

Ejemplo

```
let promesa = new Promise((resolve, reject) => {
  reject('Error en la operación.');
});

promesa
  .then((resultado) => {
    console.log(resultado);
  })
  .catch((error) => {
    console.error('Se produjo un error: ' + error);
  });
```

La promesa se rechaza inmediatamente con un mensaje de error. El método .catch captura y maneja el error.

async y await

“async” y “await”

async y **await** son palabras clave en JavaScript que permiten escribir código asíncrono de manera más clara y fácil de entender. **async** se utiliza para declarar una función asíncrona, y **await** se utiliza dentro de una función asíncrona para esperar la resolución de una promesa.

async

- La palabra clave **async** se coloca antes de la declaración de una función.
- Una función declarada con async **siempre devuelve una promesa.**
- Si la función **async** devuelve un valor, ese valor se envuelve automáticamente en una promesa resuelta.
- Si la función **async** lanza una excepción, la promesa devuelta se rechaza con el valor de la excepción.

async

Sintaxis:

```
async function miFuncion() {
  // Código asíncrono
  return 'Hola';
}

miFuncion().then((resultado) => {
  console.log(resultado); // 'Hola'
});
```

await

- La palabra clave **await** solo puede ser utilizada **dentro de una función async.**
- **await** pausa la ejecución de la función async hasta que la promesa a la que se aplica se resuelve.
- Devuelve el valor resuelto de la promesa.
- Si la promesa es rechazada, **await** lanza la excepción rechazada.

await

```
async function miFuncion() {
  let promesa = new Promise((resolve) => {
    setTimeout(() => resolve('Hola'), 2000);
  });
  let resultado = await promesa; // Espera hasta que la promesa se resuelve
  console.log(resultado); // 'Hola'
}
miFuncion();
```

Sintaxis:

await AlgoQueDevuelveUnaPromesa;

Comparamos código

```
function obtenerDatos() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve('Datos obtenidos');
    }, 2000);
  });
}
```

Con .then y .catch:

```
obtenerDatos()
  .then((datos) => {
    console.log(datos);
  })
  .catch((error) => {
    console.error(error);
  });
```

Con async y await:

```
async function procesarDatos() {
  try {
    let datos = await obtenerDatos();
    console.log(datos);
  } catch (error) {
    console.error('Error al obtener
                   datos:', error);
  }
}

procesarDatos();
```

Ejemplo Completo “async” y “await”

Explicación:

- **obtenerDatos** es una función que simula la obtención de datos con una promesa que se resuelve después de 2 segundos.
- **procesarDatos** es una función asíncrona que utiliza await para esperar la resolución de obtenerDatos.
- **try** y **catch** se utilizan para manejar cualquier error que pueda ocurrir durante la espera de la promesa.

“async” y “await”

En este ejemplo, **async** y **await** simplifican la lectura y el manejo de operaciones asíncronas al evitar la necesidad de encadenar múltiples then y catch.

Ventajas de async y await

- **Código más claro y legible:** El código asíncrono escrito con async y await se parece más al código síncrono, lo que facilita su comprensión.
- **Manejo sencillo de errores:** Usando try y catch, se puede manejar errores de manera más clara que con múltiples catch en promesas encadenadas.
- **Encadenamiento fácil:** Las funciones asíncronas pueden esperar otras funciones asíncronas de manera natural, evitando el anidamiento profundo de then.

Ejemplo previo con “async” y “await”

```
async function ejecutarPasos() {
  await pasoUno();
  await pasoDos();
  await pasoTres();
  console.log('Todos los pasos completados');
}

ejecutarPasos();
```

Ambos ejemplos proporcionan una forma más legible y manejable de manejar operaciones asíncronas complejas, reduciendo la necesidad de anidar múltiples callbacks.

Resumen

Asincronía: Es la capacidad de ejecutar tareas sin detener el resto del código. En JavaScript, permite esperar resultados (como datos de un servidor) sin congelar la página. Ejemplo: setTimeout(), eventos, etc.

Callback: Es una función que se pasa como argumento a otra función para que se ejecute más tarde. Puede ser sincrónico o asincrónico.

Resumen

Promesa (Promise): Es un objeto que representa el resultado futuro de una operación asincrónica. Tiene 3 estados: pending, fulfilled, rejected.

```
fetch("datos.json")
.then(res => res.json())
.then(data => console.log(data))
.catch(err => console.error(err));
```

async / await: Es una forma moderna y más legible de trabajar con promesas.

- async convierte una función en asincrónica (devuelve una promesa).
- await espera el resultado de una promesa sin usar .then().

```
async function cargar() {
  const res = await fetch("datos.json");
  const data = await res.json();
  console.log(data);
}
```

Ejercicio

Convertir la siguiente función de callbacks a promesas.

```
function leerArchivo(callback) {
  setTimeout(() => {
    callback('Contenido del archivo');
  }, 1000);
}

leerArchivo((contenido) => {
  console.log(contenido);
});
```

Resolución

```
function leerArchivo() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve('Contenido del archivo');
    }, 1000);
  });
}

leerArchivo().then((contenido) => {
  console.log(contenido);
});
```

Ejercicio

Crea una función obtenerDatos que devuelva una promesa. La promesa debe rechazar después de 1 segundo con el mensaje "Error al obtener datos". Luego, utiliza async y await para llamar a esta función, maneja el error con try y catch y muestra el mensaje de error en la consola.

Resolución

```
function obtenerDatos() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      reject('Error al obtener datos');
    }, 1000);
  });
}
```

Resolución

```
async function ejecutarObtenerDatos() {
  try {
    let datos = await obtenerDatos();
    console.log(datos);
  } catch (error) {
    console.error('Se produjo un error:', error);
  }
}

ejecutarObtenerDatos();
```

Ejercicio: Ejecutar Promesas en Paralelo

Crea tres funciones tareaUno, tareaDos y tareaTres que devuelvan promesas. Cada promesa debe resolverse después de 2 segundos con el mensaje "Tarea Uno completada", "Tarea Dos completada" y "Tarea Tres completada" respectivamente. Luego, utiliza Promise.all y async/await para llamar a estas funciones en paralelo y mostrar los mensajes en la consola.

Resolución

```
function tareaUno() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve('Tarea Uno completada');
    }, 2000);
  });
}
function tareaDos() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve('Tarea Dos completada');
    }, 2000);
  });
}
```

Resolución

```
function tareaTres() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve('Tarea Tres completada');
    }, 2000);
  });
}
async function ejecutarTareasParalelas() {
  let [mensajeUno, mensajeDos, mensajeTres] = await Promise.all([tareaUno(),
tareaDos(), tareaTres()]);
  console.log(mensajeUno);
  console.log(mensajeDos);
  console.log(mensajeTres);
  console.log('Todas las tareas completadas');
}
ejecutarTareasParalelas();
```