

CALLBACK, PROMESAS Y PROGRAMACION ASINCRONA

MODULO 1 - SESION 4

SONDEO DE TEMAS ANTERIORES

¿Preguntas de arreglos, funciones y
estructuras de decisión?



CALLBACK, PROMESAS Y PROGRAMACION ASINCRONA.



En JavaScript, las callbacks, promesas y programación asíncrona son herramientas clave para manejar tareas que requieren tiempo, como solicitudes de red, operaciones de E/S (entrada/salida) y otras operaciones que no se pueden ejecutar de manera sincrónica.

CALLBACK

Join at menti.com | use code 8918 4103

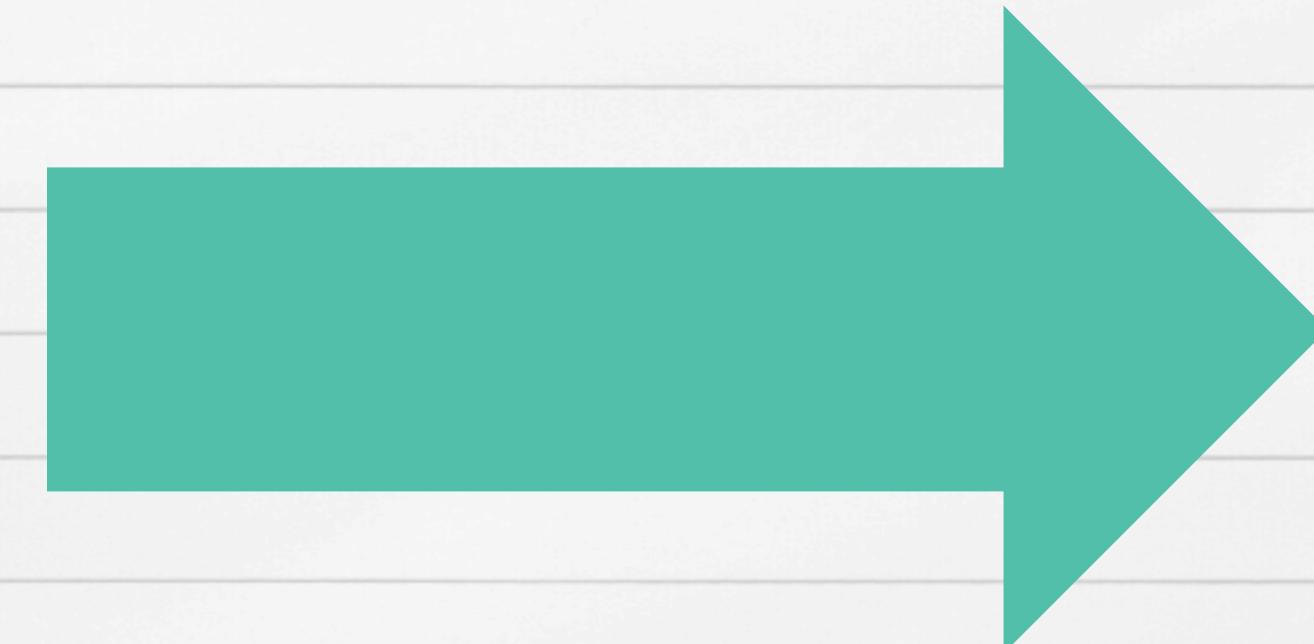
Participemos en la
siguiente
encuesta

CALLBACK

Retroacciones o devoluciones de llamada:

Una callback es una función que se pasa como argumento a otra función y se ejecuta después de que esta última finaliza su tarea. Son comunes en operaciones asíncronas, donde se utiliza para manejar la respuesta una vez que la operación se completa.

Veamos un ejemplo:



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

function saludar(nombre) {
  console.log("Hola " + nombre);
}

function procesarEntradaUsuario(callback) {
  const rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout
  });

  rl.question("Por favor ingresa tu nombre: ", (nombre) => {
    callback(nombre);
    rl.close();
  });
}

procesarEntradaUsuario(saludar);
```

```
PS C:\Users\jwill\Desktop> node comprobacion.js
Por favor ingresa tu nombre: William Morales
Hola William Morales
PS C:\Users\jwill\Desktop> □
```

Veamos un ejercicio sin usar callback

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

function saludar(nombre) {
  console.log("Hola " + nombre);
}

function procesarEntradaUsuario() {
  const rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout
  });
  rl.question("Por favor ingresa tu nombre: ", (nombre) => {
    saludar(nombre);
    rl.close();
  });
}

procesarEntradaUsuario();
```

```
PS C:\Users\jwill\Desktop> node comprobacion.js
Por favor ingresa tu nombre: Juan Canales
Hola Juan Canales
PS C:\Users\jwill\Desktop> 
```

Diferencia entre el uso y no uso de callback

La diferencia fundamental entre usar una callback y no usarla radica en cómo se estructura y organiza el flujo de tu programa, especialmente en situaciones donde se manejan tareas asíncronas o se requiere modularidad en el código.

Diferencia usando callback

- Las callbacks se utilizan para manejar operaciones asíncronas de manera más controlada y flexible.
- Permiten pasar una función como argumento a otra función para ser ejecutada más adelante, lo que es útil en situaciones como solicitudes HTTP, operaciones de archivo, etc.
- Ayudan a evitar la pirámide de callbacks anidadas (callback hell) al permitir un enfoque más modular y estructurado del código.

Diferencia sin usar callback

- Cuando no usas callbacks, la lógica puede estar más directamente integrada en la función que ejecuta la tarea asíncrona.
- Puede simplificar el código en casos donde la lógica de procesamiento es simple y sigue inmediatamente después de la operación asíncrona.
- Es útil en situaciones donde la tarea asíncrona y la lógica de procesamiento están fuertemente acopladas y no necesitan ser separadas en funciones distintas.

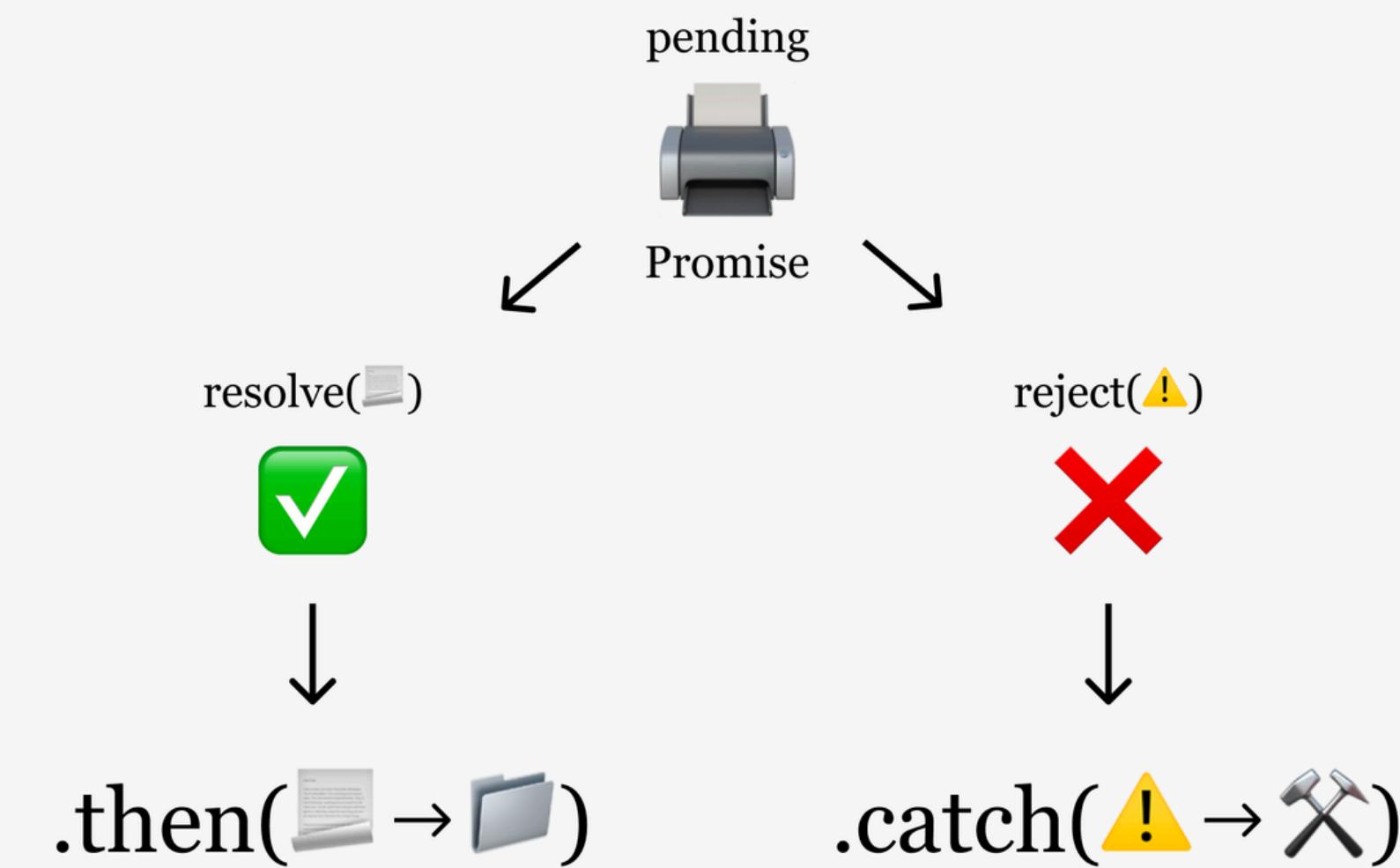
En resumen, la elección entre usar una callback o no depende del contexto y la complejidad de las tareas asíncronas que estés manejando. Las callbacks son una herramienta poderosa para gestionar operaciones asíncronas de manera flexible y modular, mientras que en casos simples y directos, puedes optar por una estructura más integrada sin callbacks.



Promesas en JS

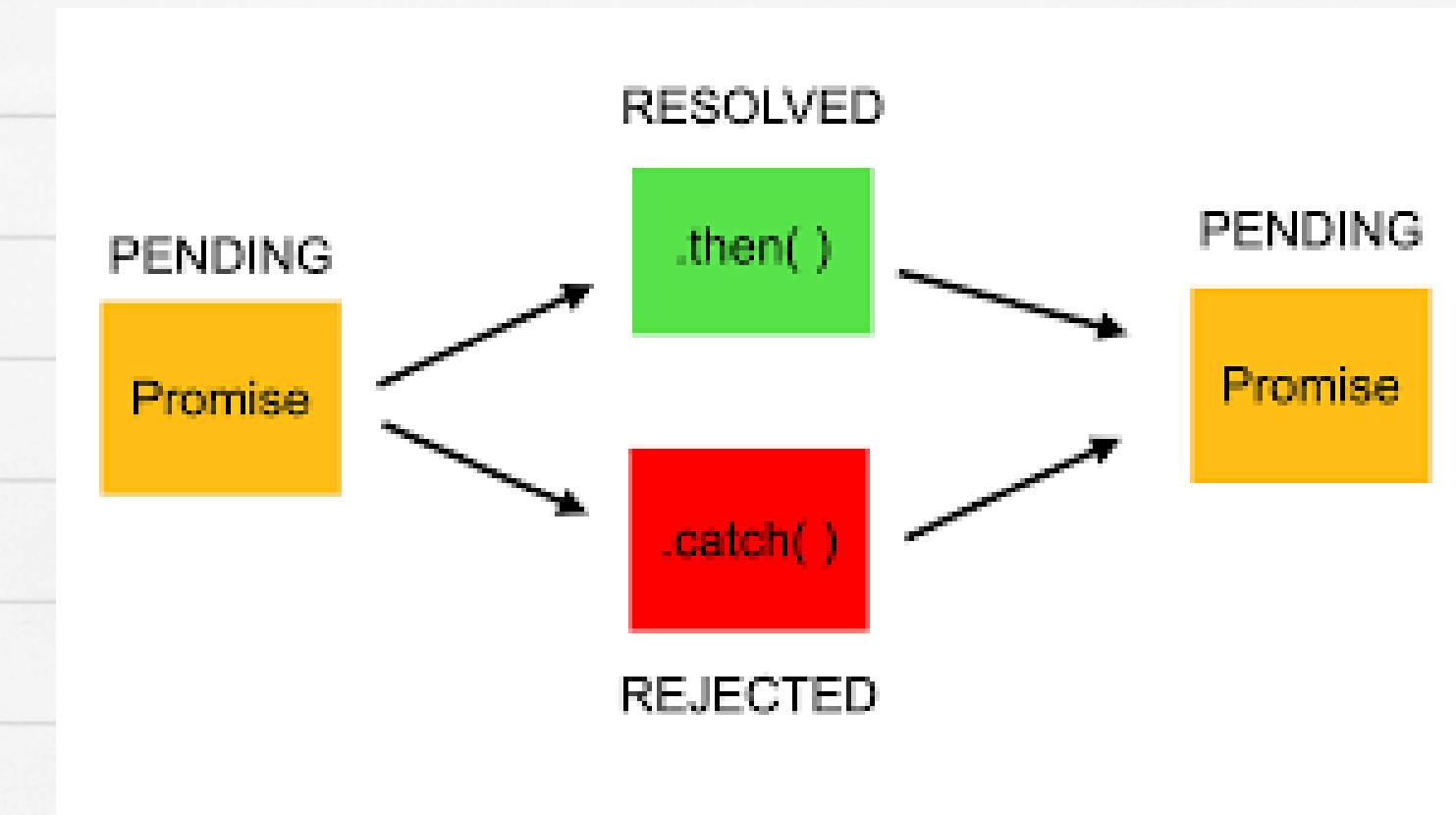
Promesas

Las promesas en JavaScript son objetos que representan la finalización (o el fracaso) eventual de una operación asíncrona y su valor resultante. Proporcionan una forma más elegante y eficiente de manejar operaciones asíncronas en comparación con las callbacks tradicionales.



Características de las promesas

- Una promesa puede estar en uno de tres estados: pendiente, resuelta (fulfilled) o rechazada (rejected).
- Cuando una promesa se resuelve, se ejecuta la función then asociada y se pasa el valor resuelto como argumento.
- Cuando una promesa se rechaza, se ejecuta la función catch asociada para manejar el error.



Porque usar promesas

Las promesas son útiles cuando trabajas con operaciones asíncronas como solicitudes HTTP, lectura/escritura de archivos, o cualquier tarea que tome un tiempo significativo y no deba bloquear el hilo de ejecución principal.

Las promesas proporcionan un manejo más limpio y estructurado de operaciones asíncronas en comparación con las callbacks anidadas (callback hell). Además, permiten encadenar múltiples operaciones de manera legible y manejar errores de manera centralizada.

```
let miPromesa = new Promise((resolve, reject) => {
  setTimeout(() => {
    let exito = false;
    if (exito) {
      resolve("Operación exitosa");
    } else {
      reject("Error en la operación");
    }
  }, 2000);
});
miPromesa
  .then((mensaje) => {
    console.log(mensaje);
  })
  .catch((error) => {
    console.error(error);
  });
}
```

```
PS C:\Users\jwill\Desktop> node comprobacion.js
Error en la operación
PS C:\Users\jwill\Desktop> 
```

```
1  function obtenerDatos(url) {
2      return fetch(url)
3          .then(response => {
4              if (!response.ok) {
5                  throw new Error("Error en la solicitud HTTP");
6              }
7              return response.json();
8          });
9  }
10 obtenerDatos('https://jsonplaceholder.typicode.com/posts/1')
11     .then(data => {
12         console.log(data);
13     })
14     .catch(error => {
15         console.error(error);
16     });
17 }
```

```
PS C:\Users\jwill\Desktop> node comprobacion.js
{
  userId: 1,
  id: 1,
  title: 'sunt aut facere repellat provident occaecati excepturi optio reprehenderit',
  body: 'quia et suscipit\n' +
    'suscipit recusandae consequuntur expedita et cum\n' +
    'reprehenderit molestiae ut ut quas totam\n' +
    'nostrum rerum est autem sunt rem eveniet architecto'
```

```
function obtenerDatos(url) {
  return fetch(url)
    .then(response => {
      if (!response.ok) {
        throw new Error("Error en la solicitud HTTP"+ response.status);
      }
      return response.json();
    });
}

obtenerDatos('https://jsonplaceholder.typicode.com/posts/999')
  .then(data => {
    console.log(data);
  })
  .catch(error => {
    console.error(error);
  });

```

```
PS C:\Users\jwill\Desktop> node comprobacion.js
Error: Error en la solicitud HTTP404
    at C:\Users\jwill\Desktop\comprobacion.js:5:23
    at process.processTicksAndRejections (node:internal/process/task_queues:95:5)
PS C:\Users\jwill\Desktop> 
```

Las promesas son como los try catch

Las promesas en JavaScript no son exactamente equivalentes a los bloques try-catch, aunque tienen ciertas similitudes en términos de manejo de errores.

Manejo de errores

Promesas: En las promesas, se utiliza el método catch para manejar errores que ocurren durante la ejecución de la operación asíncrona.

Try-Catch: En los bloques try-catch, se utiliza el bloque try para envolver el código que puede lanzar un error, y el bloque catch para manejar cualquier error que ocurra dentro del bloque try.

En ejecución asíncrona.

Promesas: Las promesas permiten manejar operaciones asíncronas de manera más estructurada, encadenando operaciones con `then` y manejando errores con `catch`.

Try-Catch: Los bloques `try-catch` se utilizan principalmente para manejar errores en código síncrono y no son adecuados para manejar errores en operaciones asíncronas directamente.

Encadenamiento y legibilidad

Promesas: Las promesas permiten encadenar múltiples operaciones asíncronas de manera legible y estructurada, lo que facilita la comprensión del flujo de ejecución.

Try-Catch: Los bloques `try-catch` pueden resultar menos legibles y más complejos de manejar cuando se trata de manejar errores en operaciones asíncronas o en el flujo de código que involucra múltiples llamadas a funciones.

Uso de código Asíncrono.

Promesas: Las promesas son especialmente útiles en código asíncrono, como solicitudes HTTP, lectura/escritura de archivos, etc., donde se necesita un manejo estructurado de resultados y errores.

Try-Catch: Los bloques try-catch son más adecuados para manejar errores en código síncrono, como operaciones matemáticas, manipulación de cadenas, etc.

Las promesas en JavaScript son una forma más estructurada y flexible de manejar operaciones asíncronas y manejar errores relacionados con ellas. Mientras que los bloques try-catch son más adecuados para el manejo de errores en código síncrono y no son la opción preferida para operaciones asíncronas.

Programación Asíncrona.

La programación asíncrona en JavaScript se refiere a la capacidad de ejecutar múltiples tareas de manera simultánea o en segundo plano, sin bloquear el hilo de ejecución principal. Esto permite que el código continúe ejecutándose mientras se realizan operaciones que pueden tomar tiempo, como solicitudes HTTP, operaciones de E/S (entrada/salida), temporizadores, entre otras.

La programación asíncrona en JavaScript es crucial para aplicaciones web y de servidor, donde es común realizar múltiples tareas simultáneamente, como cargar recursos externos, manejar interacciones de usuario, realizar operaciones de red, entre otras. Permite que la aplicación sea más eficiente y receptiva al no bloquear el hilo de ejecución principal, lo que mejora la experiencia del usuario y la eficacia del programa en general.

La programación Asíncrona siempre requiere de setTimeout.

Si bien setTimeout es una forma común de simular operaciones asíncronas para propósitos de demostración y pruebas, existen muchas otras situaciones donde la programación asíncrona es necesaria sin involucrar setTimeout.

Por ejemplo

Solicitudes HTTP: Cuando haces una solicitud a un servidor web utilizando fetch, XMLHttpRequest, o cualquier biblioteca de solicitudes HTTP como Axios, estas operaciones son inherentemente asíncronas ya que el servidor no responde de inmediato y la respuesta puede tardar en llegar.

Por ejemplo

Operaciones de E/S (Entrada/Salida): Cualquier operación que involucre lectura/escritura de archivos, acceso a bases de datos, o interacciones con dispositivos de hardware, generalmente se realiza de manera asíncrona en JavaScript para evitar bloquear el hilo de ejecución principal.

Eventos del DOM: Las interacciones del usuario con una página web, como hacer clic en un botón, mover el mouse, o ingresar texto en un campo de entrada, generan eventos que se manejan de manera asíncrona en JavaScript.

Temporizadores nativos: Si bien setTimeout y setInterval son funciones que simulan operaciones asíncronas, también existen temporizadores nativos en el lenguaje, como requestAnimationFrame, que se utilizan para tareas específicas relacionadas con la animación y la actualización del DOM.

¿PREGUNTAS?

Ejecutemos la practica.