Asincronismo en JavaScript

Módulo 1: Introducción al Asincronismo

1.1 ¿Qué es el asincronismo?

El asincronismo en programación se refiere a la capacidad de ejecutar tareas de forma no secuencial, permitiendo que el programa continúe su ejecución sin esperar a que una tarea larga o que requiera mucho tiempo se complete.

Pregunta: ¿Por qué crees que el asincronismo es importante en JavaScript?

1.2 Conceptos clave

1. **Operaciones síncronas vs asíncronas**
2. **Callbacks**
3. **Promesas**
4. **Async/Await**

Actividad: Investiga y escribe una breve definición de cada uno de estos conceptos.

Módulo 2: Callbacks

2.1 ¿Qué es un callback?

Un callback es una función que se pasa como argumento a otra función y se ejecuta después de que esa función haya terminado.

2.2 Ejemplo práctico

```
function obtenerDatos(callback) {
  setTimeout(() => {
    const datos = { id: 1, nombre: "Juan" };
    callback(datos);
  }, 2000);
}

function procesarDatos(datos) {
  console.log("Datos procesados:", datos);
}

obtenerDatos(procesarDatos);
console.log("Esperando datos...");
```

Explicación del código:

1. Definimos una función `obtenerDatos` que simula una operación asíncrona usando `setTimeout`.
2. La función `obtenerDatos` acepta un callback como parámetro.
3. Después de 2 segundos, se llama al callback con los datos obtenidos.
4. Definimos una función `procesarDatos` que será nuestro callback.

5. Llamamos a `obtenerDatos` pasando `procesarDatos` como callback.
6. Inmediatamente después, imprimimos "Esperando datos...".

Pregunta: ¿En qué orden crees que se imprimirán los mensajes en la consola? ¿Por qué?

2.3 Reto

Crea una función que simule la obtención de datos de un usuario y otra que simule la obtención de sus publicaciones. Usa callbacks para asegurarte de que las publicaciones se obtienen solo después de tener los datos del usuario.

Módulo 3: Promesas

3.1 ¿Qué es una promesa?

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

3.2 Ejemplo práctico

```
function obtenerDatos() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const exito = true;
      if (exito) {
        resolve({ id: 1, nombre: "María" });
      } else {
        reject("Error al obtener los datos");
      }
    }, 2000);
  });
}

obtenerDatos()
  .then(datos => console.log("Datos obtenidos:", datos))
  .catch(error => console.error(error));

console.log("Esperando datos...");
```

Explicación del código:

1. Definimos una función `obtenerDatos` que devuelve una promesa.
2. Dentro de la promesa, usamos `setTimeout` para simular una operación asíncrona.
3. Si la operación es exitosa, llamamos a `resolve` con los datos.
4. Si hay un error, llamamos a `reject` con un mensaje de error.
5. Usamos `.then()` para manejar el caso de éxito y `.catch()` para manejar errores.
6. Imprimimos "Esperando datos..." inmediatamente después de llamar a `obtenerDatos()`.

Pregunta: ¿Cuáles son las ventajas de usar promesas en lugar de callbacks?

3.3 Actividad

Refactoriza el reto de callbacks del Módulo 2 para usar promesas en lugar de callbacks.

Módulo 4: Async/Await

4.1 ¿Qué es async/await?

Async/await es una sintaxis que hace que las promesas sean más fáciles de escribir y leer. `async` define una función asíncrona y `await` pausa la ejecución hasta que una promesa se resuelva.

4.2 Ejemplo práctico

```
async function obtenerYProcesarDatos() {
  try {
    const datos = await obtenerDatos();
    console.log("Datos obtenidos:", datos);
    const resultado = await procesarDatos(datos);
    console.log("Resultado:", resultado);
  } catch (error) {
    console.error("Error:", error);
  }
}

function obtenerDatos() {
  return new Promise(resolve => {
    setTimeout(() => resolve({ id: 1, nombre: "Ana" }), 2000);
  });
}

function procesarDatos(datos) {
  return new Promise(resolve => {
    setTimeout(() => resolve(`Datos de ${datos.nombre} procesados`), 1000);
  });
}

obtenerYProcesarDatos();
console.log("Iniciando proceso...");
```

Explicación del código:

1. Definimos una función asíncrona `obtenerYProcesarDatos` usando la palabra clave `async`.
2. Dentro de esta función, usamos `await` para esperar la resolución de las promesas.
3. Usamos un bloque `try/catch` para manejar errores.
4. Las funciones `obtenerDatos` y `procesarDatos` devuelven promesas.
5. Llamamos a `obtenerYProcesarDatos()` e inmediatamente después imprimimos "Iniciando proceso...".

Pregunta: ¿Cómo crees que async/await mejora la legibilidad del código asíncrono?

Reflexión final: ¿Cómo crees que el asincronismo mejora el rendimiento y la experiencia del usuario en aplicaciones web? ¿Puedes pensar en ejemplos de la vida real donde se utilice programación asíncrona?