Resumen de Asincronismo en JavaScript

1. Introducción al Asincronismo

El asincronismo en JavaScript permite que múltiples tareas se gestionen sin bloquear la ejecución del programa. Esto se logra mediante el **Event Loop**, que permite que el código continúe ejecutándose mientras se esperan respuestas de operaciones de larga duración como peticiones de red, acceso a archivos o temporizadores.

Concepto de asincronismo

- En lenguajes tradicionales como Java, el código se ejecuta de manera secuencial (síncrona).
- JavaScript adopta un enfoque **asíncrono**, delegando tareas y permitiendo que el programa siga ejecutándose sin esperar que cada tarea finalice.

---

2. JavaScript es Single-Threaded y el Event Loop

JavaScript se ejecuta en un solo hilo (**single-threaded**), lo que significa que solo puede ejecutar una tarea a la vez. Sin embargo, el **Event Loop** permite gestionar múltiples tareas asíncronas sin bloquear el hilo principal.

Funcionamiento del Event Loop

1. JavaScript ejecuta primero el código principal de manera síncrona.
2. Las tareas asíncronas se delegan al entorno del navegador o Node.js.
3. Cuando terminan, se envían a la **Task Queue**.
4. El **Event Loop** verifica si la **Call Stack** está vacía y, si es así, ejecuta las tareas pendientes.

---

3. Diferencia entre Código Bloqueante y No Bloqueante

🔘 **Código bloqueante:** Detiene la ejecución del programa hasta que una tarea finaliza.
🔘 **Código no bloqueante:** Permite que el programa siga ejecutando otras tareas mientras espera la respuesta.

📌 **Ejemplo de código bloqueante:**

```
js
CopiarEditar
while (Date.now() - start < 2000) {} // Bloquea la ejecución durante 2 segundos
```

📌 **Ejemplo de código no bloqueante:**

```
js
CopiarEditar
setTimeout(() => console.log("Esto se ejecuta después"), 2000);
```

---

4. Callbacks: Primera Solución al Asincronismo

Un **callback** es una función que se pasa como argumento a otra función y se ejecuta cuando la tarea asíncrona finaliza.

📌 **Ejemplo de Callback:**

```
js
CopiarEditar
function obtenerDatos(callback) {
  setTimeout(() => {
    const datos = { nombre: "Usuario", edad: 30 };
    callback(datos);
  }, 2000);
}
obtenerDatos((datos) => console.log("Datos recibidos:", datos));
```

🔴 **Problema: Callback Hell**
Cuando los callbacks se anidan demasiado, el código se vuelve difícil de leer y mantener.

📌 **Ejemplo de Callback Hell:**

```
js
CopiarEditar
paso1(() => {
  paso2(() => {
    paso3(() => {
      console.log("Finalizado");
    });
  });
});
```

👉 **Solución:** Usar **Promesas** o **Async/Await**.

---

5. Promesas: Alternativa a los Callbacks

Las **Promesas** representan operaciones asíncronas con tres estados:
✅ *Pending (pendiente)*
✅ *Fulfilled (cumplida)*
✅ *Rejected (rechazada)*.

📌 **Ejemplo de Promesa:**

```
js
CopiarEditar
function obtenerDatosPromesa() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const datos = { nombre: "Usuario", edad: 30 };
      resolve(datos);
    }, 2000);
  });
}

obtenerDatosPromesa()
  .then((datos) => console.log("Datos recibidos:", datos))
  .catch((error) => console.error("Error:", error));
```

---

6. Async/Await: Mejorando la Legibilidad

`async` y `await` permiten escribir código asíncrono con una sintaxis más clara y fácil de entender.

📌 **Ejemplo de Async/Await:**

```
js
CopiarEditar
async function obtenerDatos() {
  try {
    let respuesta = await fetch("https://jsonplaceholder.typicode.com/posts/1");
    let datos = await respuesta.json();
    console.log(datos);
  } catch (error) {
    console.error("Error:", error);
  }
}
obtenerDatos();
```

✅ **Ventajas de Async/Await:**

- Evita el Callback Hell.
- Hace que el código sea más limpio y fácil de leer.
- Manejo de errores más sencillo con `try/catch`.

---

7. Diferencias entre Microtareas y Macrotareas

El **Event Loop** maneja diferentes tipos de tareas en distintas colas:
✅ **Macrotareas:** `setTimeout`, `setInterval`, eventos del DOM, tareas de red.
✅ **Microtareas:** Promesas resueltas, `MutationObserver`. Se ejecutan **antes** que las macrotareas.

📌 **Ejemplo del orden de ejecución:**

```
js
CopiarEditar
console.log("Inicio");
setTimeout(() => console.log("Macrotarea: setTimeout"), 0);
Promise.resolve().then(() => console.log("Microtarea: Promesa"));
console.log("Fin");

// Salida: Inicio → Fin → Microtarea → Macrotarea
```

---

8. Web Workers: Ejecutando Código en Paralelo

Los **Web Workers** permiten ejecutar scripts en hilos separados para evitar bloquear la interfaz de usuario.

📌 **Ejemplo de Web Worker:**

```
js
```

```
CopiarEditar
// worker.js
self.onmessage = function(e) {
  let resultado = e.data * 2;
  self.postMessage(resultado);
};

// script.js
let worker = new Worker("worker.js");
worker.postMessage(5);
worker.onmessage = function(e) {
  console.log("Resultado del Worker:", e.data);
};
```

---

9. Mejoras en el Manejo de Errores Asíncronos

🔹 **Promesas:** Manejan errores con `.catch()`.

🔹 **Async/Await:** Usa `try/catch` para capturar excepciones.

📌 **Ejemplo de manejo de errores con Async/Await:**

```
js
CopiarEditar
async function obtenerDatosSeguros() {
  try {
    let respuesta = await fetch("https://jsonplaceholder.typicode.com/posts/1");
    let datos = await respuesta.json();
    console.log(datos);
  } catch (error) {
    console.error("Error al obtener los datos:", error);
  }
}
obtenerDatosSeguros();
```

---

10. Conclusión

El **asincronismo en JavaScript** es clave para escribir código eficiente.

✅ **Promesas y Async/Await** han mejorado la forma de manejar código asíncrono.

✅ **El Event Loop** permite gestionar tareas sin bloquear el hilo principal.

✅ **Comprender la diferencia entre microtareas y macrotareas** optimiza el rendimiento de las aplicaciones.