ASINCRONISMO EN JAVASCRIPT

TEMA 01 – INTRODUCCIÓN AL ASINCRONISMO EN JAVASCRIPT

1.1. Introducción al asincronismo en JavaScript

Una de sus características más relevantes de JavaScript es su modelo de ejecución asíncrono, conocido por permitir que múltiples tareas se gestionen sin bloquear la aplicación mientras se esperan respuestas u operaciones de larga duración.

¿Qué significa “asincronismo” en el contexto de la programación?

El término **asincronismo** hace referencia a la capacidad de un programa de ejecutar ciertas **tareas** en **segundo plano**, permitiendo que el **flujo principal** continúe con la ejecución de otras instrucciones **sin** tener que **detenerse** hasta que se complete la tarea anterior. En un lenguaje de programación tradicionalmente imperativo y secuencial, como podría ser C o Java en su forma más básica, las sentencias se ejecutan de manera **síncrona**: la ejecución de una instrucción depende de que la anterior haya finalizado. Esto implica que, en presencia de operaciones de **E/S** (Entrada/Salida) o conexiones de red que requieren mucho tiempo, el **programa** queda **bloqueado** hasta obtener los resultados.

JavaScript, en cambio, adopta un enfoque diferente: **no se queda bloqueado mientras se espera la finalización de operaciones costosas** (por ejemplo, llamadas a una API externa, lecturas de archivos de disco, consultas a bases de datos, etc.). En lugar de ello, se apoya en un mecanismo de gestión de tareas que **permite registrar** las **operaciones asíncronas** y **continuar** con la ejecución del código **sin bloqueos**. Al completarse dichas operaciones, se activa una función de devolución de llamada (**callback**), o se resuelve una Promesa (**Promise**), posibilitando el

procesamiento de los resultados cuando están disponibles, sin haber entorpecido el rendimiento del resto de la aplicación.

Este modelo de asincronismo es clave para el desarrollo de aplicaciones web modernas que buscan ofrecer una experiencia de usuario fluida, evitando bloqueos o tiempos de espera prolongados que puedan frustrar la interacción del usuario con la aplicación. En este sentido, JavaScript se ha posicionado como un referente en la creación de interfaces altamente dinámicas y reactivas, donde el asincronismo no solo es una ventaja, sino prácticamente una necesidad.

1.2. Naturaleza de un solo hilo en JavaScript

Un aspecto fundamental para comprender el asincronismo en JavaScript es su naturaleza de un solo hilo (**single-threaded**). En términos generales, un hilo es la secuencia de ejecución de instrucciones dentro de un proceso. Un lenguaje single-threaded dispone de un único hilo principal encargado de ejecutar el código. Esto significa que el intérprete de JavaScript procesa las líneas de código de arriba hacia abajo en un solo flujo.

A diferencia de otros lenguajes o entornos que pueden utilizar múltiples hilos para paralelizar tareas, **JavaScript ejecuta todo su código en un único hilo**. Sin embargo, esto no le impide manejar varias operaciones de manera simultánea, al menos desde la perspectiva del desarrollador. ¿Cómo lo consigue? La clave reside en el **Event Loop** (bucle de eventos).

Event Loop

El **Event Loop** es el **mecanismo interno** que permite coordinar las tareas asíncronas en JavaScript. Este bucle recibe y **gestiona los eventos y tareas pendientes de ser procesados**, uno tras otro, reanudando la ejecución del flujo principal únicamente cuando está listo para ello. En términos simplificados:

1. Se mantiene una **cola de tareas** en la que se registran todos los **callbacks**, **promesas** u **operaciones asíncronas** que están **pendientes**.

2. **JavaScript ejecuta, en primer lugar, el código principal de manera síncrona**, línea a línea, hasta que se completa o encuentra una operación asíncrona.

3. Cuando se desencadena una **operación asíncrona** (por ejemplo, una petición HTTP), esta **se delegará** al entorno externo (como las APIs del navegador), para que se procese.

4. Mientras tanto, **el flujo principal no se bloquea**: continúa ejecutando las siguientes líneas de código.

5. Una vez **finalizada la operación asíncrona**, **se añade a la cola de tareas**. El **Event Loop comprueba si el hilo principal está libre** y, cuando lo está, **retoma la tarea pendiente**, ejecutando el callback o resolviendo la Promesa.

Gracias a este proceso, JavaScript puede resultar muy eficiente al coordinar tareas que involucran E/S o consultas remotas sin que la aplicación quede bloqueada.

1.3. Callbacks, promesas y async/await

A continuación, se presenta una breve introducción a los callbacks, las promesas y la sintaxis async/await en JavaScript, centrada en la importancia de cada uno de estos elementos dentro de la programación asíncrona. Dado que estamos en una etapa inicial dentro del asincronismo en JavaScript, no se profundizará ahora en ninguno de estos aspectos, dejando esta tarea para más adelante, en documentos dedicados.

Callbacks

Un *callback* (o función de retorno) es una función que se pasa como parámetro a otra función y se ejecuta una vez que la tarea principal ha concluido. Históricamente, los callbacks han sido la forma más habitual de manejar la asincronía en JavaScript, especialmente antes de que existieran las promesas y la sintaxis async/await.

Ventajas

• Permiten seguir ejecutando código mientras se espera la respuesta de una operación (por ejemplo, una petición a una API externa).

• Son relativamente sencillos de utilizar para operaciones pequeñas.

Desventajas

• Cuando se encadenan varios callbacks (por ejemplo, varias llamadas asíncronas consecutivas), puede aparecer lo que se conoce como *callback hell*, dificultando la lectura y mantenimiento del código.

Ejemplo básico de callback

Imagina que deseas simular la obtención de datos desde un servidor, y una vez que los tienes, imprimes esos datos en la consola.

```
function obtenerDatos(accionAlTerminar) {
  console.log("Obteniendo datos del servidor...");
  // Simulamos el retardo en la obtención de datos con setTimeout
  setTimeout(() => {
    const datos = { nombre: "Alumno", curso: "Desarrollo Web" };
    // Una vez obtenidos los datos, ejecutamos el callback
    accionAlTerminar(datos);
  }, 2000);
}
function imprimirDatos(datos) {
  console.log("Datos recibidos:", datos);
}
// Llamamos a la función principal y le pasamos el callback
obtenerDatos(imprimirDatos);
```

En este ejemplo, la función obtenerDatos recibe una función de callback llamada accionAlTerminar que se ejecuta cuando los datos ya están

listos. Este método es funcional pero, como verás más adelante, puede resultar poco legible cuando se tienen muchas tareas encadenadas.

2. Promesas (Promises)

Las *promesas* llegaron para aportar una forma más clara y organizada de manejar la asincronía. Una promesa es un objeto que representa la finalización exitosa o el fracaso de una operación asíncrona. Existen tres estados básicos en una promesa:

1. ***Pending*** (pendiente): la promesa está en proceso de ser resuelta.
2. ***Fulfilled*** (cumplida): la operación terminó satisfactoriamente.
3. ***Rejected*** (rechazada): la operación falló.

Ventajas

• Ofrecen una sintaxis más clara respecto a los callbacks anidados.

• Facilitan el manejo de errores, gracias a los métodos `then()` y `catch()`.

• Permiten encadenar múltiples operaciones asíncronas sin caer en el *callback hell*.

Ejemplo básico con promesas

En el siguiente ejemplo, convertimos la simulación de obtener datos en una función que retorna una promesa:

```
function obtenerDatosPromesa() {
  console.log("Obteniendo datos del servidor...");
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const datos = { nombre: "Alumno", curso: "Desarrollo Web" };
      // Simulamos un escenario de éxito
      if (datos) {
        resolve(datos);  // La promesa se cumple
```

```
      } else {
          reject("Error al obtener los datos"); // La promesa se
rechaza
      }
    }, 2000);
  });
}
// Uso de la promesa
obtenerDatosPromesa()
  .then((respuesta) => {
    console.log("Datos recibidos:", respuesta);
  })
  .catch((error) => {
    console.error("Ha ocurrido un error:", error);
  });
```

Aquí, la función *`obtenerDatosPromesa`* retorna una promesa. Si todo va bien, se llama a *`resolve(datos)`* y la promesa pasa a estado fulfilled. Si algo falla, se llama a *`reject(...)`*, y la promesa pasa a estado rejected, ejecutando el método *`catch()`*.

Async/Await

La sintaxis *async/await* es una **forma más reciente** e intuitiva de manejar las operaciones asíncronas. Básicamente, *`async`* **marca una función como asíncrona**, y *`await`* **hace que la ejecución se detenga hasta que la promesa se resuelva o se rechace,** permitiéndonos escribir código que se lee como si fuese sincrónico.

Ventajas

- Código más legible y fácil de seguir.
- Mantenimiento simplificado, sobre todo en proyectos grandes.

• Mejor manejo de errores gracias al uso de try/catch dentro de funciones asíncronas.

Ejemplo básico con async/await

Tomando la misma función *obtenerDatosPromesa*, podemos consumirla usando async/await:

```
// Función que retorna una promesa
function obtenerDatosPromesa() {
  console.log("Obteniendo datos del servidor...");
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const datos = { nombre: "Alumno", curso: "Desarrollo Web" };
      // Simulamos un escenario de éxito
      if (datos) {
        resolve(datos); // La promesa se cumple
      } else {
          reject("Error al obtener los datos"); // La promesa se
rechaza
      }
    }, 2000);
  });
}
// Función asíncrona que utiliza la promesa
async function mostrarDatos() {
  try {
    console.log("Llamando a la función obtenerDatosPromesa...");
    const respuesta = await obtenerDatosPromesa();
    console.log("Datos recibidos:", respuesta);
  } catch (error) {
    console.error("Ha ocurrido un error:", error);
  }
}
```

```
// Llamada a la función asíncrona
mostrarDatos();
```

En este ejemplo, la palabra clave *`await`* pausa la ejecución hasta que la promesa se resuelva o rechace. Si la promesa se cumple, la variable respuesta obtiene el valor devuelto por *`resolve(datos)`*. Si se produce un error, se maneja en el bloque catch del *`try/catch`*.

Conclusión

1. **Callbacks**: fueron la primera herramienta para manejar la asincronía; funcionan bien en situaciones simples, pero pueden producir problemas de legibilidad con operaciones más complejas.

2. **Promesas**: ofrecen un mejor control de flujo y manejo de errores que los callbacks. Presentan una forma más clara de encadenar varias operaciones asíncronas.

3. **Async/Await**: mejora la legibilidad y el control de los flujos asíncronos, haciendo que el código se aproxime mucho al estilo sincrónico, aunque por debajo siguen funcionando con promesas.

1.4. Ejemplos

A continuación se presentan ejemplos finales de los conceptos tratados en este documento introductorio.

Ejemplo 1: Saludo retardado con setTimeout

Este primer ejemplo muestra cómo una función se ejecuta de forma diferida después de un intervalo de tiempo, sin detener el flujo principal del programa.

```
console.log("Mensaje 1: Inicio del programa.");

setTimeout(function() {
  console.log("Mensaje 2: Este saludo aparece tras 2 segundos.");
}, 2000);
```

```
console.log("Mensaje 3: Fin del programa (se ejecuta antes del saludo retrasado).");
```

1. El intérprete de JavaScript ejecuta la línea *`console.log("Mensaje 1...");`* inmediatamente y muestra *“Mensaje 1: Inicio del programa.”*.
2. Encuentra la función *`setTimeout(...)`*. Ésta agenda la ejecución de la función callback dentro de 2 segundos, pero **no bloquea** la ejecución de las siguientes líneas.
3. Mientras pasan esos 2 segundos, el hilo principal continúa, y se muestra *“Mensaje 3: Fin del programa (se ejecuta antes del saludo retrasado).”*.
4. Una vez transcurridos los 2 segundos, el **Event Loop** inserta la tarea pendiente en la cola de eventos, y finalmente se ejecuta la función callback, mostrando *“Mensaje 2: Este saludo aparece tras 2 segundos.”*.

Con ello, se ilustra cómo JavaScript, pese a trabajar con un único hilo, permite coordinar tareas de forma asíncrona sin detener el flujo de ejecución.

Ejemplo 2: Cálculo en segundo plano con setTimeout y funciones callback

Aunque JavaScript es single-thread, a menudo necesitamos realizar operaciones de cierta complejidad (por ejemplo, cálculos aritméticos intensivos) sin bloquear la interfaz de usuario o el flujo principal. En este ejemplo, se simula un cálculo intensivo usando *`setTimeout`* para pasar la tarea a segundo plano, permitiendo a la aplicación seguir funcionando.

```
function calcularFactorial(num, callback) {
  // Función recursiva para calcular el factorial
  function factorial(n) {
    if (n <= 1) return 1;
    return n * factorial(n - 1);
  }
```

```
  console.log("Iniciando cálculo factorial de", num);
  // Simula una tarea compleja en segundo plano
  setTimeout(function() {
    const resultado = factorial(num);
    // Una vez calculado, se ejecuta el callback con el resultado
    callback(resultado);
  }, 0);
}
// Uso de la función 'calcularFactorial'
calcularFactorial(10, function(resultado) {
  console.log("El factorial es:", resultado);
});
// Mientras tanto, el programa continúa...
console.log("El programa sigue ejecutándose sin esperar el cálculo...");
```

1. La función *`calcularFactorial`* recibe un número y un callback.
2. Dentro de *`calcularFactorial`*, se realiza la llamada a *`setTimeout`* con un tiempo de retardo de 0 milisegundos. Aunque parezca que es “inmediato”, en realidad el callback de setTimeout se **agenda** para la siguiente iteración del Event Loop, permitiendo a JavaScript continuar ejecutando otras tareas mientras tanto.
3. El hilo principal registra la llamada a *`setTimeout`* y continúa con la ejecución, por lo que se muestra en consola el mensaje *`“El programa sigue ejecutándose…”`*.
4. Cuando el Event Loop detecta que ya puede atender la tarea pendiente, ejecuta la función callback que calcula el factorial y muestra el resultado en la consola.

Este patrón es muy útil para fraccionar operaciones de alto coste computacional, evitando “congelar” la interfaz o bloquear otros procesos durante la ejecución.

Ejemplo 3: Obteniendo datos de una API con fetch

En este último ejemplo, se muestra cómo realizar una petición HTTP asíncrona para obtener datos de un servicio externo (una API), utilizando la función `fetch()`. Se utiliza la API pública de JSON Placeholder para demostrar la operación.

```
// URL de la API pública
const apiURL = "https://jsonplaceholder.typicode.com/posts/1";

console.log("Iniciando petición a la API...");

fetch(apiURL)
  .then(function (response) {
    // Se recibe la respuesta y se convierte a formato JSON
    return response.json();
  })
  .then(function (data) {
    // Cuando se dispone de los datos en formato objeto JS, se muestran en consola
    console.log("Datos recibidos:", data);
  })
  .catch(function (error) {
    // En caso de error en la conexión u otro problema, se captura aquí
    console.error("Error al obtener los datos:", error);
  });

console.log("La petición se ha iniciado, el programa sigue con otras tareas...");
```

1. La llamada a `fetch(apiURL)` se ejecuta de inmediato; sin embargo, **no se bloquea** el hilo principal mientras se espera la respuesta del servidor.

2. El método `then()` se encarga de procesar la respuesta cuando llegue. JavaScript almacena internamente una promesa que se resolverá al completar la petición.

3. Mientras tanto, el resto del código sigue ejecutándose, de modo que el mensaje `"La petición se ha iniciado…"` aparece inmediatamente en la consola, **antes** de que los datos de la API estén disponibles.

4. Cuando el servidor devuelve la respuesta, el Event Loop programa la ejecución de la función callback correspondiente en la cola de tareas. Con esto, finalmente se llama a `then(function(data) {...})`, donde ya se manejan los datos devueltos.

5. Si ocurre algún error (fallo de red, URL incorrecta, etc.), el flujo salta al catch, capturando así la excepción de la promesa rechazada.

Este ejemplo demuestra de forma muy clara cómo se gestiona de manera asíncrona la comunicación con servicios externos.

Conclusiones

Estos tres ejemplos ilustran cómo JavaScript gestiona de forma asíncrona operaciones que tardan un tiempo en completarse (esperar un intervalo de tiempo, recibir datos de una API o efectuar un cálculo costoso). Observa que en todos los casos:

- La ejecución principal no se detiene.
- El **Event Loop** se encarga de reasignar las tareas pendientes una vez que estén listas.
- Se usan mecanismos como **callbacks**, **promesas** y funciones como **setTimeout** y **fetch** para coordinar el trabajo en segundo plano.