CAPITULO NRO. 15 – ASINCRONISMO EN JAVASCRIPT

Tipo de Clase: Teórico Práctica

Duración: 120 minutos

Objetivo de la clase:

Al finalizar el presente capítulo, los alumnos deberán comprender que JavaScript es un lenguaje básicamente asíncrono, la forma de trabajar del Navegador es totalmente asíncrona. En determinadas ocasiones necesitamos tener un sincronismo en esta ejecución y terminación de los procesos, es por ello que en este capítulo explicaremos el funcionamiento de los procesos y como lograr un sincronismo, esto significa que si ejecuto un proceso1 la ejecución debe esperar a que termine el proceso1 para continuar con el proceso2 y este debe esperar para continuar con un tercer proceso por dar un ejemplo.

Existen tres formas o técnicas con las que se pueden conseguir que los procesos sean síncronos y para ello veremos las funciones CallBack, las Promesas y los procesos Async await.

Introducción al concepto de Asincronismo:

el asincronismo en JavaScript es un concepto fundamental que permite que el código se ejecute de manera no secuencial, permitiendo que ciertas operaciones se realicen mientras otras están en progreso. Esto es especialmente útil en el contexto de la programación web, donde es común realizar tareas que pueden tomar tiempo, como la obtención de datos de un servidor, sin bloquear la ejecución del resto del código.

Diferencias entre procesos síncronos y asíncronos

Síncrono: Las tareas se ejecutan secuencialmente. Cada tarea debe completarse antes de que la siguiente pueda comenzar.
Asíncrono: Las tareas pueden comenzar y completar en diferentes momentos, sin esperar a que otras tareas terminen.
Event Loop: Es el mecanismo que maneja las operaciones asíncronas en JavaScript. Monitorea la cola de tareas y ejecuta las que están listas cuando el stack de ejecución está vacío.

Eventos y el Event Loop:

JavaScript utiliza un único hilo de ejecución, pero puede manejar múltiples operaciones de manera no bloqueante gracias al event loop. El event loop es un mecanismo que permite que el JavaScript maneje operaciones asincrónicas al sacar elementos de la cola de tareas y ejecutarlos cuando la pila de llamadas está vacía.

Que cosas son síncronas en JavaScript

- estructuras condicionales
- estructuras repetitivas
- Operaciones de comparación

Que cosas son asíncronas en JavaScript

- Los Eventos, por ejemplo “click”,”load”, etc. Y todo lo que se ejecute dentro de esos eventos es totalmente asíncrono.

Los eventos en JavaScript, especialmente en el contexto del navegador web, son inherentemente asíncronos. Cuando ocurre un evento, como hacer clic en un botón o cargar una página, el navegador coloca ese evento en la cola de eventos y continúa ejecutando el código. El código relacionado con ese evento se ejecutará más tarde, cuando el navegador decida manejar ese evento de la cola.

Por lo tanto, los eventos son asíncronos en el sentido de que el código relacionado con ellos se ejecuta en un momento posterior, después de que haya ocurrido el evento y el navegador decida manejarlo. Esta naturaleza asíncrona de los eventos es fundamental para construir aplicaciones web interactivas y receptivas.
podríamos decir que JavaScript, especialmente en el entorno del navegador, tiende a ser asíncrono por defecto debido a su naturaleza orientada a eventos. Esto significa que las operaciones en JavaScript no bloquean el hilo principal de ejecución, lo que permite que otras tareas se ejecuten mientras se esperan ciertas operaciones, como las operaciones de red o de temporización.

Para manejar esta asincronía y controlar el flujo de ejecución de manera más eficiente, JavaScript ofrece diferentes mecanismos:

Callbacks: Los callbacks son funciones que se pasan como argumentos a otras funciones y se ejecutan cuando se completa una tarea, como una operación de red o temporización.

Promesas: Las promesas son objetos que representan el resultado eventual (éxito o fracaso) de una operación asíncrona. Permiten un manejo más flexible y legible de las operaciones asíncronas y evitan el "callback hell".

Async/await: La combinación de las palabras clave async y await permite escribir código asíncrono de manera más similar a la sincrónica. Las funciones async devuelven promesas y el operador await pausa la ejecución de una función async hasta que una promesa sea resuelta.

Estos mecanismos proporcionan formas de gestionar la asincronía y, cuando es necesario, forzar cierto grado de sincronismo en el flujo de ejecución de JavaScript. Permiten escribir código más claro, legible y mantenible al manejar operaciones asíncronas de manera más estructurada y fácil de seguir.

El asincronismo en JavaScript es una característica clave que permite que las operaciones no bloqueen la ejecución del programa, haciendo posible la realización de múltiples tareas al mismo

tiempo. Aquí hay un resumen de los conceptos más importantes relacionados con el asincronismo en JavaScript.

Técnicas y Herramientas para el Asincronismo

1 – LAS FUNCIONES CALLBACK

Las funciones callback son una característica fundamental en JavaScript que permite manejar la asincronía y la ejecución de código dependiente de eventos o procesos asíncronos

¿Qué es una función callback?

Una función callback es simplemente una función que se pasa como argumento a otra función y se invoca después de que cierto proceso o evento ha ocurrido. Es una forma de asegurarse de que cierto código se ejecute únicamente cuando una tarea asíncrona haya sido completada o cuando ocurra algún evento.

Características principales:

Argumento de función: En JavaScript, las funciones son de primera clase, lo que significa que pueden ser tratadas como cualquier otro tipo de dato, incluyendo ser pasadas como argumentos a otras funciones.

Asincronía: Las funciones callback son esenciales para manejar operaciones asíncronas como las que involucran peticiones HTTP, temporizadores (setTimeout, setInterval), manejo de eventos (click, load, submit, etc.), entre otros.

Uso común:

Eventos de usuario: Por ejemplo, en una página web, una función callback puede ser utilizada para manejar la respuesta después de que el usuario haga clic en un botón.

Peticiones HTTP: Al realizar una solicitud a un servidor, se puede proporcionar una función callback que procese la respuesta una vez que se reciba.

Ejemplos:

Ejemplo 01 – Dos procesos Asíncronos Independientes: En este ejemplo visualizaremos la problemática de lanzar dos procesos ASÍNCRONOS de forma simultánea para observar como terminan ambos en momentos diferentes no siguiendo la línea de ejecución

Ejemplo 02 – Funciones CallBack: En este segundo ejemplo, mostraremos como sería una solución simple al problema del asincronismo y para ello recurrimos a una función callback que es llamar una función dentro de la otra. Cuando el primer proceso haya terminado.

Ejemplo 03 – Funciones CallBack II: En el tercer ejemplo, adaptaremos la función llamadora para recibir como parámetro cualquier función que se desea ejecutar posteriormente. es decir, la

función que inicia el proceso recibe como parámetro la siguiente función a ejecutar, es más flexible que el ejemplo anterior.

Ejemplo 04 – Funciones CallBack III: En este cuarto ejemplo, la función llamadora recibe como parámetro la próxima función a ejecutar, pero el cambio más importante es mostrar que la función callBack al momento de ser invocada, puede definirse como una arrow function, como una función anónima.

Ejemplo 05 – Funciones CallBack IV: En este quinto ejemplo, la función principal que es la que inicia el proceso, no tan solo llama a la función CALLBACK sino también le pasa resultados.

2 – PROMESAS = Promis

Las promesas son objetos en JavaScript que representan la eventual finalización o el fracaso de una operación asincrónica. Proporcionan una forma más estructurada y flexible de trabajar con código asíncrono en comparación con las funciones callback tradicionales.

Características principales:
Estado: Una promesa puede estar en uno de tres estados: pendiente, cumplida (resuelta) o rechazada.

Encadenamiento: Permite encadenar operaciones asincrónicas de manera más legible y mantenible utilizando los métodos .then() y .catch().

Manejo de errores: Proporciona un manejo más robusto de errores a través del método .catch() para capturar cualquier excepción que ocurra durante la ejecución de la promesa.

Uso común:
Peticiones HTTP: Al realizar solicitudes a servidores, las promesas son utilizadas para manejar la respuesta de manera asíncrona.

Carga de archivos: Cuando se carga un archivo de manera asíncrona, una promesa puede manejar la finalización o el error de esa operación.

Animaciones y temporizadores: Pueden utilizarse para manejar animaciones y esperas asíncronas en interfaces de usuario.

Beneficios:
Claridad: Facilitan la escritura y lectura de código asincrónico, reduciendo la anidación de callbacks y mejorando la estructura del código.

Manejo de errores: Ofrecen un método centralizado para manejar errores, lo que mejora la robustez del código y facilita la depuración.

Consideraciones:
Compatibilidad: Las promesas son compatibles con la mayoría de los navegadores modernos, pero es recomendable verificar la compatibilidad en entornos específicos si se requiere soporte para versiones más antiguas.
Encadenamiento adecuado: Para evitar el callback hell, es importante utilizar el encadenamiento de promesas de manera efectiva y utilizar async/await para estructuras más complejas.

A continuación, daremos varios ejemplos

Ejemplo 07 – Promesas – Ejemplo I: En este simple ejemplo mostraremos como utilizar e instanciar el objeto promesa a partir de la clase PROMESA, como guardarlo en una constante y posteriormente capturar el caso de éxito ("resolve") y el caso de fallo ("reject"). El objetivo de este ejemplo es observar cómo se instancia la promesa y como se capturan sus casos de éxito o fracaso.

Ejemplo 08 – Promesas – Ejemplo II: En este ejemplo, en lugar de instanciar un objeto Promesa y guardarlo en una constante, lo que realizaremos es construir una función que retornará una promesa ya instanciada, esta es la forma más utilizada de cómo implementar las promesas, y tiene como objetivo organizar mejor una cadena de promesas enganchadas unas de otras.

Ejemplo 09 – Promesas – Ejemplo III: En este ejemplo, la función que retorna la promesa instanciada, tiene la flexibilidad de recibir como parámetro la función a ejecutar dentro de la promesa.

Ejemplo 10 – Promesas – Ejemplo IV: En este ejemplo, la función que retorna la promesa instanciada, se utiliza para enganchar dos procesos asíncronos que se desean ejecutar de forma síncrona. Por el lado del .then() de la primera promesa, se debe colocar siempre, la palabra **RETURN e invocar nuevamente la promesa siguiente.**

3 – Async & Await

La palabra clave async:
se usa para declarar una función asíncrona. Una función marcada con async siempre devolverá una promesa.

await
La palabra clave await se usa dentro de las funciones asíncronas para esperar a que una promesa se resuelva. Cuando await se encuentra una promesa, la ejecución de la función se pausa hasta que la promesa se resuelve, y luego continúa con el valor resuelto.

Beneficios de async y await

Legibilidad mejorada: El código se lee de manera más secuencial y lógica, similar al código síncrono tradicional.