

## Contenidos a Trabajar

1. ¿Qué es el Asincronismo?
2. ¿Cómo funciona el Single Thread?
  - a. Call Stack
  - b. Event Loop
  - c. Callback queue
3. Manejo de asincronismo en Javascript
  - a. Callbacks
  - b. Promesas
  - c. Async/Await
4. Conclusión

## ¿Qué es el Asincronismo?

El asincronismo es un paradigma de programación que permite que una tarea pueda ejecutarse en segundo plano sin bloquear la ejecución del programa principal.

En otras palabras, se pueden realizar varias tareas simultáneamente, sin necesidad de esperar a que una tarea se complete antes de comenzar la siguiente. El asincronismo es importante porque nos permite construir aplicaciones más eficientes y más rápidas.

## ¿Cómo funciona en JavaScript?

JavaScript del lado del cliente es un lenguaje de programación basado en eventos, lo que significa que el código se ejecuta en respuesta a eventos que ocurren en el entorno. El asincronismo en JavaScript se logra mediante la utilización de callbacks, promesas, y `async/await`.

Es importante manejar el asincronismo en JavaScript para permitir que las tareas que tardan mucho tiempo en ejecutarse se realicen en segundo plano sin bloquear la ejecución del programa principal. Si no se maneja el asincronismo adecuadamente, se corre el riesgo de que el programa falle o incluso se bloquee por completo, lo que puede afectar negativamente la experiencia del usuario.



JavaScript es un lenguaje de programación de un solo subproceso (single-threaded), lo que significa que todas las tareas deben ejecutarse en una sola secuencia de comandos. A diferencia de otros lenguajes de programación como Java o C#, que pueden ejecutar múltiples subprocesos simultáneamente, JavaScript solo puede ejecutar una tarea a la vez.

Sin embargo, aunque JavaScript es de un solo subproceso, puede manejar múltiples tareas asíncronas simultáneamente mediante el uso de mecanismos como el Event Loop y las promesas. El Event Loop se encarga de manejar la cola de tareas, lo que permite que las tareas asíncronas se ejecuten en segundo plano mientras se sigue ejecutando el programa principal.



## ¿Cómo funciona el Single Thread?

JavaScript maneja el Single Thread utilizando una estructura de datos llamada Call Stack y técnicas de asincronismo como Callbacks, Promises y Async/Await, junto con el mecanismo de Event Loop, para permitir que el programa realice múltiples tareas en segundo plano sin bloquear la ejecución del programa principal.

Cuando una función se llama, se agrega a la **Call Stack** y se ejecuta de forma **sincrónica**, lo que significa que el programa debe esperar hasta que la función se complete antes de continuar con la siguiente tarea. Si la función tarda mucho tiempo en ejecutarse, puede hacer que el programa se bloquee.

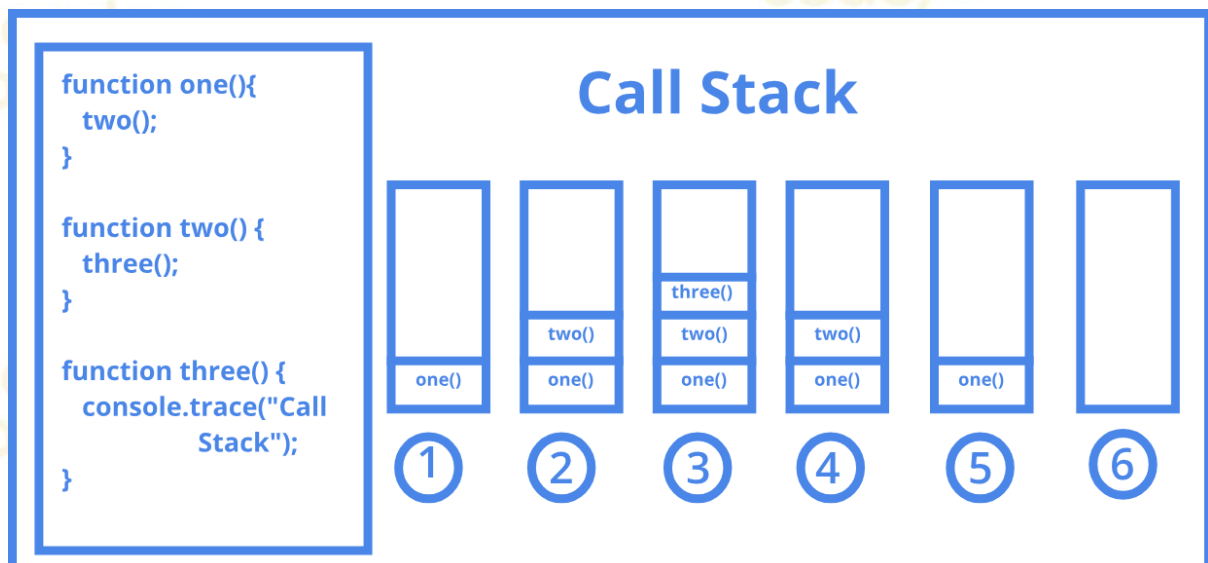
Por eso muchas veces necesitamos crear procesos mediante una **función asíncronica** que espere el resultado mientras el programa se sigue ejecutando.

Esta función se coloca en una cola de tareas llamada Callback Queue, que espera hasta que todas las tareas sincrónicas se completen y la Call Stack esté vacía antes de ejecutar dicha función. Esto se logra mediante el uso de un mecanismo llamado **Event Loop**, que se encarga de monitorear la **Call Stack** y la **Callback Queue** y ejecutar la siguiente tarea en la cola de tareas sólo cuando la **Call Stack** está vacía

## Call Stack

Para entender cómo funciona el asincronismo en JavaScript, es necesario entender algunos conceptos clave. El primero de ellos es **Call Stack** (pila de llamadas), que es una estructura de datos que mantiene un registro de las funciones que están siendo ejecutadas en un momento dado.

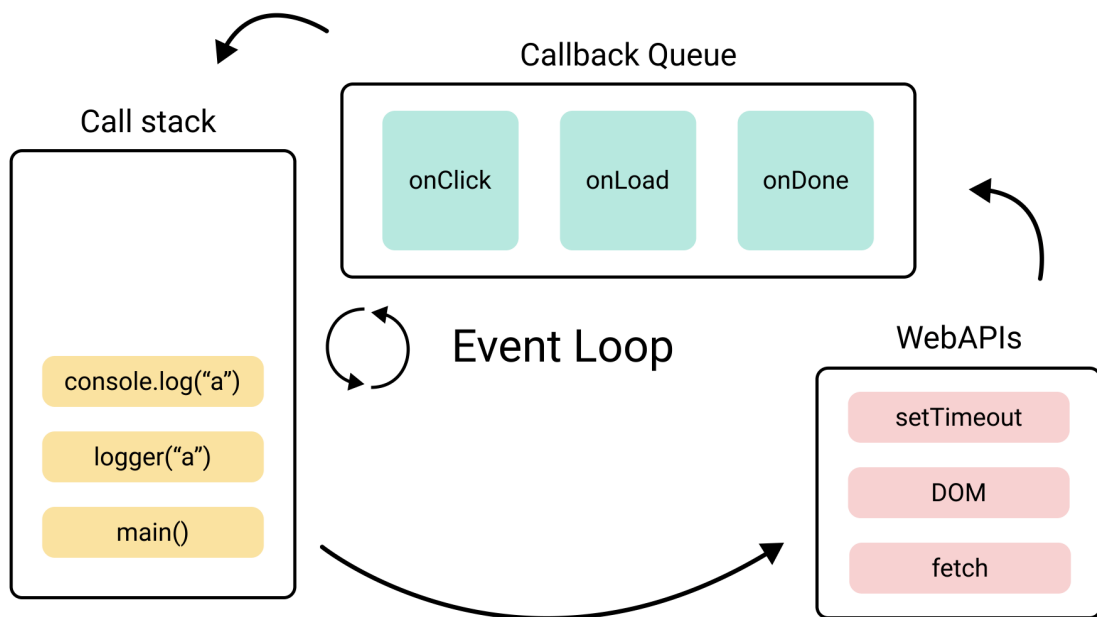
Cuando una función se llama a sí misma, se agrega a la pila de llamadas. Cuando se completa la ejecución de una función, se elimina de la pila de llamadas. La pila de llamadas en JavaScript es importante porque es una estructura de datos que ayuda a controlar el flujo de la ejecución del programa.



## Event Loop

El segundo concepto clave es el Event Loop (bucle de eventos), que es un mecanismo que permite que el código asíncrono se ejecute en segundo plano sin bloquear la ejecución del programa principal. El Event Loop en JavaScript es una estructura que mantiene un registro de los eventos que ocurren en el entorno.

Cuando ocurre un evento, el Event Loop se encarga de manejar ese evento y agregarlo a una cola de callbacks o **Callback Queue**. La cola de callbacks es una estructura de datos que mantiene un registro de las funciones que se deben ejecutar en respuesta a un evento.





## Callback Queue

La **Callback Queue** es una estructura de datos en JavaScript que se utiliza para almacenar las funciones callback (subprocesos relacionados) o resultados que se deben ejecutar o devolver después de que se hayan completado las tareas asíncronas en segundo plano.

Cuando se llama a una función asíncrona en JavaScript, como por ejemplo una solicitud de datos a una fuente externa o una animación en el navegador, la función no se ejecuta de forma sincrónica. En su lugar, se coloca en una cola de tareas llamada **Callback Queue**. Una vez que se han completado todas las tareas sincrónicas y la **Call Stack** está vacía, el **Event Loop** se encarga de ejecutar las funciones en la **Callback Queue** en orden de llegada.

Un ejemplo común de uso de la **Callback Queue** es consulta a una API externa (fuente de datos). Cuando se realiza una dicha solicitud, el programa no espera a que se complete la solicitud antes de continuar ejecutando las demás tareas. En su lugar, la solicitud se realiza en segundo plano y la función de callback se agrega a la Callback Queue. Cuando la solicitud a la API se completa, la función de callback se ejecuta y se procesa la respuesta.

## Manejo de asincronismo en Javascript

Vimos como Javascript maneja procesos asincrónicos internamente a pesar de contar un con solo hilo de ejecución o **"Single Thread"**. En la práctica conocer esto es determinante para saber cuando debemos crear funciones o procesos síncronos o asíncronos.

En esta ocasión aprenderemos a crear estos procesos asíncronos y para ello contamos con diversas herramientas.

### Callbacks

Un callback es una función que se pasa como argumento a otra función y se ejecuta cuando se completa la tarea. El siguiente ejemplo ilustra el uso de un callback:

```
function tareaAsincronica(callback) {  
  setTimeout(function() {  
    console.log('Tarea asincronica completada.');    callback();  
  }, 3000);  
}  
  
console.log('Inicio de la tarea.');tareaAsincronica(function() {  
  console.log('Fin de la tarea.');});
```



En este ejemplo, la función **tareaAsincronica** es una función asíncrona que simula una tarea que tarda 3 segundos en completarse. La función acepta un argumento de callback, que se ejecuta cuando se completa la tarea. En este caso, el callback simplemente muestra un mensaje en la consola indicando que la tarea ha finalizado.

```
Inicio de la tarea.  
Tarea asincronica completada.  
Fin de la tarea.
```

## Promesas

Las promesas son una forma más moderna y elegante de lograr el asincronismo en JavaScript. Una promesa es un objeto que representa el resultado eventual de una tarea asíncrona. El resultado eventual puede ser un valor o un error. Es por eso que las promesas cuentan con dos estados finales:

**Resolved y Rejected.**

Cual de los dos estados tomará como resultado la función se definirá al momento de crear la **Promesa** indicando que devolver si el proceso fue exitoso o erróneo.

Por otra parte, para poder recuperar el resultado de una **Promesa** utilizaremos los métodos **.then()**, **.catch()** y **.finally()**.

Veamos el siguiente ejemplo donde se muestra cómo declarar una promesa:

```
function tareaAsincronica() {  
  return new Promise(function(resolve, reject) {  
    setTimeout(function() {  
      if(Math.random() < 0.5) {  
        resolve('Tarea asincronica completada.');      } else {  
        reject(new Error('Tarea asincronica fallida.));  
      }  
    }, 3000);  
  });  
}
```

En este ejemplo, la función **tareaAsincronica** devuelve una promesa que se resuelve después de 3 segundos. La promesa utiliza los métodos **resolve** y **reject** para indicar si la tarea se completó exitosamente o falló.

En este caso colocamos una condición `Math.random() < 5` para **simular** que la promesa pueda llegar a fallar, ya que de otra forma siempre sería exitosa.

Luego al momento de ejecutar esta función debemos recuperar el resultado pero eso no es posible asignándole la ejecución a una variable, ya que como el programa sigue corriendo, va a leer la línea de código y asignarle a la variable el resultado **"Promise { <pending> }"** dado que la ejecución de ese código asíncrono todavía ha finalizado.

Es por eso que debemos utilizar los métodos anteriormente mencionados, veamos un ejemplo:

```
console.log('Inicio de la tarea.');
```

```
tareaAsincronica()  
  .then(function(resultado) {  
    console.log(resultado);  
  })  
  .catch(function(error) {  
    console.log(error.message);  
  })  
  .finally(function() {  
    console.log('Fin de la tarea.');  });
```

El método **then** establece que debe suceder una vez que la promesa se resuelve con éxito, mientras que el método **catch** determina que sucederá si la promesa falla y el método **then** no se ejecuta. Por último, el método **finally** es **opcional** y se llama al finalizar los dos anteriores, independientemente de si la promesa se resuelve o se rechaza.

\*Nota: El método **catch** también es opcional pero en caso que nuestra función falle no obtendremos un resultado válido o esperado. Por eso siempre es recomendable utilizarlo para manejar y decidir que hacer frente a posibles errores.



## Async/await

El `async/await` es una forma más moderna de lograr el asincronismo en JavaScript que se basa en las promesas.

- El **async** es una palabra clave que se utiliza para definir una función asincrónica.
- El **await** es una palabra clave que se utiliza dentro de una función asincrónica para esperar la resolución de una promesa antes de continuar con la ejecución del código.

Estas palabras claves se utilizan en reemplazo de los métodos **then**, **catch** y **finally** y siempre es recomendable utilizarlas dentro de un bloque de **try/catch** que nos permite ejecutar “cualquier” pieza de código dentro del **try** y capturar un fallo mediante el **catch**.

Veamos cómo quedaría el ejemplo anterior trabajando con **async/await**:

```
async function ejecutarTareaAsincronica() {  
  try {  
    console.log('Inicio de la tarea.');    const resultado = await tareaAsincronica();  
    console.log(resultado);  
  } catch(error) {  
    console.log(error.message);  
  } finally {  
    console.log('Fin de la tarea.');  }  
}  
  
ejecutarTareaAsincronica();
```

A simple vista pueden parecer similares o incluso que tiene las mismas líneas de código pero el valor agregado radica en que con **async/await** podemos ejecutar múltiples funciones asíncronas una debajo de otra y esperar el resultado colocando la palabra **await**. En cambio con la forma tradicional el mismo resultado se lograría concatenando un **.then()** dentro de otro cuando los procesos son dependientes.

## Conclusión

El asincronismo es una parte importante de la programación en JavaScript. Permite que las tareas se ejecuten en segundo plano sin bloquear la ejecución del programa principal y se logra mediante el uso de mecanismos como el Event Loop y las promesas. El **async/await** es una forma moderna y conveniente de lograr el asincronismo en JavaScript y se basa en las promesas. Al comprender el asincronismo y sus mecanismos subyacentes, los desarrolladores pueden escribir programas más eficientes y efectivos en JavaScript.