## Investigación de funciones asíncronas:

Un **callback** es una función que se pasa como argumento a otra función y se ejecuta después de que se complete una tarea. Aunque son útiles, pueden llevar a lo que se llama *callback hell* cuando se anidan múltiples callbacks, lo que dificulta la legibilidad del código.

Las **Promises**, introducidas en ES6, son una mejora sobre los callbacks. Permiten manejar operaciones asíncronas de una manera más legible, ya que evitan la anidación excesiva al proporcionar métodos como .then() y .catch(). Una Promise representa un valor que puede estar disponible ahora, en el futuro o nunca, y tiene tres estados posibles:

* Pending (pendiente).
* Fulfilled (resuelta).
* Rejected (rechazada).

Con la introducción de async/await, que es una sintaxis más moderna, trabajar con Promises se volvió aún más intuitivo.

## Implementación del Proyecto

### Proceso seguido

1. **Diseño inicial**:
   1. Implementé tres botones con diferentes estrategias para manejar tareas asíncronas: **callbacks**, **promises**, y **async/await**.
   2. Diseñé un contenedor (#output) para mostrar los resultados de las operaciones.
2. **Implementación de funciones**:
   1. Usé setTimeout para simular tareas asíncronas que toman tiempo en completarse.
   2. Creé una función basada en **callback** para ejecutar tareas y devolver resultados tras cierto tiempo.
   3. Usé **promises** para resolver tareas y controlar el flujo con .then() y .catch().
   4. Implementé async/await para hacer que el código se vea más limpio y fácil de entender.
3. **Eventos en botones**:
   1. Asigné eventos click para cada botón, ejecutando su respectiva función asíncrona al activarse.

## Conclusión

Este proyecto me permitió comprender mejor la evolución de cómo JavaScript maneja operaciones asíncronas. Usar callbacks fue un buen repaso, pero me di cuenta de lo complicado que puede ser mantener un código limpio y legible con esta técnica. Las Promises ofrecen una solución más organizada, y el uso de async/await simplifica aún más el flujo de trabajo.

La principal dificultad fue entender cómo manejar errores en cada enfoque, especialmente en async/await. Sin embargo, al practicar, logré identificar patrones que facilitan su implementación.