Async and Await

This example async and await

```
function delay(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}
async function fetchData() {
  try {
    console.log('Fetching data...');
    await delay(2000); // Simulating an asynchronous operation
    console.log('Data fetched successfully!');
  } catch (error) {
    console.log('An error occurred:', error);
  }
}
async function process() {
  console.log('Starting the process...');
  await fetchData(); // Waiting for fetchData() to complete
  console.log('Process completed.');
}
process();
```

output is :

```
Starting the process...
Fetching data...
Data fetched successfully!
Process completed.
```

async:)

Asynchronous programming in JavaScript allows you to perform tasks without blocking the execution of other code. This is particularly useful when dealing with operations that take time, such as fetching data from a server, reading/writing files, or making API calls.

In JavaScript, asynchronous behavior is achieved through the use of callbacks, promises, and the async/await syntax.

1- Callbacks: In traditional asynchronous programming, you would pass a callback function to an asynchronous operation, and that function would be invoked once the operation completes. Callbacks can become difficult to manage when dealing with multiple asynchronous operations or when handling errors.

2- Promises: Promises provide a more structured way to handle asynchronous operations. A promise represents a value that may be available now, in the future, or never. It has three states: pending, fulfilled, or rejected. Promises have built-in methods like then() and catch() to handle the resolved value or an error.

3- async/await: Introduced in ES2017, the async/await syntax simplifies working with promises. The async keyword is used to declare an asynchronous function, and the await keyword is used to pause the execution of an async function until a promise is fulfilled or rejected.

await:)

When await is used before a Promise, it pauses the execution of the async function until the Promise is settled. If the Promise is fulfilled, the await expression returns the resolved value of the Promise. If the Promise is rejected, it throws an error, which can be caught using a try...catch block.

Here are a few key points to keep in mind when using await:

1- await can only be used inside an async function: To use await, you need to define an async function. The async keyword is used to declare a function as asynchronous, allowing the use of await inside that function.

2- await pauses the execution of the async function: When await is encountered, it pauses the execution of the function until the awaited Promise is settled. This allows other code outside the await expression to continue executing.

3- await can be used with any object that returns a Promise: await can be used with any expression that returns a Promise, including built-in functions like fetch for making HTTP requests or custom functions that return Promises.

4- Error handling with try...catch: When using await, it's recommended to wrap it in a try...catch block to handle any potential errors. If the awaited Promise is rejected, the error can be caught and handled within the catch block