ANSWER 4-

In JavaScript, there are several approaches to handle asynchronous code and ensure proper execution and coordination. Here are some commonly used techniques:

Callbacks: Callbacks are functions passed as arguments to asynchronous functions. The callback is executed once the asynchronous operation is complete. This approach allows you to specify what should happen after the asynchronous operation finishes. However, working with nested callbacks can lead to callback hell and make the code difficult to read and maintain.

Example:

```
function fetchData(callback) {
  // Simulating an asynchronous operation
  setTimeout(function() {
    const data = "Some data";
    callback(data); // Execute the callback with the fetched data
  }, 1000);
}


fetchData(function(data) {
  console.log(data); // Handle the fetched data
});
```

Promises: Promises provide a more structured way to handle asynchronous code. A Promise represents the eventual completion (or failure) of an asynchronous operation and allows you to chain actions using then and catch methods. Promises can be resolved with a value or rejected with an error. This approach helps to avoid the callback pyramid and provides better error handling.

Example:

```
function fetchData() {
 return new Promise(function(resolve, reject) {
   // Simulating an asynchronous operation
   setTimeout(function() {
     const data = "Some data";
     resolve(data); // Resolve the promise with the fetched data
     // or
     // reject(new Error("Something went wrong")); // Reject the promise with an error
   }, 1000);
 });
}

fetchData()
 .then(function(data) {
   console.log(data); // Handle the fetched data
 })
 .catch(function(error) {
   console.log(error); // Handle errors
 });
```

Async/await: Async/await is a modern approach to handle asynchronous code in a more synchronous-looking manner. It allows you to write asynchronous code using a synchronous-like syntax. The async keyword is used to define an asynchronous function, and the await keyword is used to pause the execution until a promise is resolved or rejected.

Example:

```
function fetchData() {
 return new Promise(function(resolve, reject) {
   // Simulating an asynchronous operation
```

```
    setTimeout(function() {
      const data = "Some data";
      resolve(data); // Resolve the promise with the fetched data
      // or
      // reject(new Error("Something went wrong")); // Reject the promise with an error
    }, 1000);
  });
}

async function handleData() {
  try {
    const data = await fetchData(); // Wait for the promise to resolve
    console.log(data); // Handle the fetched data
  } catch (error) {
    console.log(error); // Handle errors
  }
}

handleData();
```

These techniques provide different ways to handle asynchronous code in JavaScript. Promises and async/await are generally preferred due to their readability, error handling capabilities, and better code organization compared to callbacks.