# CODE EXERCISE

# Asynchronous JavaScript Coding Exercises


***CODING EXERCISES TEST YOUR SKILLS***



## Asynchronous JavaScript Coding Exercises

🌈 Dive Deep into Asynchronous JavaScript with Our Latest Coding Exercises! 🚀

Unlock the power of asynchronous programming in JavaScript with our collection of 10 engaging coding exercises. From mastering callbacks and promises to leveraging the elegant async/await syntax, these exercises are designed to equip you with the skills needed to write efficient, non-blocking JavaScript code.

Perfect for both beginners and seasoned developers, these exercises come with detailed explanations and complete code samples, ensuring a comprehensive learning experience.

#JavaScript #AsynchronousProgramming #WebDevelopment #CodingExercises #AsyncAwait #Promises #TechLearning

Let's make our web applications more responsive and faster by mastering asynchronous JavaScript. Share your thoughts or questions below, and let's engage in a productive discussion. Happy coding! 🎉

## Exercise 1: Basic Callback Function

Objective: Understand how to use a callback function to handle asynchronous operations.

```
function fetchData(callback) {
  setTimeout(() => { // Simulates fetching data from an API
    callback('Data fetched');
  }, 1000);
}
fetchData((data) => {
  console.log(data); // Expected output: Data fetched
});
```

Explanation: Demonstrates the basic usage of a callback function to handle data that is fetched asynchronously.

## Exercise 2: Promise Basics

Objective: Learn to create and use a Promise to handle asynchronous operations.

```
function fetchData() {
  return new Promise((resolve, reject) => {
    setTimeout(() => resolve('Data fetched'), 1000);
  });
}
fetchData().then(data => console.log(data)); // Expected output: Data fetched
```

Explanation: Introduces the concept of Promises as a way to handle asynchronous tasks, using resolve to handle successful operations.

## Exercise 3: Handling Promise Rejection

Objective: Learn to handle errors in Promises using catch.

```
function fetchData() {
  return new Promise((resolve, reject) => {
    setTimeout(() => reject('Error fetching data'), 1000);
  });
}
fetchData().then(data => console.log(data)).catch(error => console.error(error)); //
```

Expected output: Error fetching data

Explanation: Teaches error handling in Promises with catch, providing a mechanism to handle rejected promises.

## Exercise 4: Chaining Promises

Objective: Understand how to chain Promises for sequential asynchronous operations.

```
function firstTask() {
  return new Promise(resolve => setTimeout(() => resolve('First task completed'), 1000));
}
function secondTask() {
  return new Promise(resolve => setTimeout(() => resolve('Second task completed'), 1000));
}
firstTask().then(result => {
  console.log(result);
  return secondTask();
}).then(result => console.log(result));
```

Explanation: Demonstrates chaining multiple promises to ensure that asynchronous operations are completed in sequence.

## Exercise 5: Async/Await Basic

Objective: Use async and await to handle asynchronous operations more intuitively.

```
async function fetchData() {
  return 'Data fetched';
}
```

```
async function run() {
  const data = await fetchData();
  console.log(data); // Expected output: Data fetched
}
run();
```

Explanation: Introduces async/await syntax as a cleaner and more readable way to handle asynchronous operations compared to Promises and callbacks.

## Exercise 6: Error Handling with Async/Await

Objective: Learn to handle errors in asynchronous functions using try/catch.

```
async function fetchData() {
  throw 'Error fetching data';
}
async function run() {
  try {
    const data = await fetchData();
    console.log(data);
  } catch (error) {
    console.error(error); // Expected output: Error fetching data
  }
}
run();
```

Explanation: Teaches error handling in async/await syntax using try/catch blocks to catch and handle errors in asynchronous operations.

## Exercise 7: Using Promise.all

Objective: Execute multiple asynchronous operations in parallel and handle their results together.

```
function fetchData1() {
  return new Promise(resolve => setTimeout(() => resolve('Data 1 fetched'), 1000));
}
function fetchData2() {
  return new Promise(resolve => setTimeout(() => resolve('Data 2 fetched'), 2000));
}
async function run() {
  const [data1, data2] = await Promise.all([fetchData1(), fetchData2()]);
  console.log(data1, data2); // Expected output: Data 1 fetched Data 2 fetched
}
run();
```

Explanation: Shows how to use Promise.all to efficiently handle multiple promises by running them in parallel and waiting for all of them to complete.

## Exercise 8: Async/Await with Fetch API

Objective: Fetch data from an API using async/await and the Fetch API.

```
async function fetchData() {
  const response = await fetch('https://api.example.com/data');
  const data = await response.json();
  console.log(data);
}
```

```
fetchData();
```

Explanation: Combines async/await with the Fetch API to perform asynchronous HTTP requests in a clean and readable way.

## Exercise 9: Custom Async Iterator

Objective: Create a custom asynchronous iterator to iterate over data asynchronously.

```
async function* asyncGenerator() {
  let i = 0;
  while(i < 3) {
    yield await new Promise(resolve => setTimeout(() => resolve(i++), 1000));
  }
}
async function run() {
  for await (let num of asyncGenerator()) {
    console.log(num); // Expected output: 0 1 2
  }
}
run();
```

Explanation: Introduces asynchronous generators and for await...of loops to handle asynchronous iteration, useful for processing streams of data.

## Exercise 10: Handling Multiple Fetch Requests

Objective: Fetch multiple URLs in parallel and process the data once all requests are completed.

```
async function fetchData(urls) {
  const promises = urls.map(url => fetch(url).then(response => response.json()));
  return Promise.all(promises);
}
const urls = ['https://api.example.com/data1', 'https://api.example.com/data2'];
fetchData(urls).then(data => console.log(data));
```

Explanation: Demonstrates handling multiple asynchronous fetch requests in parallel using Promise.all, a common pattern in web development for working with multiple API endpoints.