Asynchronous JavaScript

Mastering Non-Blocking Code

Understanding JavaScript's Execution Flow :

JavaScript is **single-threaded**, meaning it runs one operation at a time. However, many tasks (like network requests, timers, or file I/O) can take time. Asynchronous programming lets your program start these tasks and continue running other code without waiting for those tasks to finish immediately.

1. Synchronous vs. Asynchronous Code

Synchronous Code

- What it means: Each line of code is executed one after the other. No new line starts until the previous one finishes.
- Example:

index.js

```
console.log("Step 1");
console.log("Step 2");
console.log("Step 3");
// Output: Step 1 → Step 2 → Step 3
```

Explanation: The code prints each step in order because each line must finish before the next one begins.

Asynchronous Code

- What it means: Some operations (like waiting for a timer or a network request) are started, and the program continues executing the rest of the code while waiting for those operations to finish.

- Example:

index.js

```
console.log("Step 1");

setTimeout(() => {
  console.log("Step 2");
}, 2000);

console.log("Step 3");
// Output: Step 1 → Step 3 → (after 2 seconds) Step 2
```

Explanation:

- "Step 1" is logged immediately.
- setTimeout schedules "Step 2" to run after 2 seconds.
- "Step 3" is logged immediately after "Step 1" because the timer doesn't block code execution.

2. The Event Loop (A Quick Note)

- What is it? The event loop is JavaScript's mechanism to manage asynchronous tasks. It checks the call stack (where synchronous code is executed) and the task queue (where asynchronous callbacks wait) to decide what to run next.

- Simple Explanation:

  1. JavaScript runs all synchronous code.

  2. When asynchronous code (like a timer or network response) is ready, its callback is added to the task queue.

  3. Once the synchronous code is finished, the event loop takes tasks from the queue and runs them.

This helps JavaScript handle tasks like UI updates, timers, or network requests smoothly.

3. Callbacks

What is a Callback?

- A callback is simply a function that is passed as an argument to another function and gets called once an operation is complete.

Example:

index.js

```
function greet(name, callback) {
  console.log("Hello, " + name + "!");
  callback(); // Call the function passed as an argument
}

function askQuestion() {
  console.log("How are you?");
}

greet("Alex", askQuestion);
// Output:
// Hello, Alex!
// How are you?
```


4. Promises

What is a Promise?

- A Promise is an object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value.

- It has three states:
  - **Pending**: The initial state.
  - **Fulfilled**: The operation completed successfully.
  - **Rejected**: The operation failed.

Creating and Using a Promiseses

index.js

```
let myPromise = new Promise((resolve, reject) => {
  // Simulate an asynchronous task using setTimeout
  setTimeout(() => {
    let success = true; // Change this to false to simulate an error
    if (success) {
      resolve("✅ Success!");
    } else {
      reject("❌ Failed!");
    }
  }, 2000);
});

// Handling the promise:
myPromise
  .then(result => {
    console.log(result); // This will run if the promise is resolved
  })
  .catch(error => {
    console.error(error); // This will run if the promise is rejected
  });
```

Explanation:

- new Promise(...): Creates a new Promise.
- resolve(): Called when the operation succeeds.
- reject(): Called when the operation fails.
- .then(...): Runs when the promise is fulfilled.
- .catch(...): Runs when the promise is rejected.

5. Async/Await

What is Async/Await?

- async keyword: Used to declare a function as asynchronous. This function automatically returns a Promise.

- await keyword: Pauses the execution of an async function until the Promise is resolved or rejected. It makes asynchronous code look and behave like synchronous code.

Example:

index.js

```
// A function that returns a Promise
function simulateAsyncTask() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve("✅ Async/Await Success!");
    }, 2000);
  });
}

// Using async/await:
async function runTask() {
  try {
    console.log("Task started...");
    // Wait until the promise resolves
    const result = await simulateAsyncTask();
    console.log(result);
    console.log("Task finished.");
  } catch (error) {
    console.error(error);
  }
}

runTask();
// Output:
// Task started...
// (after 2 seconds)
// ✅ Async/Await Success!
// Task finished.
```


Explanation:

- The async keyword before the function runTask means it can use await inside.
- await simulateAsyncTask() pauses the function until simulateAsyncTask completes.
- The try...catch block is used to handle any errors that might occur.

Final Tips for Beginners

- Practice: Try writing small examples to see how asynchronous operations work.
- Visualize: Draw diagrams to understand the event loop and the flow of asynchronous code.
- Error Handling: Always consider how to handle errors (using .catch() with promises or try...catch with async/await).

6. Real-World Example: Fetching Data

Modern JavaScript often uses the Fetch API for network requests, which returns a Promise.

Example Using Promises:

index.js

```
fetch("https://jsonplaceholder.typicode.com/users/1")
  .then(response => response.json()) // Convert the response to JSON
  .then(data => console.log(data))
  .catch(error => console.error("Error fetching data:", error));
```

Example Using Async/Await:

index.js

```
async function fetchUser() {
  try {
    let response = await fetch("https://jsonplaceholder.typicode.com/users/1");
    let data = await response.json(); // Convert the response to JSON
    console.log(data);
  } catch (error) {
    console.error("Error fetching data:", error);
  }
}

fetchUser();
```

Explanation:

- Fetch API: Makes network requests and returns a Promise.
- .json(): A method on the response object that also returns a Promise to parse the JSON.
- Both examples show how to handle asynchronous network operations, but the async/await version is often easier to read and write.

Hopefully You Found It Usefull!

"Be sure to save this post so you can come back to it later

like Comment Share