# Asynchronous JavaScript - II

As discussed earlier, the use of repetitive callbacks can lead to a phenomenon known as "**callback hell**" or the "**pyramid of doom.**" To overcome this issue, modern programming languages and frameworks introduced a powerful concept called Promises. In this section, we will delve into Promises and understand their purpose, states, and how they are created and utilized.

## Promises

- Promises are objects that represent the eventual completion or failure of an asynchronous operation.
- They provide a cleaner and more organized way to handle asynchronous code, making it easier to read, write, and maintain.
- They represent the eventual completion or failure of an asynchronous operation and allow you to attach callbacks that will be executed when the operation is resolved or rejected.

### States of Promises

Promises can be in one of three states:

- **Pending:** The initial state when a Promise is created, and its outcome is yet to be determined.
- **Fulfilled:** The Promise has successfully completed, and the associated value (result) is available.
- **Rejected:** The Promise encountered an error or failure, and the reason for the failure is available.

## Promise Constructor

- To create a new Promise, we use the Promise constructor. The Promise constructor takes a single argument, which is a callback function with two parameters: **resolve** and **reject**.
    - The resolve function is used to fulfill the Promise and pass the resolved value.
    - The reject function is used to reject the Promise and pass the reason for rejection.

- Example:

```
const myPromise = new Promise((resolve, reject) => {
    // Perform an asynchronous task

    if (/* Task completed successfully */) {
      resolve('Task completed.'); // Resolve with a value
    } else {
      reject('Task failed.'); // Reject with a reason/error
    }
  });
```

In the example above, inside the callback function, you perform your desired asynchronous task. If it completes successfully, you call **resolve()** passing any relevant data as an argument (e.g., **`'Task completed.'`**). If there's an error or failure during execution, you call **reject()** passing an appropriate reason or error message (e.g., **`'Task failed.'`**).

## Working with Promises:

Once created, promises expose several methods for handling their resolution and rejection states:

- **.then():** Attaches a callback function to be executed when the promise is fulfilled.
- **.catch():** Attaches a callback function to be executed when the promise is rejected.
- **.finally():** Attaches a callback function to be executed regardless of the promise's state (whether fulfilled or rejected).
- Example:

```
myPromise
  .then((result) => {
    console.log(result); // Handle successful fulfillment
  })
  .catch((error) => {
    console.error(error); // Handle rejection/error
  })
  .finally(() => {
    console.log('Task completed, regardless of outcome.'); //
Perform cleanup tasks if needed
});
```

In the example above, we use `.then()` to attach a success callback that will receive the resolved value as its argument. If there's an error during execution, `.catch()` handles it by receiving and processing the rejected reason. Finally, `.finally()` allows you to execute code that needs to run regardless of whether the promise was fulfilled or rejected.

## Promises with fetch API

Promises are commonly used in conjunction with the Fetch API to handle asynchronous network requests and process their responses. The Fetch API provides a modern, promise-based approach for making HTTP requests.

## Using Promises with Fetch:

- To make an HTTP request using the Fetch API, we create a Promise that resolves when the response is received successfully. We can then use `.then()` to handle the response data or perform further operations.
- Example:

```
fetch('https://api.example.com/data')
  .then((response) => {
    if (!response.ok) {
      throw new Error('Network response was not okay.');
    }

    return response.json();
  })
  .then((data) => {
    console.log(data);
  })
  .catch((error) => {
    console.error(error);
});
```

In this example, `fetch()` sends an HTTP GET request to `'https://api.example.com/data'`. Inside the first .then(), we check if the server responded successfully (`response.ok`). If not, we throw an error. Otherwise, we parse the JSON response by calling `response.json()` and pass it along to our next `.then()`. Finally, any errors during execution are caught and handled by `.catch()`.

# Promise Chaining

- Promises can be chained together by returning another promise within each `` `.then()` `` block. This allows us to perform sequential operations or transform data based on previous results.
- By chaining promises together, you can create more complex flows for handling asynchronous operations and avoid nested callbacks or "callback hell."
- Example

```
fetch('https://api.example.com/data')
  .then((response) => {
    if (!response.ok) {
      throw new Error('Network response was not okay.');
    }

    return response.json();
  })
  .then((data) => {
    // Perform additional processing or transformation of data
    const transformedData = processData(data);

    // Make another fetch request based on transformed data
    return fetch('https://api.example.com/another-endpoint', {
      method: 'POST',
      body: JSON.stringify(transformedData),
      headers: {
        'Content-Type': 'application/json'
      }
    });
  })
  .then((response) => {
     if (!response.ok) {
       throw new Error('Network response was not okay.');
```

```
      }

      console.log('Request completed successfully.');
    })
    .catch((error) => {
      console.error(error);
});
```

In this example, after the initial fetch request and parsing of the JSON response, we perform additional processing on the data. Then, within our second `.then()`, we make another fetch request to a different endpoint using `fetch()` again. We pass along the transformed data as the request payload in this case.

# Async/Await

- Despite the introduction of Promises as a powerful mechanism to handle asynchronous operations in JavaScript, the code can still become complex and difficult to read when dealing with multiple asynchronous tasks.
- To address this, the Async/Await syntax was introduced, providing a more concise and synchronous-like way to write asynchronous code.
- Despite the existence of Promises, async/await has become increasingly popular due to its simplicity and improved code readability.

## Necessity of Async/Await

- While Promises offer an elegant way to handle asynchronous operations, they can sometimes lead to complex chaining or nesting when multiple asynchronous tasks need to be executed sequentially. This can result in **"callback hell"** or hard-to-read code structures.
- Async/await addresses this issue by allowing developers to write asynchronous code using a more synchronous style, making it easier to understand and maintain.

# Error Handling with Async/Await in JavaScript

- To use async/await, we mark a function as `async`, which automatically makes it return a Promise.
- Within an async function, we use the `await` keyword before calling any Promise-based functions or expressions.
- The `await` keyword pauses the execution of the function until the awaited Promise is resolved or rejected.
- Example:

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

    if (!response.ok) {
      throw new Error('Network response was not okay.');
    }

    const data = await response.json();

    console.log(data);
  } catch (error) {
     console.error(error);
  }
}
```

In this example, we define an async function called `` `fetchData()` ``. Inside this function:

- We use **try-catch** blocks for proper error handling.
- We call `fetch()` with await keyword before it so that execution pauses until the HTTP request's response is received.

- If there are errors during fetching (e.g., network error), an exception will be thrown and caught by the surrounding catch block.
- Using the if statement, we check if the response received is not okay (e.g., a non-successful HTTP status code). If it's not okay, we throw an Error using the `throw` keyword. Throwing an Error in an async function will cause the Promise to be rejected.
- If the response is successful, we parse it using `` `response.json()` `` with await keyword before it to pause execution until the JSON data is extracted.

By using async/await, our code structure becomes much cleaner compared to traditional promise chains with `.then()` callbacks. It allows us to write asynchronous code in a more linear and sequential manner, resembling synchronous code.

### Advantages of Async/Await over Promises:

1. **Readability:** Async/await provides a more intuitive and readable syntax for handling asynchronous operations, making the code easier to understand and maintain.
2. **Error Handling:** With async/await, error handling is simplified by using try-catch blocks instead of separate `.catch()` handlers for each Promise.
3. **Sequential Execution:** Async/await allows us to write asynchronous code that executes sequentially without deeply nested or chained callbacks, resulting in cleaner and more manageable code structures.
4. **Debugging:** Debugging async/await-based code is simpler since it closely resembles synchronous programming flow with step-by-step execution.

## Event Loop

The event loop is a crucial component of JavaScript's concurrency model. It manages the execution order of asynchronous operations, ensuring that they are executed in an efficient and non-blocking manner.

## Working of the JavaScript Program:

- **Initial Execution:**

  When a JavaScript program starts running, the main script is executed synchronously, line by line, pushing functions and their local variables onto the Call Stack.

- **Web API Interaction:**

  If an asynchronous task, like an API request, is encountered, it is handed over to the appropriate Web API. The JavaScript runtime continues executing subsequent tasks without waiting for the result.

- **Event Loop Process:**

  The Event Loop continuously checks the Call Stack. If it is empty, it looks into the Microtask Queue first, executing any pending microtasks until it is empty. This ensures timely execution of high-priority tasks.

- **Callback Queue Execution:**

  After processing the Microtask Queue, the Event Loop moves on to the Callback Queue. It takes the oldest task from the Callback Queue and pushes it onto the Call Stack for execution. This process repeats until the Callback Queue is empty.

To better understand the event loop, let's break it down into its key components as mentioned below:

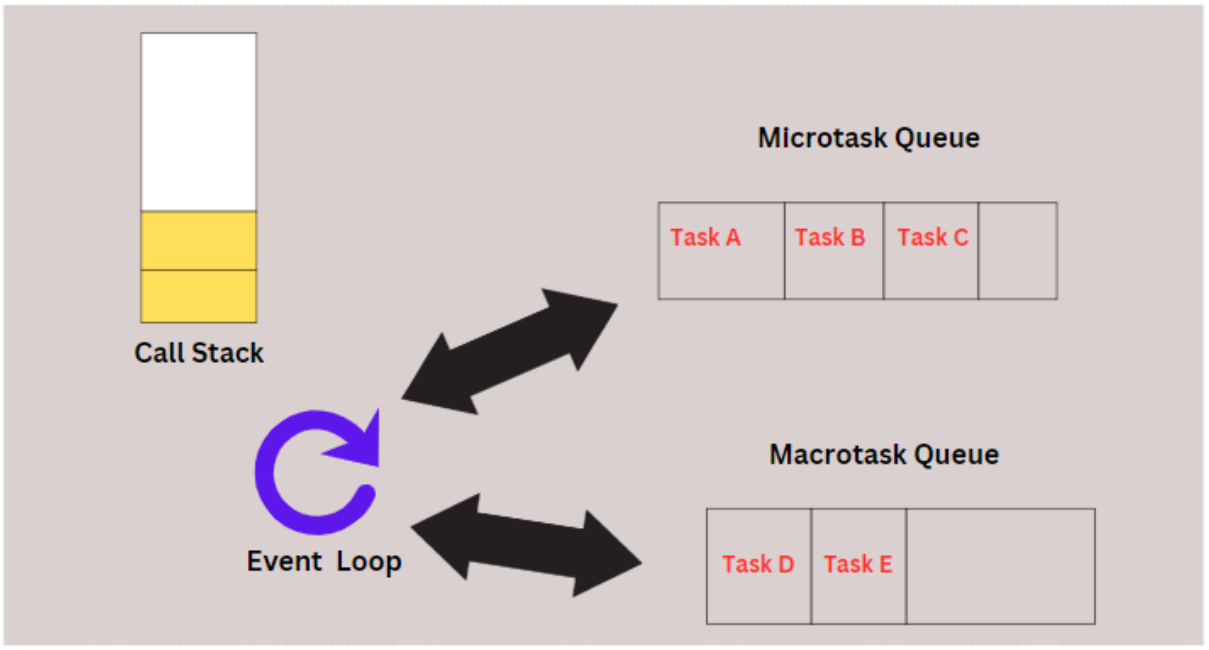


Figure:1

**1. Call Stack:**

The call stack is responsible for keeping track of function calls during program execution. When a function is called, it gets pushed onto the call stack, and when it returns, it gets popped off.

**2. Web APIs:**

Web APIs are provided by web browsers and allow us to perform various asynchronous tasks such as making HTTP requests (`fetch()`), setting timers (`setTimeout()`), or handling user events (`addEventListener()`). These tasks are performed outside the JavaScript runtime environment.

**3. Callback Queue:**

When an asynchronous task completes (e.g., timer expires or network request finishes), its associated callback function is placed in the callback queue.

**4. Microtask Queue:**

Microtasks have higher priority than regular callbacks and are used for tasks like Promise resolutions via `resolve()`, `reject()`, or using `async/await`. They get executed before regular callbacks from the callback queue.

# Summarizing it

In this lecture, we have covered the following topics:

- Promises:
    - Covers the states of Promises (pending, fulfilled, rejected) and the Promise constructor.
    - Explains how to create Promises and introduces methods like resolve, reject, `.then()`, `.catch()`, and `.finally()`.
- Promises with Fetch API:
    - Demonstrates using Promises with the Fetch API to make asynchronous data requests.
    - Highlights how Promises simplify handling the response and error scenarios.
- Promise Chaining:
    - Explores Promise chaining, where multiple Promises can be sequenced and their results handled in a linear manner.
    - Illustrates how Promise chaining improves code organization and readability.
- Async/Await:
    - Introduces Async/Await, a more concise and synchronous-like syntax for handling asynchronous operations.
    - Describes how to mark functions as async and use the `await` keyword to pause execution until Promises resolve.
- Error Handling using Try/Catch:
    - Discusses error handling in Promises and Async/Await using the try/catch block.

- Event Loop and Queues:
    - Mentions the Event Loop's role in managing asynchronous tasks in JavaScript.
    - Describes the Callback Queue and Microtask Queue and the preference of Microtasks over Callbacks for priority execution.

## References

- Promise: Link
- Async/await: Link