Comprehensive Notes on Asynchronous JavaScript

Introduction to Asynchronous JavaScript

Asynchronous JavaScript enables **non-blocking operations**, which means that long-running tasks such as fetching data from a server, reading files, or performing computations can occur without halting the execution of other code. This is crucial in JavaScript because it operates on a **single-threaded model**, meaning it can execute only one task at a time. Without asynchronous programming, JavaScript would be stuck waiting for each task to complete before moving on, leading to **performance issues** and **frozen interfaces** in web applications.

Key Benefits of Asynchronous Programming:

1. **Improved User Experience**: The browser remains responsive even during time-consuming tasks.
2. **Efficient Resource Usage**: Tasks like fetching data or interacting with APIs can occur while other tasks are executed.
3. **Concurrency**: Multiple operations can run simultaneously, even though JavaScript uses a single-threaded event loop.

Key Concepts of Asynchronous JavaScript

1. Callback Functions

A **callback function** is a function passed as an argument to another function to be executed later. It serves as a way to defer execution until a specific task is completed. This is one of the earliest and most foundational approaches to handling asynchronous tasks in JavaScript.

Features of Callbacks:

- They allow asynchronous code to notify the program when it has completed its task.
- Used in scenarios like **event listeners**, **timers**, and **API calls**.

Example: Using `setTimeout` with a Callback

The `setTimeout` function accepts a callback that runs after a specified time delay (in milliseconds):

```
setTimeout(() => console.log("Callback executed after 2 seconds"), 2000);
```

Explanation:

1. The `setTimeout` function schedules the callback function to run **after 2 seconds**.
2. While waiting, the JavaScript engine continues executing the rest of the code without blocking.

Synchronous vs. Asynchronous Callback Example:

- Synchronous:

```
function add(a, b, callback) {
  const sum = a + b;
  callback(sum);
}
add(5, 3, (result) => console.log(`Result: ${result}`)); // Executes
immediately
```

- Asynchronous:

```
console.log("Start");
setTimeout(() => console.log("Executed after delay"), 1000);
console.log("End");
// Output:
// Start
// End
// Executed after delay
```

---

2. AJAX (Asynchronous JavaScript and XML)

AJAX allows **dynamic updates** to parts of a webpage without requiring a full page reload. Although originally named for XML-based data, modern implementations of AJAX often use **JSON** and the **Fetch API** instead of XML.

Core Features of AJAX:

- Facilitates **server communication** in the background.
- Improves **performance and interactivity** by eliminating the need for full-page refreshes.
- Traditionally used the `XMLHttpRequest` object, which has now been largely replaced by `fetch`.

Example with `XMLHttpRequest`:

Below is a simple implementation of AJAX using `XMLHttpRequest` to fetch data about a country:

```
const getCountryData = function (country) {
  const request = new XMLHttpRequest(); // Create a new request
  request.open("GET", `https://restcountries.com/v2/name/${country}`); //
Specify the HTTP method and URL
  request.send(); // Send the request to the server

  // Add an event listener to handle the response
  request.addEventListener("load", function () {
    const [data] = JSON.parse(this.responseText); // Parse the JSON response
    console.log(data); // Log the retrieved data
```

```
  });
};
getCountryData("portugal"); // Fetch data about Portugal
```

Explanation:

1. `new XMLHttpRequest()`: Creates a new HTTP request object.
2. `open(method, URL)`: Configures the request, specifying the HTTP method (`GET` in this case) and the endpoint.
3. `send()`: Sends the request to the server.
4. `addEventListener("load")`: Waits for the server's response and executes the callback once the data is received.
5. `JSON.parse(responseText)`: Converts the server's JSON string response into a JavaScript object.

---

3. Callback Hell

Callback Hell refers to a situation where multiple nested callbacks result in **deeply indented, hard-to-read code**. This often happens when handling sequential asynchronous operations using callbacks, making debugging and maintaining the code highly challenging.

Example of Callback Hell:

Here's a basic example using `setTimeout` to simulate sequential tasks:

```
setTimeout(() => {
  console.log("1 second passed");
  setTimeout(() => {
    console.log("2 seconds passed");
    setTimeout(() => {
      console.log("3 seconds passed");
    }, 1000);
  }, 1000);
}, 1000);
```

Output:

```
1 second passed
2 seconds passed
3 seconds passed
```

Why This is a Problem:

1. **Readability**: Code becomes difficult to understand due to heavy nesting.
2. **Maintainability**: Modifying or extending the logic can introduce errors.
3. **Error Handling**: Managing errors in deeply nested callbacks is complex.

4. Promises

Promises were introduced in ES6 as a cleaner way to handle asynchronous operations, avoiding callback hell and improving error handling. A **promise** represents a **placeholder for the result of an asynchronous task**, which will eventually resolve to a value or reject with an error.

States of a Promise:

1. **Pending**: The initial state; the result is not yet determined.
2. **Fulfilled**: The operation completed successfully, and the promise is resolved with a value.
3. **Rejected**: The operation failed, and the promise is rejected with an error.

Creating a Promise:

A promise is created using the `Promise` constructor, which takes a function with two parameters: `resolve` (to indicate success) and `reject` (to indicate failure).

```
const lotteryPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    if (Math.random() >= 0.5) {
      resolve("You WIN 💰"); // Resolve the promise if condition is met
    } else {
      reject(new Error("You lost 💩")); // Reject the promise if condition
fails
    }
  }, 2000);
});
```

Consuming a Promise:

Promises are consumed using `.then()` for resolved values and `.catch()` for errors:

```
lotteryPromise
  .then((result) => console.log(result)) // Handle success
  .catch((err) => console.error(err.message)); // Handle error
```

Why Promises are Better than Callbacks:

1. **Chaining**: Promises can be chained, avoiding deeply nested structures.
2. **Error Propagation**: Errors are caught and propagated automatically with `.catch()`.
3. **Readability**: Code is easier to read and maintain compared to nested callbacks.

5. Promise Methods

Promises provide several built-in methods to handle multiple asynchronous operations efficiently. These methods are crucial when dealing with parallel tasks or when you need to aggregate results from multiple promises.

Key Promise Methods:

1. `Promise.all()`:
    - Waits for **all promises** in an array to resolve or for any to reject.
    - If any promise is rejected, the entire operation fails, and the error is returned.
    - Commonly used when all results are required to proceed further.

    Example:

    ```
    Promise.all([
      fetch(`https://restcountries.com/v2/name/portugal`).then((res) =>
        res.json()
      ),
      fetch(`https://restcountries.com/v2/name/canada`).then((res) =>
        res.json()
      ),
    ])
      .then((data) => console.log(data)) // Resolves when all fetch calls
    succeed
      .catch((err) => console.error(err)); // Fails if any fetch call rejects
    ```

    Explanation:

    - In the above example, two API calls are made in parallel.
    - The results of both promises are returned as an array if successful.
    - If one request fails (e.g., network issue or invalid URL), the `.catch()` block is triggered.

2. `Promise.race()`:
    - Returns the result of the **first settled promise** (resolved or rejected) from an array of promises.
    - Useful for scenarios like setting timeouts on operations.

    Example:

    ```
    const timeout = new Promise((_, reject) =>
      setTimeout(() => reject(new Error("Request timed out")), 3000)
    );

    const fetchPromise = fetch(
      `https://restcountries.com/v2/name/portugal`
    ).then((res) => res.json());

    Promise.race([fetchPromise, timeout])
    ```

```
  .then((data) => console.log(data))
  .catch((err) => console.error(err));
```

Explanation:

- This approach ensures that if the API call takes longer than 3 seconds, the timeout promise rejects the operation.

3. `Promise.allSettled()`:

- Returns the results of all promises in an array, regardless of whether they **resolve or reject**.
- Unlike `Promise.all()`, it doesn't fail on rejection and provides the status of each promise.

Example:

```
Promise.allSettled([
  fetch(`https://restcountries.com/v2/name/portugal`).then((res) =>
    res.json()
  ),
  fetch(`https://invalid-url.com`).then((res) => res.json()),
]).then((results) => console.log(results));
```

Output:

```
[
  {
    status: "fulfilled",
    value: {
      /* data for Portugal */
    },
  },
  { status: "rejected", reason: TypeError("Failed to fetch") },
];
```

Explanation:

- `Promise.allSettled()` is great when you need to handle individual successes or failures gracefully, without short-circuiting the operation.

4. `Promise.any()`:

- Returns the first successfully resolved promise from an array.
- If all promises reject, it throws an **AggregateError** containing all rejection reasons.

Example:

```

```

```
Promise.any([
  fetch(`https://invalid-url.com`).then((res) => res.json()),
  fetch(`https://restcountries.com/v2/name/portugal`).then((res) =>
    res.json()
  ),
])
  .then((data) => console.log(data))
  .catch((err) => console.error(err));
```

Explanation:

- In this example, the second fetch call succeeds first, so its data is returned.
- If all promises fail, an `AggregateError` is thrown.

---

6. Async/Await

`Async/await` is a **syntactic sugar** built on top of promises, introduced in ES2017 (ES8). It makes asynchronous code look and behave more like synchronous code, improving readability and error handling.

Key Features

- `async` functions automatically return a promise.
- `await` pauses the execution of the async function until the promise resolves or rejects.

Example

```
const getCountryData = async function (country) {
  try {
    const res = await fetch(`https://restcountries.com/v2/name/${country}`);
    if (!res.ok) throw new Error("Country not found"); // Custom error handling
    const data = await res.json();
    console.log(data);
  } catch (err) {
    console.error(err); // Handle errors from both fetch and JSON parsing
  }
};
getCountryData("portugal");
```

Explanation:

1. `async` **function**:
   - Declares the function as asynchronous, automatically wrapping the return value in a promise.
2. `await` **keyword**:
   - Waits for the `fetch` promise to resolve before continuing to the next line.
3. **Error handling**:

- The `try...catch` block ensures both fetch and runtime errors (like invalid JSON or network failures) are handled in a single place.

Why Use Async/Await?

1. **Readability**: Avoids chaining `.then()` calls.
2. **Error Handling**: Errors can be caught directly in `try...catch` blocks, making debugging easier.
3. **Synchronous Flow**: Asynchronous operations appear to run in a sequential order.

---

7. Error Handling

Error handling is critical in asynchronous JavaScript because operations like network requests, file reads, or timers often fail unpredictably.

Error Handling with Promises

Errors in promises are handled using the `.catch()` method:

```
fetch(`https://restcountries.com/v2/name/invalid-country`)
  .then((res) => {
    if (!res.ok) throw new Error("Country not found"); // Custom error
    return res.json();
  })
  .then((data) => console.log(data))
  .catch((err) => console.error(err.message)); // Logs error messages
```

Error Handling with Async/Await

The `try...catch` block provides a clean way to handle errors in `async/await` functions:

```
const getCountryData = async function (country) {
  try {
    const res = await fetch(`https://restcountries.com/v2/name/${country}`);
    if (!res.ok) throw new Error("Country not found");
    const data = await res.json();
    console.log(data);
  } catch (err) {
    console.error(`Something went wrong: ${err.message}`);
  }
};
getCountryData("invalid-country");
```