Asynchronous JavaScript - I

Introduction

- JavaScript (JS) is a programming language that is known for being synchronous and single-threaded, which means it executes code line by line in a sequential manner.
- This synchronous nature can sometimes lead to blocking behavior, where subsequent lines of code have to wait for the completion of previous operations before executing.
- An example illustrating the blocking nature of JS is as follows:

```
console.log('Message before alert');
alert('blocking JS');
console.log('Message after alert');
```

In this example, when executed in a browser environment, the first `**console.log**` statement will be displayed in the console. However, once **`alert('blocking JS')`** is encountered, it blocks further execution until the user closes the alert dialog. As a result, the last `**console.log**` statement will not be executed until after closing the alert dialog.

Asynchronous Nature

- Although JavaScript is primarily synchronous and single-threaded, it also supports asynchronous behavior through mechanisms like the event loop.
- The event loop allows non-blocking execution by handling tasks asynchronously without interrupting other operations.

- Several APIs leverage this asynchronous capability in JavaScript:
    - `setTimeout`: Allows you to schedule a function to run after a specified delay.
    - `setInterval`: Executes a function repeatedly at defined intervals.
    - `fetch`: Enables making HTTP requests asynchronously.
- These APIs make use of callback functions or Promises/async-await syntax to handle asynchronous operations effectively while allowing other code to continue executing concurrently.
- The event loop ensures that such asynchronous tasks are queued and processed separately from regular synchronous execution. It monitors task completion and triggers their associated callbacks when ready.

As we delve deeper into concepts like Promises and async-await later in these notes, we'll explore how they further enhance JavaScript's ability to handle asynchronous programming efficiently.

setTimeout API

- The `setTimeout` API in JavaScript allows you to schedule the execution of a function after a specified delay.
- It takes two parameters: a callback function and the delay time in milliseconds. Additionally, it can accept extra arguments that will be passed to the callback function when it is invoked.
- **Syntax:**

```
setTimeout(callback, delay, arg1, arg2, ...);
```

    - **callback:** A function to be executed after the specified delay.
    - **delay:** The amount of time (in milliseconds) to wait before executing the callback function.

- **arg1, arg2, ...:** Extra arguments (optional) that will be passed as parameters to the callback function.

- **Example:**

```
function greet(name) {
  console.log(`Hello ${name}!`);
}

// Execute greet() with an argument after 2 seconds (2000 milliseconds)
const timeoutId = setTimeout(greet, 2000, "John");

// Cancel timeout before it executes
clearTimeout(timeoutId);
```

In this example, we have defined a **`greet`** function that takes a parameter **`name`**. By passing **`"John"`** as an additional argument in **`setTimeout`**, it gets passed into the **`greet`** function when it is called asynchronously after a delay of 2 seconds. However, if we want to cancel or stop this timeout from executing before its completion, we can use **`clearTimeout`** by passing in its timer ID (**`timeoutId`**) obtained from **`setTimeout`**.

setInterval API

- The **`setInterval`** API is similar to **`setTimeout`**, but instead of executing a callback once after a specific interval, it repeatedly executes the callback at defined intervals until cleared or stopped.

- Like **`setTimeout`**, it can also accept extra arguments for passing values into the callback.

- **Syntax:**

```
setInterval(callback, interval[, arg1[, arg2[, ...]]]);
```

  - **callback:** A function to be executed repeatedly at each interval.
  - **interval:** The time duration (in milliseconds) between each invocation of the callback.
  - **arg1, arg2, …:** Optional additional arguments that will be passed to the callback function.

- **Example:**

```
function greet(name) {
  console.log(`Hello ${name}!`);
}

// Greet with a name every second (1000 milliseconds)
const intervalId = setInterval(greet, 1000, "Alice");

// Stop greeting after 5 seconds
setTimeout(() => {
   clearInterval(intervalId); // Clearing interval using
clearInterval()
}, 5000);
```

In this example, we define a `greet` function that takes a `name` parameter. Using `setInterval`, we repeatedly call the `greet` function every second while passing `"Alice"` as an extra argument. After 5 seconds, we clear the interval using the timer ID obtained from `setInterval`. This prevents further execution of the scheduled callbacks.

Additional Methods: clearInterval and clearTimeout

To stop or cancel timeouts and intervals before they are executed or completed, JavaScript provides two methods:

- **clearTimeout:** This method cancels a timeout previously set by calling `setTimeout`. It takes one parameter, which is the timer ID returned by `setTimeout`.
- **clearInterval:** This method clears an interval set by calling `setInterval`. It also accepts one parameter - the timer ID returned by `setInterval`.

Asynchronous HTTP Requests

- In JavaScript, performing HTTP requests is a common requirement for interacting with web servers and retrieving data. With the introduction of asynchronous programming, developers can make HTTP requests in a non-blocking manner, allowing the rest of the program to continue executing while waiting for the response.
- There are several ways to accomplish this, including the older XMLHttpRequest (XHR) approach using callback functions, the modern Promise-based approach using the fetch API, and the more recent async-await syntax.

XMLHttpRequest (XHR)

The XMLHttpRequest (XHR) object is an older approach for making asynchronous HTTP requests from a web browser. It utilizes callback functions to handle different phases of the request-response cycle.

Note: Please remember that callbacks and their usage have been covered in Lecture 5. If you need further information on callbacks, please refer back to that lecture material.

XHR Phases:

1. Initiating an HTTP Request:

To initiate an HTTP request using XHR, follow these steps:

A. Create an instance of `XMLHttpRequest` using the `new` keyword:

```
const xhr = new XMLHttpRequest();
```

B. Open a connection with the server by specifying the method (`GET`, `POST`, etc.) and URL:

```
xhr.open('GET', 'https://api.example.com/data');
```

C. Set up a callback function to handle the response when it's received:

```
xhr.onload = function() {
  // Handle successful response here
  console.log(xhr.responseText);
};

xhr.onerror = function() {
  // Handle error here
  console.error('An error occurred.');
};

xhr.onprogress = function(event) {
   // Track progress of request if needed
   console.log(`Loaded ${event.loaded} bytes`);
 };

 // ... Other event handlers can be set as well ...
```

D. Send the request to the server:

```
xhr.send();
```

2. Handling Response Data:

When the response is received successfully, or if there's any error during transmission or processing, corresponding callback functions are invoked.

- The `onload` callback handles a successful response.
- The `onerror` callback handles errors such as network issues or failed connections.
- The `onprogress` callback can be used to track the progress of a request, like showing download progress.

Within these callbacks, you can access and process the response data using `xhr.responseText`.

Callback Hell

- Dealing with callbacks in XHR can lead to **"callback hell"** or **"pyramid of doom,"** where nested callbacks become difficult to manage and maintain code readability. This issue arises when multiple asynchronous operations are involved or when error handling becomes complex.
- To address this problem, JavaScript introduced Promises. Promises provide a cleaner way to handle asynchronous operations by utilizing chaining and avoiding excessive nesting of callbacks. Promises allow for better organization of code and more readable syntax.
- By using promises with modern approaches like fetch API or async-await syntax, developers can avoid callback hell and write more maintainable code that is easier to understand.

Summarizing it

In this lecture, we have covered the following topics:

- Introduction to synchronous and single-threaded JS: JavaScript is a synchronous and single-threaded language, meaning it processes tasks one at a time in a sequential manner.
- Asynchronous nature of JS: JavaScript also supports asynchronous behavior, allowing tasks to run independently without blocking the main thread.
- APIs like setTimeout and setInterval: These APIs are used to introduce delays and execute functions at specified time intervals, enabling asynchronous behavior in JavaScript programs.
- Asynchronous HTTP requests: JavaScript provides different methods for performing asynchronous HTTP requests, allowing communication with web servers and retrieval of data without blocking the execution.
- XMLHttpRequest (XHR) and its various phases: XHR is an older method for making HTTP requests in JavaScript, involving various phases like initiating the request, handling responses, and error handling.
- Callback Hell: Callback Hell refers to the issue of nesting multiple callback functions within one another, making the code complex and challenging to read and maintain.

References

- Asynchronous JavaScript: **Link**
- XMLHttpRequest: **Link**