15. Asynchronous JavaScript & EVENT LOOP from scratch

📘 Index (Table of Contents)

❓ Questions and Answers

1. What Is Asynchronous JavaScript?

Q: What does asynchronous mean in the context of JavaScript?

A: Asynchronous JavaScript allows the execution of non-blocking operations. This means that JavaScript can initiate a task (like fetching data) and move on to other tasks without waiting for the previous one to complete. Once the task is finished, a callback function is executed to handle the result.

Analogy: Consider a chef (JavaScript) who starts baking a cake (asynchronous task). While the cake is baking, the chef prepares other dishes (other tasks). Once the cake is ready, the chef serves it (callback function).

2. How Does the Call Stack Work?

Q: What is the call stack, and how does it function in JavaScript?

A: The call stack is a data structure that keeps track of function calls in JavaScript. When a function is invoked, it's added to the top of the stack. Once the function execution is complete, it's removed from the stack. JavaScript is single-threaded, meaning it can only execute one operation at a time, processing functions in a Last In, First Out (LIFO) order.

Example:

```
function greet() {
  console.log("Hello");
}
greet();
```

Explanation: Here, the `greet` function is pushed onto the call stack when invoked and popped off once it finishes executing.

3. What Role Do Web APIs Play?

Q: What are Web APIs, and how do they relate to asynchronous operations?

A: Web APIs are provided by the browser (or Node.js) to handle asynchronous operations like `setTimeout`, `fetch`, and DOM events. When such an operation is called, JavaScript delegates it to the Web API, which handles the task in the background. Once completed, the callback function is placed in the callback queue to be executed when the call stack is empty.

Example:

```
setTimeout(() => {
  console.log("Executed after 2 seconds");
}, 2000);
```

Explanation: The `setTimeout` function is a Web API that waits for the specified time before executing the callback function.

4. What Is the Callback Queue?

Q: What is the callback queue, and how does it interact with the call stack?

A: The callback queue is a queue that holds callback functions waiting to be executed. Once the call stack is empty, the event loop picks the first function from the callback queue and pushes it onto the call stack for execution.

Analogy: Imagine a line at a ticket counter (callback queue). The person at the front of the line (callback function) is served (executed) when the counter (call stack) is free.

5. What Is the Event Loop?

Q: What is the event loop, and how does it manage asynchronous operations?

A: The event loop is a mechanism that continuously checks the call stack and the callback queue. If the call stack is empty, it moves the first function from the callback queue to the call stack for execution. This process ensures that asynchronous operations are handled efficiently without blocking the main thread.

Visual Representation:

```
Call Stack:      [ ]
Callback Queue:  [ callback1, callback2, ... ]
Event Loop:      Checks if Call Stack is empty, then moves functions from C
allback Queue to Call Stack
```

6. What Is the Microtask Queue?

Q: What is the microtask queue, and how does it differ from the callback queue?

A: The microtask queue holds tasks like promises and `MutationObserver` callbacks. It has a higher priority than the callback queue. After the current script execution completes and before the event loop picks up tasks from the callback queue, it first checks and executes all tasks in the microtask queue.

Example:

```
Promise.resolve().then(() => console.log("Microtask executed"));
console.log("Synchronous task executed");
```

Explanation: Even though the promise is resolved asynchronously, its callback is placed in the microtask queue and executed before any tasks in the callback queue.

7. How Does `setTimeout` Behave with a 0ms Delay?

Q: What happens when `setTimeout` is called with a delay of 0 milliseconds?

A: Even with a 0ms delay, the callback function is not executed immediately. It is placed in the callback queue and will only be executed when the call stack is empty. This behavior ensures that the callback doesn't block the execution of other code.

Example:

```
setTimeout(() ⇒ console.log("Executed after 0ms"), 0);
console.log("Synchronous task executed");
```

Explanation: The synchronous task is logged first, and the `setTimeout` callback is logged afterward, demonstrating that the callback waits for the call stack to be empty.

8. What Are Best Practices for Asynchronous Code?

Q: What are some best practices for writing asynchronous JavaScript code?

A: Best practices include:

- **Avoid Blocking the Call Stack:** Ensure that long-running operations do not block the call stack, allowing the event loop to process other tasks.
- **Use Promises and `async` / `await`:** These provide a more readable and maintainable way to handle asynchronous code compared to nested callbacks.
- **Handle Errors Properly:** Always handle errors in asynchronous code to prevent unhandled promise rejections and improve code reliability.
- **Remove Unused Event Listeners:** To prevent memory leaks, remove event listeners that are no longer needed.

🔑 Summary and Key Takeaways

- **Asynchronous JavaScript:** Allows non-blocking operations, enabling efficient handling of tasks like I/O operations and timers.
- **Call Stack:** A LIFO structure that keeps track of function execution.
- **Web APIs:** Handle asynchronous operations and delegate tasks to the browser's environment.
- **Callback Queue:** Holds functions waiting to be executed once the call stack is empty.

- **Event Loop:** Monitors the call stack and callback queue, ensuring asynchronous tasks are executed in order.
- **Microtask Queue:** Holds tasks like promises with higher priority than the callback queue.
- **Best Practices:** Avoid blocking the call stack, use modern asynchronous patterns, handle errors, and manage event listeners properly.