Intermediate JavaScript Programming

LESSON 5: Asynchronous JavaScript & the Event Loop

Learning Objectives:

By the end of this lesson, participants will be able to:

- Understand how JavaScript handles asynchronous operations.
- Distinguish between the call stack, Web APIs, the event queue, and the microtask queue.
- Use callbacks, Promises, and async/await to write asynchronous logic.
- Understand how reactivity in JavaScript differs from traditional blocking languages like Python or C#.

I. The JavaScript Execution Model (15 min)

JavaScript uses a **single-threaded**, **non-blocking** model powered by an **event loop**. Unlike Python or C#, where threads or `await` mechanisms often rely on runtime-managed task queues, JavaScript exposes the steps more clearly and directly.

Key Concepts:

🧠 Call Stack

The **call stack** tracks what code is currently executing. It operates as **last in, first out** (LIFO):

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

Call Stack:

```
start()
→ greet()
→ console.log()
```

Each function is pushed onto the stack, then popped off when finished.

🌐 Web APIs / Background Tasks

These are **provided by the browser (or Node runtime)**. They handle time-consuming tasks **outside** the call stack:

- `setTimeout`, `setInterval`
- `fetch`, AJAX calls
- Event listeners (mouse clicks, keystrokes)

Example:

```
setTimeout(() => {
  console.log("Done");
}, 1000);
```

The callback is stored by the browser and only queued after 1000ms **and** the call stack is empty.

📥 Event Queue (Callback Queue)

This queue holds callbacks from the Web APIs, ready to execute **once the stack is clear**.

```
console.log("Start");
setTimeout(() => console.log("Timeout"), 0);
console.log("End");
```

Output:

```
Start
End
Timeout
```

⚡ Microtask Queue

This queue is prioritized **above** the Event Queue. It holds:

- Promise `.then()` callbacks
- `queueMicrotask()` calls

```
console.log("Start");
Promise.resolve().then(() => console.log("Promise"));
setTimeout(() => console.log("Timeout"), 0);
console.log("End");
```

Output:

```
Start
End
```

```
Promise
Timeout
```

Microtasks run after the current stack frame but before the event queue.

II. Callbacks and "Callback Hell" (10 min)

A **callback** is a function passed into another function to run later:

```
function doLater(callback) {
  setTimeout(callback, 1000);
}
doLater(() => console.log("Finished"));
```

Problem: Callback Hell

When callbacks are nested repeatedly:

```
step1(() => {
  step2(() => {
    step3(() => {
      console.log("Done");
    });
  });
});
```

This becomes hard to read and maintain. Promises solve this.

III. Promises (15 min)

A **Promise** represents a future value. It's either:

- **Pending**: not yet complete
- **Resolved**: completed successfully
- **Rejected**: completed with an error

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Data ready");
  }, 1000);
});

promise.then(result => console.log(result));
```

Error Handling with Promises

```
promise
  .then(result => handle(result))
  .catch(error => console.error("Error:", error));
```

Promise Chaining

```
fetch('/data')
  .then(response => response.json())
  .then(json => process(json))
  .catch(error => console.error(error));
```

In C# or Python, you might use `await` and try/catch blocks — but Promises allow the same structure with `.then()` and `.catch()`.

IV. async/await Syntax (15 min)

Introduced in ES2017, `async`/`await` allows you to write asynchronous code **like it's synchronous**, but behind the scenes it uses Promises.

```
async function loadData() {
  try {
    const response = await fetch('/data.json');
    const data = await response.json();
    console.log(data);
  } catch (err) {
    console.error("Failed to load", err);
  }
}
loadData();
```

- Functions marked `async` always return a Promise.
- You can only use `await` inside an `async` function.

This is similar to Python's `async def` and C#'s `async Task`, but JavaScript is fully non-blocking by default.

V. Recap & Q&A (5 min)

- JavaScript uses the call stack, Web APIs, event queue, and microtask queue to manage concurrency.
- Promises solve the problem of deeply nested callbacks.
- `async/await` lets you write readable asynchronous code.

- Microtasks are prioritized before other queued events.

Final Multiple-Choice Question:

What is the primary difference between the Event Queue and the Microtask Queue? A. The Microtask Queue runs after the event loop B. The Event Queue only handles Promises C. The Microtask Queue has higher priority and runs before the Event Queue D. There is no difference — both are part of the call stack

(Answer: C. The Microtask Queue has higher priority and runs before the Event Queue)