Welcome to lecture 18!

Agenda

Session Objectives

- Javascript execution model
  - Synchronous operations
  - Call Stack
  - Single-threaded programming
- Uncover Event Loop and its components
- Writing non-blocking code
  - Callbacks
  - Promises
  - Async/await
- Quiz

Introduction to JS Execution

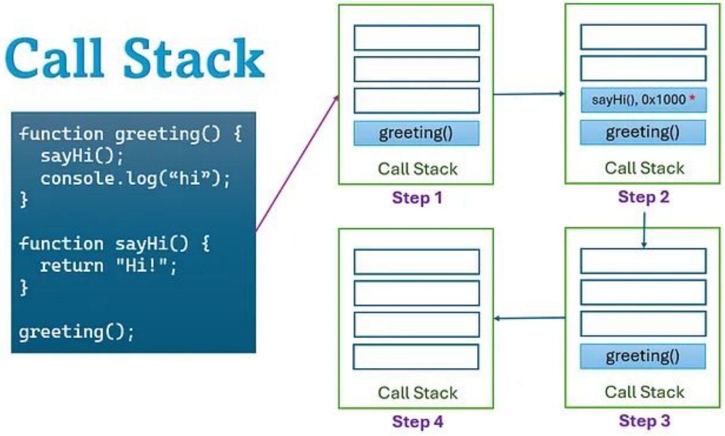
JavaScript Execution: Synchronous by Default

- What's the Synchronous Model?
  - Synchronous Model: JavaScript code statements are executed sequentially, one after another, in the order they appear
  - Each operation must complete before the next one begins; the engine proceeds to the next line only after the current one is fully processed
  - This predictable, ordered execution is the default behavior of the JavaScript engine

```javascript
console.log("Task 1: Initialize settings");
let userName = "Learner";
console.log(`Task 2: Welcome ${userName}`);
let sum = 0;
for(let i=1; i<=3; i++) sum += i; // A quick, small synchronous block
console.log(`Task 3: Calculated sum: ${sum}`);
console.log("Task 4: Script execution complete");
```

The Call Stack: Managing Function Execution

- The Call Stack is a data structure that JavaScript uses to keep track of function calls during script execution
- When a function is called, a new "frame" representing that function call is pushed onto the top of the stack
- When a function completes (returns), its frame is popped off the stack.
- Operates on a Last-In, First-Out (LIFO) principle



The Call Stack: Example

```
function garnishFood() {
  console.log("Step 3: Food garnished!");
}
function cookMainCourse(ingredient) {
  console.log(`Step 2: Cooking ${ingredient}...`);
  garnishFood(); // Pushes garnishFood onto stack
  console.log("Step 2.1: Main course cooked.");
}
function prepareMeal() {
  console.log("Step 1: Starting meal preparation...");
  cookMainCourse("Pasta"); // Pushes cookMainCourse onto stack
  console.log("Step 1.1: Meal preparation complete.");
}
prepareMeal(); // Pushes prepareMeal onto stack
console.log("Kitchen is clean!");
```

Runtime: Single Thread, Call Stack & Memory Heap

- JavaScript is single-threaded: It has only one Call Stack, meaning it can execute only one sequence of instructions (one "thread" of execution) at any given moment
- The JavaScript runtime environment primarily consists of:
  - Call Stack: Manages execution contexts (function calls).
  - Memory Heap: A large, unstructured region of memory where objects and variables are stored when your script creates them
- This single-threaded model implies that all JavaScript code for your page shares the same thread and memory space

The "Blocking" Effect: Impact of Long Synchronous Tasks

- When a synchronous task on the single Call Stack takes significant time (e.g., complex calculations, large data processing loops), it blocks the thread
- While the thread is blocked:
  - No other JavaScript (including event handlers like clicks, scrolls) can execute
  - The browser cannot perform UI updates (repaints/reflows), leading to a frozen or unresponsive page
  - User interactions appear ignored until the blocking task completes and the Call Stack clears

Asynchronous JavaScript

Web APIs & Asynchronous Tasks

- Web Browsers provide Web APIs (e.g., setTimeout, DOM event mechanisms, fetch API) that operate outside the main JavaScript thread.
- These APIs can handle tasks like timers or network requests in the background, allowing the JavaScript Call Stack to remain clear for other operations
- setTimeout(callbackFunction, delay): A common Web API that schedules callbackFunction to be executed after delay milliseconds
    - JavaScript does not wait for the delay; it initiates the timer via the Web API and continues executing subsequent code
- setInterval(callbackFunction, intervalDelay): Similar to setTimeout, but repeatedly executes callbackFunction every intervalDelay milliseconds until cleared

Asynchronous Tasks: Example

```javascript
console.log("Message A: Script Start");

setTimeout(function greet() {
  console.log("Message C: Hello from setTimeout (after 2s)!");
}, 2000);

setTimeout(function quickTask() {
  console.log("Message D: Hello from quick setTimeout (0ms)!");
}, 0); // Note: 0ms delay doesn't mean immediate execution on the main thread.

let count = 0;
const intervalId = setInterval(function tick() {
  count++;
  console.log(`Message E: Interval Tick ${count}`);
  if (count >= 3) {
    clearInterval(intervalId); // Important to clear intervals
    console.log("Message F: Interval cleared.");
  }
}, 700); // Runs every 0.7 seconds

console.log("Message B: Script End (timers & interval initiated)");
```

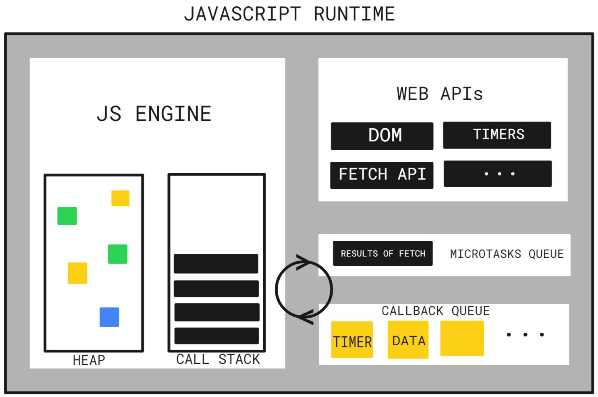
Core Asynchronous Concepts & Patterns

What is the Event Loop?

- What's the Event Loop?
  - The core mechanism in JavaScript's runtime environment that enables non-blocking asynchronous operations, despite JavaScript being single-threaded
- What does it do?
  - Its primary job is to monitor the Call Stack and two Queues: the Callback Queue (or Task Queue) and the Microtask Queue
  - It continuously checks if the Call Stack is empty and then moves functions from the queues to the Call Stack for execution
  - The Event Loop is NOT part of the Javascript Engine itself, but the hosting environment (eg: browser or Node.js)

Key Components in the Asynchronous Model

- **Call Stack:** Where JavaScript functions are executed one by one
- **Web APIs:** Browser features (setTimeout, DOM events, fetch) that handle asynchronous tasks in the background. Once done, they pass their callback functions to a queue
- **Callback Queue (Task Queue):** A First-In, First-Out (FIFO) queue holding callback functions ready to be executed after their associated Web API tasks are complete (e.g., setTimeout callbacks, DOM event handlers)
- **Microtask Queue:** A separate, higher-priority FIFO queue primarily for Promise callbacks (.then(), .catch(), .finally()) and other specific operations like MutationObserver



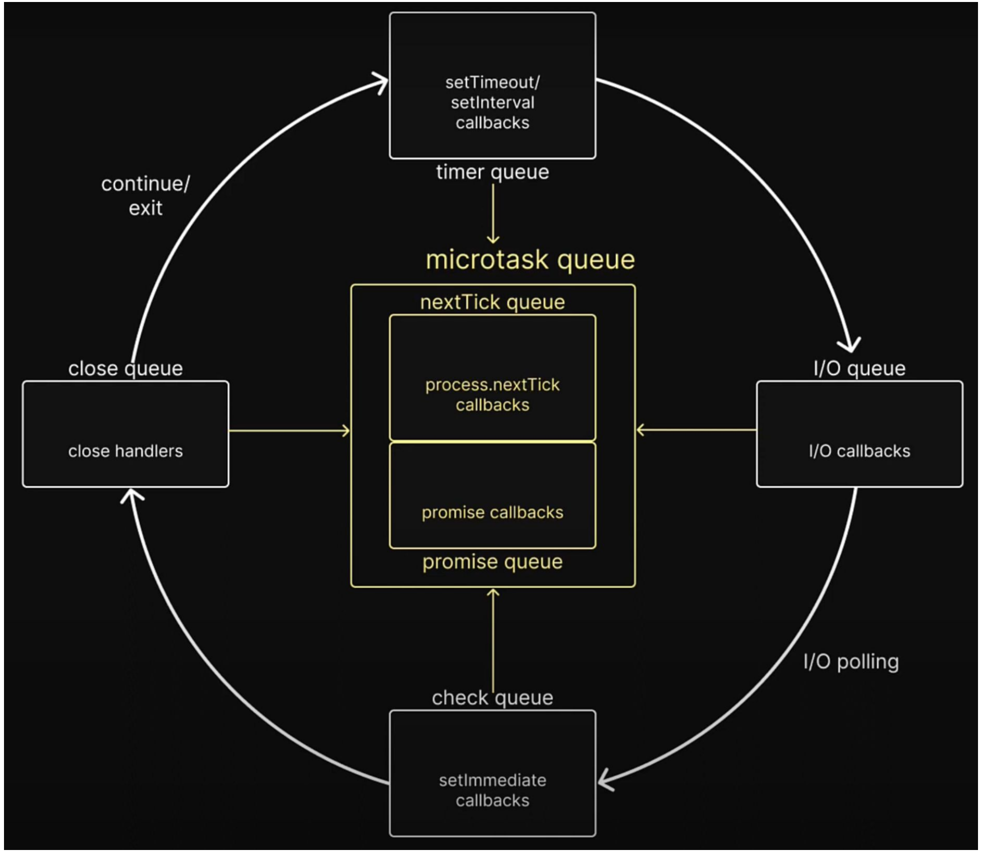
The Event Loop's Process - Step-by-Step

- The Event Loop constantly monitors: Is the Call Stack empty?
  - If the Call Stack is empty, it first checks the Microtask Queue
  - If the Microtask Queue has tasks, it executes all tasks in it, one by one, until the Microtask Queue is empty. New microtasks added during this process are also executed before moving on
  - Only if the Call Stack AND the Microtask Queue are empty, it then checks the Callback Queue (Task Queue)
  - If the Callback Queue has a task, it dequeues the oldest task and pushes it onto the Call Stack for execution
  - The loop repeats this cycle continuously
- Let’s visualize the event loop!

Microtask Queue: The Express Lane

- Holds tasks that need to be executed very soon after the current script or task finishes, but before the browser does things like rendering updates or picking up the next task from the Callback Queue
- Primarily used for:
    - Promise fulfillment/rejection callbacks (.then(), .catch(), .finally()).
    - MutationObserver callbacks (for DOM changes).
    - queueMicrotask() function (explicitly add to microtask queue).
- Key Rule: The Event Loop will process all tasks in the Microtask Queue after the current task completes and before handling any task from the Callback Queue
- Important of this priority: ensures Promise resolutions are handled consistently and quickly

Event Loop: A visual representation



Introduction to Callbacks

Understanding Callbacks

- How would you explain callbacks?
  - Think about real-world scenarios
    - You leave your number for a restaurant and ask them to call you back when a table is ready
    - You ask a friend to text you back about the weekend plans
- A callback function (or simply "callback") is a function that is passed as an argument to another (higher-order) function
  - A higher-order function accepts a function as its input and returns a function
- This higher-order function is then expected to execute (or "call back") the callback function at an appropriate time
  - The callback is invoked often after an asynchronous operation has completed or an event has occurred
- Essentially, you're saying: "Do your main task, and when you're done (or when something happens), run this specific function I'm giving you"

Callbacks in Action: Common Use Cases

- Callbacks are fundamental to many JavaScript patterns:
  - Asynchronous Operations: Handling responses after timers, network requests, file operations
  - Event Handling: Responding to user interactions (clicks, key presses, mouse movements) in the DOM
  - Array Methods: Functions like forEach(), map(), filter() use callbacks to operate on array elements
  - General Purpose: Allowing flexible and reusable code by customizing behavior
- Let’s see callbacks in action!

Callback Hell - The Dark Side of Callbacks

- When multiple asynchronous operations need to be performed in sequence, nesting callbacks within callbacks can lead to deeply indented and hard-to-read code, often called "callback hell"
- Each nested callback depends on the completion of the outer one, creating a pyramid-like structure
- Propagating errors up through multiple nested callbacks is cumbersome (often requiring if (err) checks at every level)

```
asyncOperation1(data, function(result1) {
  asyncOperation2(result1, function(result2) {
    asyncOperation3(result2, function(result3) {
      // ...and so on...
    });
  });
});
```

Promises

What is a Promise?

- What comes to mind when you think of Promise?
  - You make a promise to yourself to consume healthy food
  - The government makes a promise to keep its citizens safe
- A Promise is a JavaScript object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value
- Think of it as a placeholder for a value that you don't have yet, but will get at some point in the future (or an error if something goes wrong)
- A Promise is always in one of three states
  - Pending: The initial state; the asynchronous operation has not yet completed
  - Fulfilled (Resolved): The operation completed successfully, and the Promise now has a resulting value
  - Rejected: The operation failed, and the Promise has a reason for the failure (an error)
- Once a Promise is fulfilled or rejected, it is "settled" and its state will not change

Creating & Consuming Promises: .then() & .catch()

- Creating: new Promise((resolve, reject) => { /* async operation code */ })
  - Inside the executor function, you call resolve(value) on success or reject(error) on failure
- Consuming: We attach callbacks to a Promise using its methods
  - .then(onFulfilledCallback): Called if the Promise is fulfilled. Receives the resolved value
  - .catch(onRejectedCallback): Called if the Promise is rejected. Receives the error/reason
  - .finally(onFinallyCallback): Called when the Promise settles (either fulfilled or rejected). Good for cleanup. (Mention briefly)
- Let's see an example!

Promise Chaining: Escaping Callback Hell

- The real power of Promises shines when sequencing multiple asynchronous operations
  - Inside the executor function, you call resolve(value) on success or reject(error) on failure
- .then() can return:
  - A new Promise: The chain waits for this new Promise to settle
  - A synchronous value: This value is passed to the next .then() in the chain (wrapped in a Promise that immediately fulfills)
- This allows for flat, readable sequences instead of deeply nested callbacks
- A single .catch() at the end of a chain can handle errors from any preceding Promise in the chain
- Let's see an example!

Promise.all(): Handling Multiple Promises

- Promise.all(iterableOfPromises): Takes an array of Promises and returns a new Promise
- This new Promise fulfills when all input Promises have fulfilled. The fulfillment value is an array of the fulfilled values from the input Promises (in the same order)
- It rejects as soon as any one of the input Promises rejects, with the reason of the first Promise that rejected
- Useful for aggregating results from multiple independent asynchronous operations

Promises: Key Advantages

- Solves Callback Hell: Flatter, more readable code for sequential asynchronous operations.
- Improved Error Handling: Centralized error catching with .catch() for chains.
- Better Composability: Promises can be easily combined and chained.
- Clearer State Management: Explicit pending, fulfilled, rejected states
- Food for thought: Promises significantly improve asynchronous JavaScript, but can we make the syntax even more intuitive?

Exercise

Exercise Time!

- In the document:
  - Predict the Output for the three code snippets
  - Event Loop Detective

Async/Await

Introducing async and await

- async and await are reserved keywords in JavaScript that provide a more concise and "synchronous-looking" way to work with Promises
- They are essentially "syntactic sugar" over Promises
  - Syntactic sugar doesn’t introduce new functionality but offer a cleaner syntax for existing Promise-based logic
- async keyword: When placed before a function declaration, it makes the function implicitly return a Promise
  - If the function returns a value, it will be a Promise that resolves with that value
  - If the function throws an error, it will be a Promise that rejects with that error
- await keyword: Can only be used inside an async function. It pauses the execution of the async function until the Promise it's "awaiting" settles (either fulfills or rejects)
  - If the Promise fulfills, await returns the resolved value.
  - If the Promise rejects, await throws the rejection reason (which can be caught by try...catch).

async/await and the Event Loop

- await does not block the entire JavaScript Event Loop or freeze the browser
- It only pauses the execution within the async function where await is used
- While an async function is "awaiting" a Promise, other JavaScript code (including event handlers, other setTimeout callbacks, etc.) can still run, thanks to the Event Loop
- When the awaited Promise settles, the async function's remaining code is scheduled to run (typically as a microtask)
- Let’s see it in action!

That’s it for today.
Questions?

Quiz!

Question 1

- What is the primary role of the Event Loop in JavaScript?
    - A) To execute JavaScript code line by line.
    - B) To manage memory allocation for variables.
    - C) To monitor the Call Stack and Queues, moving callbacks to the Stack.
    - D) To directly handle asynchronous Web API operations.

Correct Answer: C

Question 2

- What happens if a long-running synchronous task executes in JavaScript?
  - A) JavaScript automatically moves it to a separate thread to avoid blocking.
  - B) The JavaScript Call Stack gets blocked, potentially freezing the UI and delaying other operations.
  - C) The browser's Event Loop pauses other JavaScript execution but keeps the UI responsive.
  - D) Web APIs take over the task to ensure smooth performance.

Correct Answer: B

JavaScript is single-threaded. A long-running synchronous task will occupy the Call Stack, preventing any other JavaScript from running and blocking UI updates until it completes

Question 3

- What is "Callback Hell" primarily characterized by?
  - A) Functions that call themselves recursively too many times.
  - B) Using too many Web APIs in a single function.
  - C) Deeply nested callback functions, leading to unreadable and hard-to-maintain code.
  - D) Errors that occur when a callback function is not provided to an asynchronous operation.

Correct Answer: C

Callbacks from Promises (.then(), .catch(), .finally()) are processed via the Microtask Queue, which has higher priority than the Callback Queue (Task Queue) that handles setTimeout, setInterval, and DOM event callbacks

Question 4

- In the JavaScript Event Loop model, which of the following typically has the highest priority for execution once the Call Stack is clear?
    - A) Callbacks from Promise .then() or .catch() methods.
    - B) Callbacks from DOM event listeners (e.g., a button click).
    - C) Callbacks from setTimeout(fn, 0).
    - D) Callbacks from setInterval().

Correct Answer: A

Question 5

- A JavaScript Promise can be in one of three states. Which of the following is NOT a valid state for a Promise?
  - A) Pending
  - B) Fulfilled (Resolved)
  - C) Rejected
  - D) Processing

Correct Answer: D

Question 6

- What is the primary purpose of the await keyword in an async function?
    - A) To define a function that will run asynchronously.
    - B) To pause the execution of the async function until a Promise settles, and then resume with the Promise's result.
    - C) To immediately execute a Promise and get its value.
    - D) To schedule a function to be added to the Callback Queue.

Correct Answer: B