Welcome to Lecture 19!

Agenda

- **Recap**
  - We know how to select, manipulate, and listen for events on DOM elements.
- **Review**: Event Loop Model
- **Error Handling**: Make our code resilient and prevent crashes
- **The fetch API**: Communicate with servers to get and send data
- **Quiz**

The Core Model: Single Thread & Call Stack

- JavaScript executes code using a single thread, meaning it can only perform one operation at a time
- It uses a Call Stack to manage function execution in a Last-In, First-Out (LIFO) order
- When a function is called, it's pushed onto the call stack. When it returns, it's popped off
- **The Challenge** (Blocking): A long-running synchronous task (e.g., a complex calculation) will occupy the Call Stack and block the entire thread. While blocked, the UI freezes, and no other code (like event handlers) can run

Asynchronous Solution: Web APIs, Queues & Event Loop

- **Web APIs**: The browser provides APIs that handle tasks in the background, outside the main JavaScript thread (e.g., setTimeout, DOM Events, fetch)
- **Callback Queue** (or Task Queue): When a Web API task is complete, its associated callback function is placed in this queue, waiting to be executed
- **Event Loop**: This is the core mechanism that enables non-blocking operations. Its job is to continuously monitor both the Call Stack and the Callback Queue
- **The Golden Rule**: The Event Loop moves a callback from the queue to the Call Stack for execution if, and only if, the Call Stack is empty

Let’s Visualize using Latentflip

Exercise 1: Predict the Order

- Given the following code snippet, in what order will the messages appear in the console? Write down your prediction and the "why."

```
console.log('A: First');

setTimeout(() => {
   console.log('B: Timer Callback (0ms)');
}, 0);

Promise.resolve().then(() => {
   console.log('C: Promise Microtask');
});

console.log('D: Last');
```

Error Handling - Building Resilient Code

The Importance of Error Handling

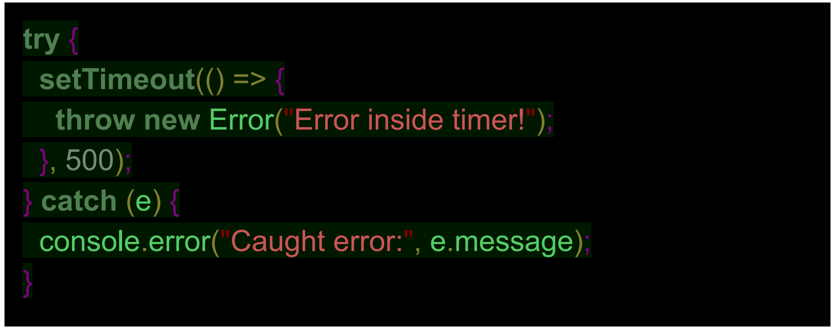
- Scenario: what should happen if a user tries to divide a number by a word? Should our calculator app just crash?
- What is error handling?
- Error handling is the process of anticipating, detecting, and resolving errors in your code to prevent application failure
- Why is it essential?
  - Robustness: Prevents a single error from crashing the entire script
  - User Experience: Allows you to display user-friendly messages instead of a broken page
  - Debugging: Provides a way to catch and log errors, making it easier to find and fix bugs

The try...catch Block for Synchronous Errors

- **try { ... }**: You place the code that might potentially throw an error inside this block. JavaScript will attempt to execute it.
- **catch (error) { ... }**: If (and only if) an error is thrown in the try block, execution immediately jumps to the catch block.
  - The error parameter is an Error object containing details like error.name and error.message.
  - This is where you put your "recovery" logic: log the error, display a UI message, etc.
- **finally { ... }** (Optional): This block of code runs after the try and catch blocks have finished, regardless of whether an error occurred.
  - Use Case: Perfect for cleanup code, like hiding a loading spinner, closing a database connection, or resetting a state variable.

Discuss: try...catch

- What happens if you have a try...catch block around a setTimeout?

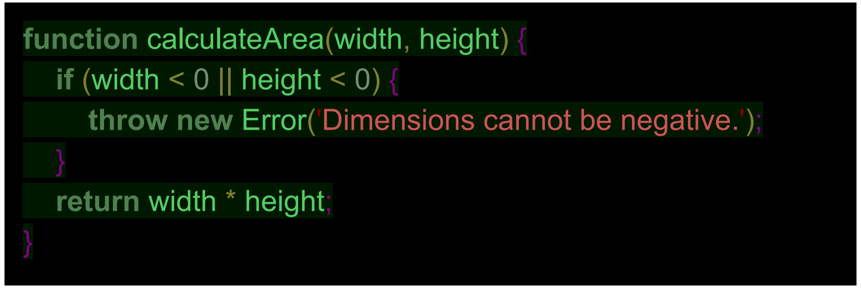


```
try {
  setTimeout(() => {
    throw new Error("Error inside timer!");
  }, 500);
} catch (e) {
  console.error("Caught error:", e.message);
}
```

- **Outcome**: the console.error will not execute
- **Why?** The try...catch block finishes executing synchronously before the asynchronous callback is ever called
    - This is also why Promises have their own built-in .catch() method

Creating Your Own Errors with throw

- The throw statement allows you to create and throw your own custom errors.
- This is powerful for enforcing rules and validating data within your application's logic.
- You can throw any value, but it's best practice to throw an Error object.
- **Syntax**: throw new Error('Your custom error message here');



```
function calculateArea(width, height) {
    if (width < 0 || height < 0) {
        throw new Error('Dimensions cannot be negative.');
    }
    return width * height;
}
```

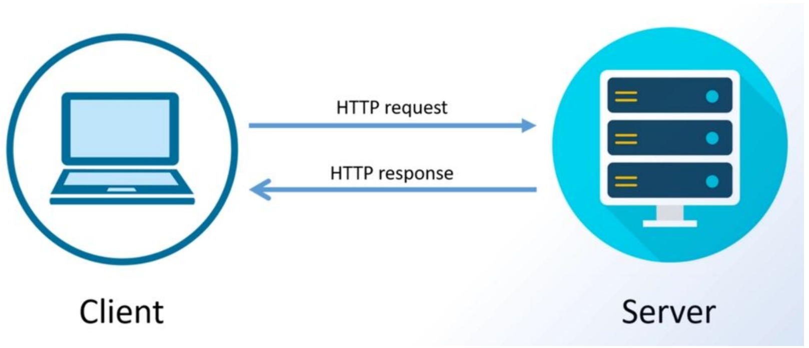
- The **throw** immediately stops the execution of the current function and looks for the nearest catch block up the Call Stack

Let’s Code!

The fetch API & Promises

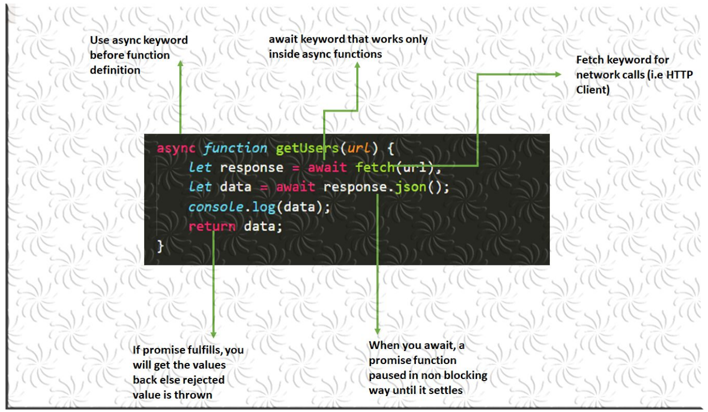
Talking to Servers: The fetch API

- Modern web apps are rarely self-contained; they constantly communicate with servers to get or send data.
- The fetch() API is the modern browser standard for making these network requests.
- It is an asynchronous Web API that is Promise-based, which means it's designed to work perfectly with our asynchronous model.
- A fetch call initiates a network request and immediately returns a Promise that will eventually resolve with the server's response
- **Remember**, we used Promises for a future value in the previous lecture? *fetch* is the perfect use case!



The fetch Promise Chain: A Two-Step Process

- Handling a fetch response is typically a two-step process using chained .then() blocks.
- Step 1: Get the Response Object.
  - fetch('...') returns a Promise that resolves to a Response object. This object contains metadata about the response (like status codes, headers), but not the actual data itself
  - Inside the first .then(), you call a method on the response to parse its body, like response.json()
- Step 2: Get the Data
  - The response.json() method is also asynchronous and returns another Promise!
  - This second Promise resolves with the actual data, now parsed as a JavaScript object
  - You chain a second .then() to access and use this final data


Use async keyword before function definition
await keyword that works only inside async functions
Fetch keyword for network calls (i.e HTTP Client)
If promise fulfills, you will get the values back else rejected value is thrown
When you await, a promise function paused in non blocking way until it settles

```
async function getUsers(url) {
    let response = await fetch(url);
    let data = await response.json();
    console.log(data);
    return data;
}
```

Handling Success (.then) and Failure (.catch)

- The full pattern looks like this:

- The single .catch() at the end is powerful: it handles both network failures (e.g., user is offline, URL is wrong) and errors that might occur during the .json() parsing
- Edge case: fetch only rejects a Promise on a network error
  - If you get a valid response from the server but it's an error status code (like a 404 Not Found or 500 Internal Server Error), the fetch Promise will still fulfill and the .catch block will not run

```
fetch('API_URL')
  .then(response => response.json())          // Step 1
  .then(data => {                              // Step 2
         console.log('Success!', data);
    // Use the data here to update the DOM
  })
  .catch(error => {                        // Handles failures
    console.error('Fetch failed!', error);
    // Update the DOM to show an error message
  });
```

Sending Data with POST Requests

- To send data or use methods other than the default GET, we pass a configuration object as a second argument to fetch
- Key Configuration Properties
  - **method: 'POST'**: Specifies the HTTP method
  - **headers: { 'Content-Type': 'application/json' }**: Tells the server we are sending data in JSON format
  - **body: JSON.stringify(newPostData)**: Contains the data payload. Crucially, the data object must be converted into a JSON string using JSON.stringify()

```
const newPostData = {
  title: 'My Awesome New Post',
  body: 'This is the content of the post.',
  userId: 1
};

fetch('https://jsonplaceholder.typicode.com/posts', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify(newPostData)
})
.then(response => response.json())
.then(createdPost => {
  console.log('Server responded with:', createdPost);
  // Note: JSONPlaceholder simulates the creation and
  // returns an object with the new ID.
})
.catch(error => console.error('Error creating post:', error));
```

Let’s Code!

Exercise

That’s it! Any doubts?

Quiz!

Question 1

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

Correct Answer: A

Question 2

- When you make a successful fetch call, what does the first Promise that fetch returns resolve to?
  - A) The final JSON data
  - B) A string containing the raw HTML
  - C) A Response object with metadata about the response
  - D) Nothing, you must use .catch() to get the data

Correct Answer: C)

It resolves to the Response object first, which then needs to be parsed

Question 3

- Predict the output order of the following code:
  - A) 1, 2, 3, 4, 5
  - B) 1, 5, 3, 4, 2
  - C) 1, 5, 2, 3, 4
  - D) 1, 3, 5, 4, 2

```
console.log(1);
setTimeout(() => console.log(2), 1000);
Promise.resolve().then(() => console.log(3));
setTimeout(() => console.log(4), 0);
console.log(5);
```

Correct Answer: B) 1, 5, 3, 4, 2

Why?

- 1 and 5 are synchronous, log first
- The Promise .then() (3) is a microtask and runs next
- The setTimeout with 0ms delay (4) is a macrotask (callback queue) and runs next
- The setTimeout with 1000ms delay (2) is also a macrotask and runs last, after its timer completes

Question 4

- In the code below, you successfully fetched a response object. What is the correct code to place at // ??? to get the final JSON data in the next step?
  - A) return response;
  - B) return response.data;
  - C) return response.json();
  - D) const data = JSON.parse(response); return data;

```
fetch('...')
  .then(response => {
     // TBD
  })
  .then(data => {
     console.log(data); // We want the final data here
  });
```

Answer: C)

Question 5

- Why is JSON.stringify() necessary when sending a JavaScript object in the body of a POST request with fetch?
  - A) It encrypts the data for security
  - B) It validates the object to make sure it has the right properties
  - C) It adds the necessary Content-Type headers to the request automatically
  - D) The body of an HTTP request must be a string, and JSON.stringify() converts the JavaScript object into a JSON formatted string

Answer: D)

Why? The core reason is the type conversion from a JS object to a string format that servers can universally understand. It does not handle encryption or headers.

Homework: Fetch a Random Dad Joke

- **Task**: Use the icanhazdadjoke.com API to fetch and display a random dad joke when a button is clicked
- API Endpoint: https://icanhazdadjoke.com/
- Important Requirement: For this specific API, you must send an Accept header to tell it you want a JSON response
- Steps:
  - Create a button and a div to display the joke
  - When the button is clicked, fetch from the URL
  - Remember to include the headers option in your fetch call
  - Use the .then() chain to parse the JSON and display the joke property from the data object on your page