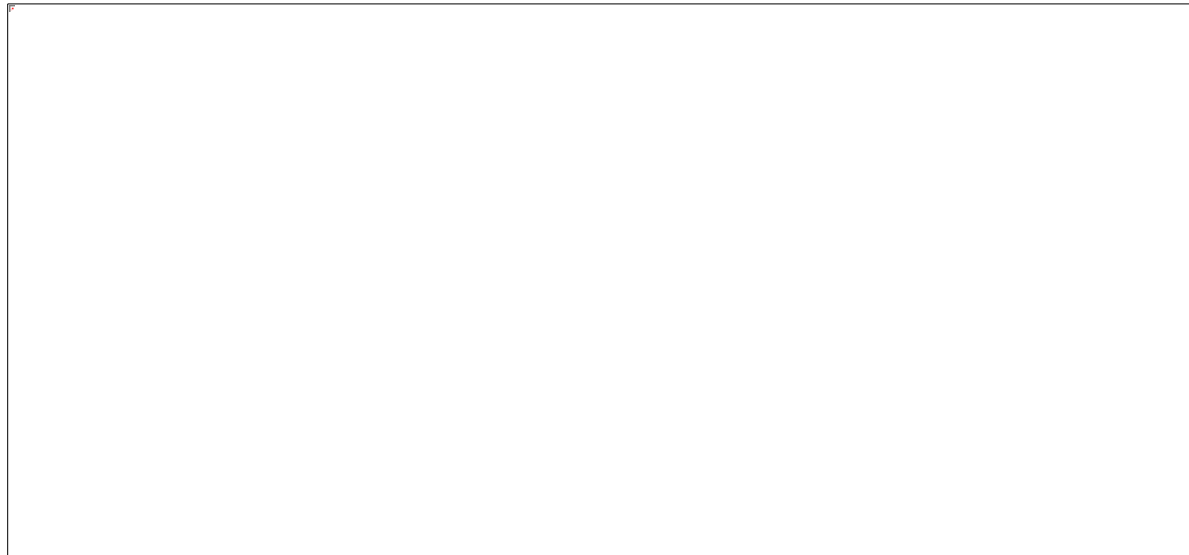


# Introduction to JavaScript

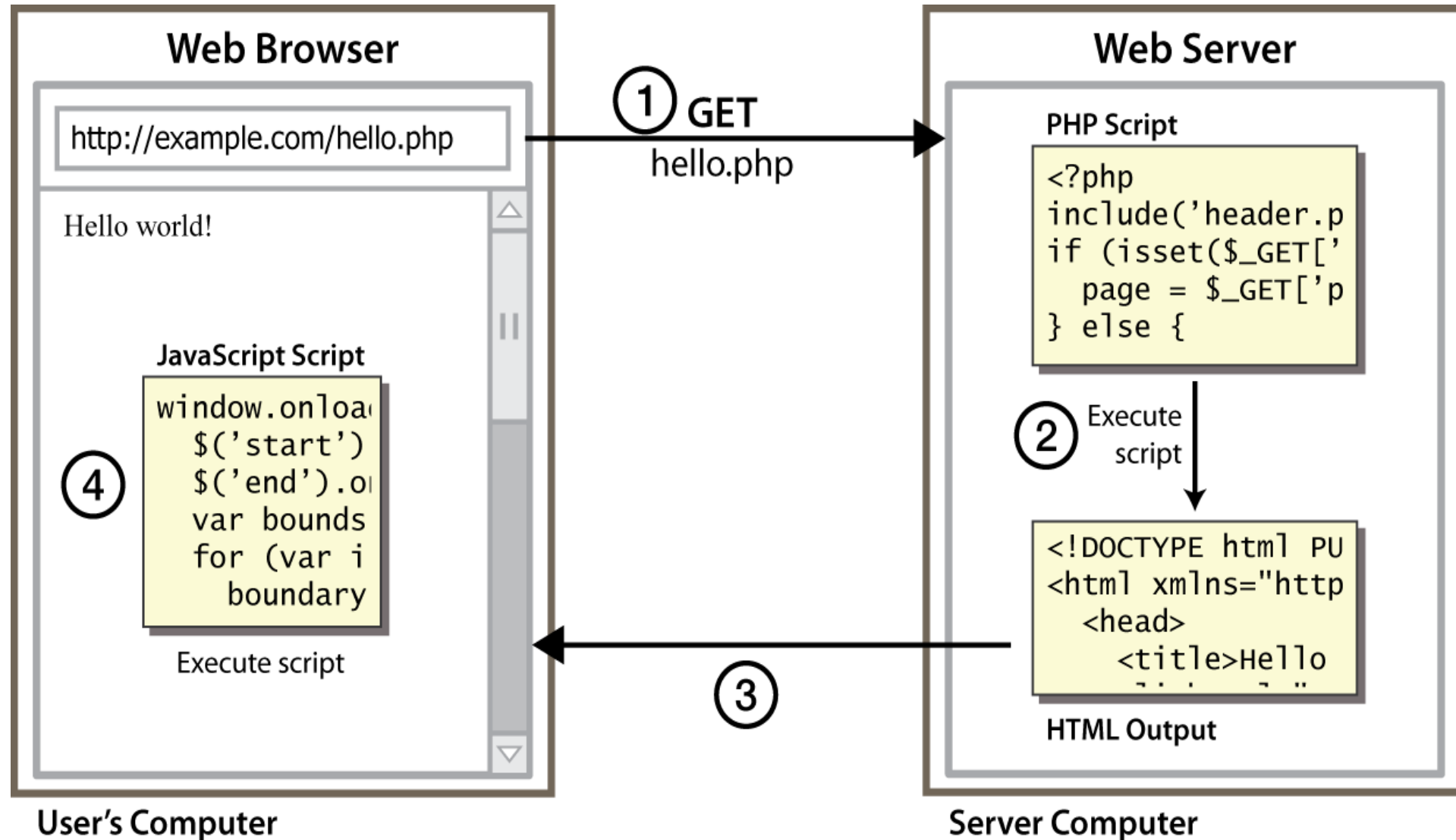
Chapter-3

# What is JavaScript?

- JavaScript is a versatile, dynamically typed, and cross-platform programming language used to make web pages interactive and engaging. It works alongside HTML and CSS to build dynamic web applications with features like animations, clickable buttons, and popup menus.
- Being single-threaded and interpreted, JavaScript executes code line by line and handles one task at a time. Its variables are dynamically typed, meaning their data types are decided at runtime.
- JavaScript supports both client-side (in the browser) and server-side (using environments like Node.js) development, making it ideal for creating responsive websites and real-time applications. It also interacts seamlessly with objects in its environment to control web page behaviour programmatically.



# Client-Side Scripting



# Why use client-side programming?

PHP already allows us to create dynamic web pages. Why also use client-side scripting?

- client-side scripting (JavaScript) benefits:
  - **usability**: can modify a page without having to post back to the server (faster UI)
  - **efficiency**: can make small, quick changes to page without waiting for server
  - **event-driven**: can respond to user actions like clicks and key presses
- server-side programming benefits:
  - **Single Language for Frontend & Backend** - Developers can use the same language (JavaScript) for both client-side and server-side, reducing the learning curve and improving code reusability.
  - **High Performance** - Powered by Google's V8 engine, Node.js executes JavaScript quickly and efficiently.
  - **Large Ecosystem (NPM)** - Access to thousands of ready-to-use packages and libraries through Node Package Manager (NPM) speeds up development.
  - **Real-Time Capabilities** - Perfect for building chat apps, live updates, and collaborative tools thanks to WebSocket and event-based communication.
  - **Cross-Platform Development** - Can run on Windows, macOS, and Linux, and can even power full-stack JavaScript apps using frameworks like Express.js and Next.js.

# Why JavaScript?

- If we write code in Java or C++, every time something happens (like button click or form submit), the browser has to send a request to the server, and the server sends a response back.
- So, to avoid sending requests all the time, they introduced a client-side language (JavaScript) which can handle some actions directly on the browser.
- This reduces server load and makes the web page faster and more interactive
- This approach:
  - Reduces the number of server requests,
  - Decreases server load,
  - Improves website performance, and
  - Makes the experience more dynamic and user-friendly.



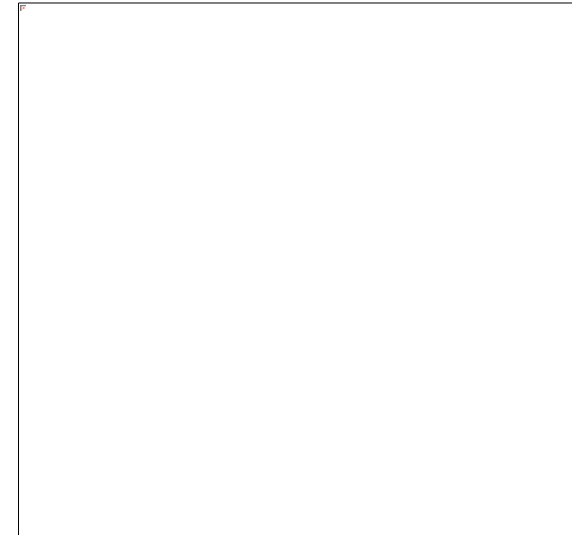
# Real-time Examples

## 1. Online Shopping Website (like Amazon)

When you click “Add to Cart,” the item count in the cart icon increases immediately.

- This is handled by JavaScript in the browser, not the server.
- The server is only contacted when you finally place the order.

Result: Faster updates, less server work.



## 2. Form Validation

When you fill a registration form and forget to enter your email, you instantly see a red message: *“Email is required.”*

- This check happens through JavaScript before sending data to the server.
- Without JavaScript, you’d have to submit the form, wait for the server to respond, and then reload the page with an error.

Result: Instant feedback, no unnecessary server requests.

### 3. Chat Applications (like WhatsApp Web)

When you send a message, it appears instantly in your chat window without reloading the page.

- JavaScript dynamically updates the chat UI.
- It also uses background requests (via APIs or WebSockets) to communicate with the server without interrupting the user.

Result: Real-time communication.

### 4. Social Media (like Instagram)

When you click the “Like ❤️” button, the heart icon changes color immediately.

- That’s JavaScript updating the display instantly.
- Meanwhile, it quietly tells the server that you liked the post (without page reload).

Result: Smooth, interactive experience.

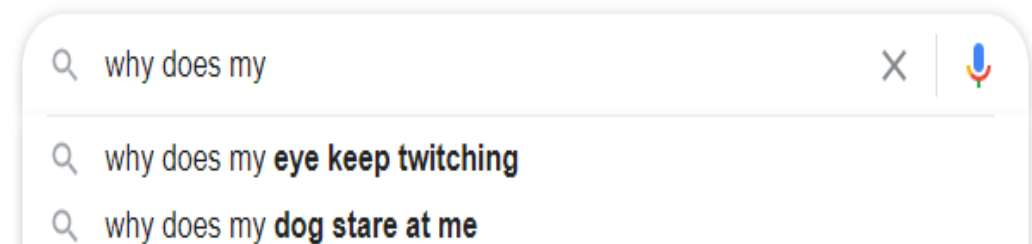


### 5. Google Search Suggestions

As you start typing “how to...” in Google, it shows suggestions like “how to cook,” “how to code,” etc.

- JavaScript sends small background requests to get suggestions and updates them live.

Result: Faster and smarter user experience.



# Example:

Let's say you're filling a signup form.

Action	Without JavaScript (only Java/Python)	With JavaScript
You forget to enter email	Page goes to server → server says "email missing" → reloads page	JS checks instantly in browser and shows message: "Please enter email"
You click "Show Password"	Reloads page	JS instantly toggles visibility
You type a comment	Sends to server and reloads	JS updates instantly

# JavaScript Runtime Environment

JavaScript needs a runtime to execute completely.

## **Browser Runtime provides:**

- Web APIs
- DOM
- Timers
- Event loop

## **Node.js Runtime provides:**

- File system
- OS operations
- Timers
- Event loop

JS alone cannot do these things.

# Real-Time Example (Browser)

```
1 <button id="btn">Click me</button>
2 <script>
3   document.getElementById("btn").addEventListener("click", () => {
4     console.log("Button clicked!");
5   });
6 </script>
```

When user clicks:

- Browser API handles the event
- Callback pushed to event queue
- Event loop sends it to call stack

# Real-Time Example (Node.js)

```
1  const fs = require("fs");
2  console.log("Start");
3  fs.readFile("data.txt", "utf8", (err, data) => {
4      console.log("File read complete");
5  });
6  console.log("End");
```

## Output:

```
Start
End
File read complete
```

Node.js uses system APIs to read files asynchronously.

# Javascript vs Java

JAVASCRIPT	JAVA
Object-oriented. No distinction between types of objects. Inheritance is through the prototype mechanism, and properties and methods can be added to any object dynamically.	Class-based. Objects are divided into classes and instances with all inheritance through the class hierarchy. Classes and instances cannot have properties or methods added dynamically.
Variable data types are not declared (dynamic typing, loosely typed).	Variable data types must be declared (static typing, strongly typed).
Cannot automatically write to hard disk.	Can automatically write to hard disk.

# Linking to a JavaScript file: script

```
<script src="filename" type="text/javascript"></script>
```

- script tag should be placed in HTML page's head
- script code is stored in a separate .js file
- JS code can be placed directly in the HTML file's body or head (like CSS)
  - but this is bad style (should separate content, presentation, and behavior)

# ES6+ Features

## 1. let and const

- Used to declare variables.
- let - value can change
- const - value cannot change

Example:

```
1  let age = 25;  
2  const PI = 3.14;
```

## 2. Arrow Functions

- Short, modern way to write functions.

Example:

```
1  const add = (a, b) => a + b;
```

### 3. Template Literals

- Use backticks (`) to create dynamic strings.

Example:

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

### 4. Default Parameters

Give functions default values when no arguments are passed.

Example:

```
function greet(name = "Guest") {  
  console.log("Hello " + name);  
}
```

### 5. Destructuring

Extract values from arrays or objects easily.

Array:

```
const [x, y] = [10, 20];
```

Object:

```
const { name, age } = user;
```

## 6. Spread Operator (...)

Expands arrays or objects.

Example:

```
const arr = [...arr1, 3, 4];
```

## 7. Rest Parameters

Collect multiple arguments into one array.

Example

```
1 function sum(...nums) { return nums.reduce((a,b)=>a+b); }
```

## 8. Classes

Cleaner way to create objects and use OOP concepts.

```
1 class Person {  
2   constructor(name) { this.name = name; }  
3   greet() { console.log("Hi " + this.name); }  
4 }
```

## 9. Promises

Used for handling asynchronous operations (like fetching data).

Example:

```
new Promise(resolve => setTimeout(() => resolve("Done"), 1000));
```

## 10. async / await

Makes asynchronous code look simple and readable.

Example:

```
1  async function run() {  
2    let r = await Promise.resolve("Hello");  
3    console.log(r);  
4  }
```

## 11. Modules (import/export)

Split code into multiple files.

Example:

```
1
2 // file1.js
3 export const name = "JS";
4
5 // file2.js
6 import { name } from "./file1.js";
```

## 12. Enhanced Object Literals

Shorter syntax for objects.

Example:

```
1 const user = {
2   age,
3   greet() { console.log("Hello") }
4 };
```

# JAVASCRIPT ARCHITECTURE

## CHAPTER-3

# Working Model of JavaScript.

- JavaScript Architecture explains how JavaScript executes code, how it handles tasks, events, asynchronous operations, and how the browser or Node.js environment helps JavaScript run smoothly.

## 1. **Single Threaded** - It can do only one task at a time.

- Think of JavaScript like:

- A single person working at a desk.

It picks up one paper, finishes it (or sends it to someone else), then picks up the next one.

## 2. **Synchronous + Asynchronous**

### • **Synchronous**

- Tasks happen step by step
  - Next line runs only after the previous line finishes

```
1 console.log("Start");  
2 console.log("Middle");  
3 console.log("End");
```

- Output will always be:

```
Start  
Middle  
End
```

- **Asynchronous**
  - JavaScript can start a task and continue with others without waiting for the task to finish.
  - These tasks run in the background (like timers, network calls).

```
1  console.log("Start");  
2  
3  setTimeout(() => {  
4    console.log("Delayed Task");  
5  }, 2000);  
6  
7  console.log("End");
```

Output:

```
Start  
End  
Delayed Task
```

**3. Event-driven** - This means JavaScript reacts to events.

Examples of events:

- Clicking a button
- Typing in a textbox
- Timer completes
- API response arrives

JavaScript waits and listens for these events, and when an event happens, it runs the callback function.

```
1 button.addEventListener("click", () => {  
2   console.log("Button clicked!");  
3 });
```

# JavaScript Execution Context

- The environment where JavaScript code runs.
  - It decides how variables, functions, and code are executed.
- Whenever JavaScript runs anything, it creates an Execution Context.

There are two main types:

## 1. Global Execution Context (GEC)

This is the default context.

It is created:

- When your JavaScript file starts running.
- Only once.

Inside Global Execution Context:

- this refers to the window object (in browser)
- A global memory is created for variables and functions

```
var x = 10;  
function greet() {  
    console.log("Hello");  
}
```

When this code starts, JavaScript creates:

- Global Memory → stores x and greet
- Global Thread → runs the code line by line

## 2. Function Execution Context (FEC)

Every time a function is called, a new context is created.

Example:

```
function add(a, b) {  
    return a + b;  
}  
add(5, 10); // creates its own execution context
```

When a function runs, JavaScript creates:

- Local Memory (variables inside function)
- Local this
- Arguments object

When the function finishes →

That function's Execution Context is destroyed.

# Phases of Execution Context

Each Execution Context has two phases:

## 1. Creation Phase (Memory Creation Phase)

- JavaScript scans the code before running it.

During this phase:

- It creates variables declared with var and sets them to undefined
- Functions are stored in memory as a whole
- this and arguments are created

```
1 console.log(x); // undefined
2 var x = 5;
```

- Why undefined?

Because in the creation phase, JavaScript allocates memory for x with value undefined.

## 2. Execution Phase (Code Execution Phase)

Now JavaScript runs the code line by line and updates values.

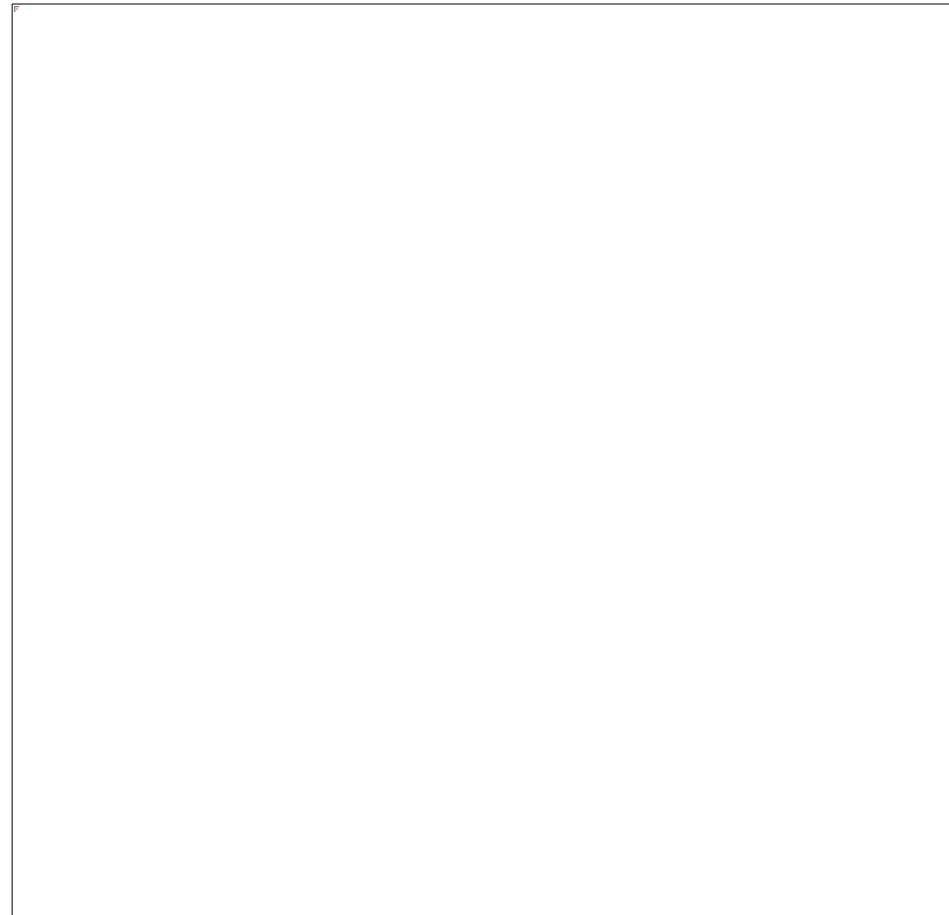
- Assigns values to variables
- Executes functions
- Calculates results

```
1  var x = 5;
2  var y = 10;
3
4  function multiply(a, b) {
5      return a * b;
6  }
7
8  var result = multiply(x, y);
9  console.log(result);
```

# JavaScript Engine

A JavaScript Engine is a program inside the browser (or Node.js) that reads, interprets, and runs JavaScript code.

- Common JS engines:
  - V8 → Used in Chrome & Node.js
  - SpiderMonkey → Firefox
  - Chakra → Old Microsoft Edge
- Every JS engine has two main components:



## 1. Memory Heap

- A place where JavaScript stores data
- Stores values.
- Works like a big storage area where values are kept until they are needed

### Example:

```
1 let user = {language:"javascript"}
```

This object goes into the heap.

## 2. Call Stack

- Manages the order in which functions run
- Follows Last In, First Out (LIFO)
- When a function is called, it goes on the stack
- When it finishes, it is removed from the stack

### Example:

```
1  function a() { b(); }  
2  function b() { console.log("Hello"); }  
3  a();
```

### Stack order:

- global
- a()
- b()
  - then b() removed
  - then a() removed

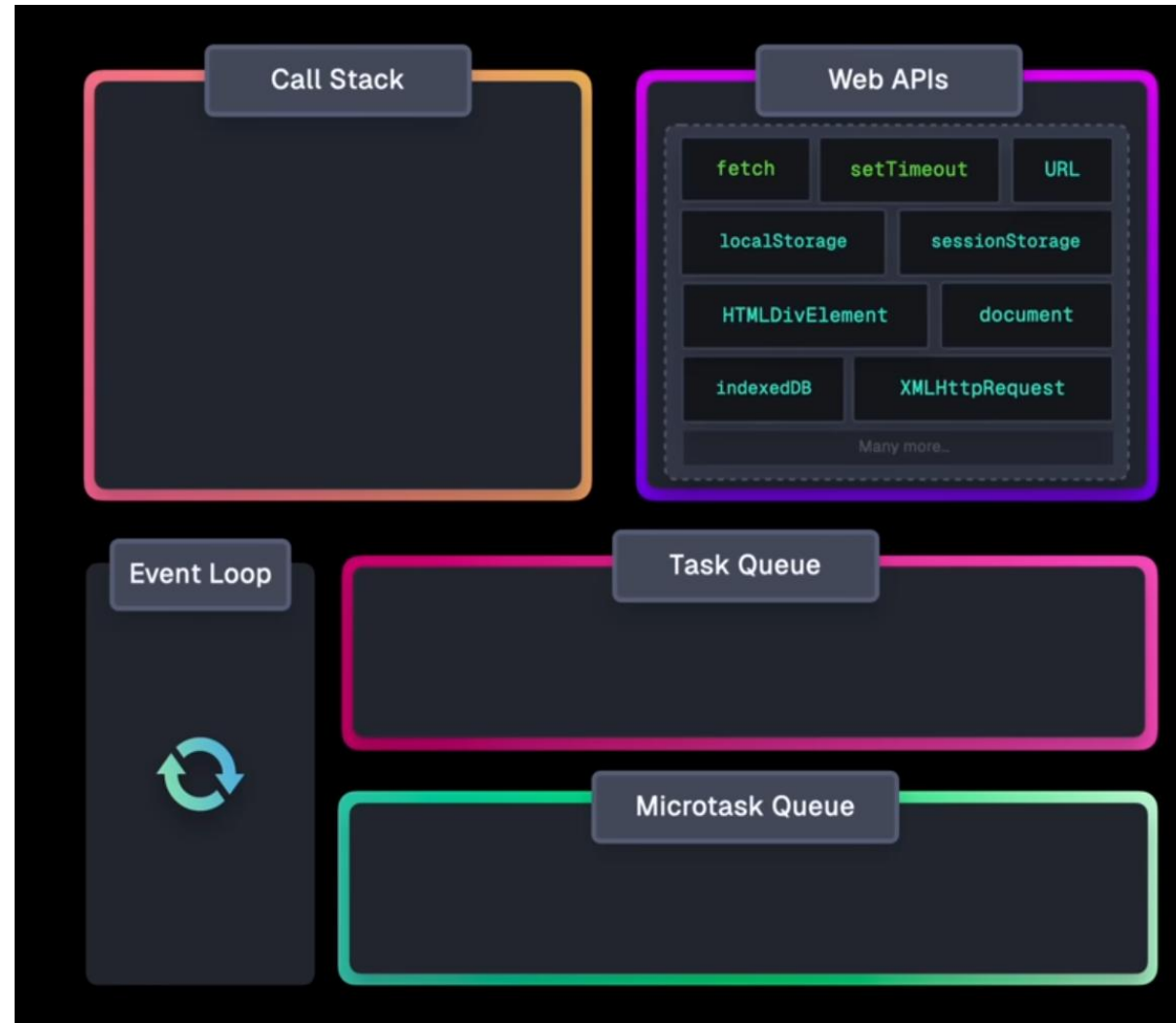
# JavaScript Runtime Architecture

- JavaScript is single-threaded - it executes one line at a time.
- But your app needs to do multiple things without blocking the page:
  - Wait for API responses
  - Handle button clicks
  - Wait for timers
  - Process promises
  - Perform async operations

JavaScript achieves this using:

- Call Stack
- Web APIs / Node APIs
- Callback Queue (Task Queue)
- Microtask Queue (Promise Queue)
- Event Loop

# Architecture



# 1. Call Stack (Where JS Executes Code)

JS runs code line-by-line inside the call stack.

**Example:**

```
1  function greet() {  
2    console.log("Hello");  
3  }  
4  
5  greet();  
6  console.log("End");
```

Order of execution:

1. greet() pushed to call stack
2. console.log("Hello")
3. Remove greet()
4. console.log("End")

The call stack executes things immediately.

## 2. Web APIs / Node APIs (For Async Work)

These are browser-provided or Node-provided features:

- setTimeout
- fetch()
- DOM events
- setInterval

Node.js: filesystem, HTTP, timers

They run outside JS, so JS doesn't get blocked.

Example:

```
1 console.log("Start");
2
3 setTimeout(() => {
4   console.log("Timer done");
5 }, 2000);
6
7 console.log("End");
```

Flow:

- JS sends setTimeout to Web APIs
- Web APIs wait 2 sec
- After time completes → callback goes to Callback Queue
- Event loop sends it to call stack

### 3. Callback Queue (Task Queue)

All async callbacks like:

- setTimeout
- setInterval
- DOM events

go into this queue.

They wait here until the call stack is empty.

**Example:**

```
1  setTimeout(() => console.log("Task done"), 0);  
2  console.log("Normal log");
```

**Output:**

```
Normal log  
Task done
```

## 4. Microtask Queue (Higher Priority Queue)

These tasks run immediately after the current script finishes, and before any timer or event callback.

- `Promise.then()`
- `async/await` (after an `await` finishes)
- `queueMicrotask()`

Example:

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

## Output:

```
Start
End
Promise   ← Microtask Queue runs first
Timeout   ← Callback Queue runs after microtasks
```

## Why does this happen?

- The JavaScript engine finishes all normal code first.
- Then the Event Loop checks Microtask Queue first.
- Only when it is completely empty → it runs the Callback Queue.

That's why:

- Promise runs first
- setTimeout() runs later

# Event Loop

The Event Loop is like a traffic police officer who controls the flow of JavaScript tasks.

## What it does

1. Keeps checking if the call stack is empty
  - The call stack is where JavaScript runs your code.
  - If it's busy → event loop waits
  - If it becomes empty → event loop sends new tasks
2. First sends tasks from the Microtask Queue
  - Microtasks = Promises, async/await (These are treated as VIP tasks.)
  - Event loop always runs them first.
3. Then sends tasks from the Callback Queue
  - Callback Queue = setTimeout, setInterval, DOM events (These are normal tasks.)
  - They run only after all microtasks are finished.

# Example

```
1  console.log("1: Start");
2
3  setTimeout(() => {
4    console.log("4: Timeout finished");
5  }, 0);
6
7  Promise.resolve().then(() => {
8    console.log("3: Promise resolved");
9  });
10
11 console.log("2: End");
```

# Internal Workflow

**Step 1:** `console.log("1: Start")`

- This is synchronous code.
- It goes to the Call Stack immediately.

Output: 1: Start

**Step 2:** `setTimeout(..., 0)`

- JS sees `setTimeout`.
- JavaScript does not run the function immediately.

What happens:

1. JS sends the timer to Web APIs
2. Web API counts 0ms
3. When finished → callback goes to the Callback Queue (Not executed yet!)

The callback waits in the callback queue until the call stack is empty and all microtasks finish.

**Step 3:** `Promise.resolve().then(...)`

- JS sees a Promise.
- Its `.then()` callback is stored in the Microtask Queue (This queue has higher priority)
- It will run immediately after all synchronous code ends.

**Step 4:** `console.log("2: End")`

- Synchronous code
- Runs immediately in the Call Stack

Output: 2: End

**Now the Call Stack is Empty**

Synchronous code is done.

**Event Loop starts working.**

3: Promise resolved

**Step 5:** Event Loop → Check Microtask Queue

- Microtasks are always executed before callbacks.
- So event loop picks:

```
console.log("3: Promise resolved")
```

Output:

```
3: Promise resolved
```

**Step 6:** Event Loop → Check Callback Queue

- Now microtasks are done.
- The event loop looks into the Callback Queue → finds the setTimeout callback.
  - Moves it to the Call Stack
  - Executes it

Output:

```
4: Timeout finished
```

## Final Output:

```
1: Start  
2: End  
3: Promise resolved  
4: Timeout finished
```