**Module 35 35-1 What really is JavaScript?**

**High Abstraction in JavaScript (Brief Overview)**

JavaScript provides **high-level abstraction**, meaning developers can perform complex operations **without needing to manage low-level details** like memory allocation or hardware interactions.

**Key Aspects of High Abstraction in JavaScript**

1. **Automatic Memory Management (Garbage Collection)**
   * JavaScript **automatically manages memory**, so developers don’t need to manually allocate or free memory.

let obj = { name: "Alice" }; // Memory allocated automatically

obj = null; // Garbage collector frees memory when no longer needed

1. **Built-in High-Level Functions**
   * JavaScript provides **methods and APIs** that abstract complex operations.

let numbers = [1, 2, 3, 4, 5];

let doubled = numbers.map(num => num \* 2); // Abstracts loop logic

console.log(doubled); // [2, 4, 6, 8, 10]

1. **Event-Driven and Asynchronous Programming**
   * JavaScript abstracts **asynchronous operations** using **callbacks, promises, and async/await**.

fetch("https://api.example.com/data")

.then(response => response.json())

.then(data => console.log(data)) // Abstracts HTTP request handling

.catch(error => console.error(error));

1. **Object-Oriented & Functional Abstractions**
   * JavaScript supports **OOP (Classes, Prototypes)** and **Functional Programming** for higher abstraction.

class Person {

constructor(name) {

this.name = name;

}

greet() {

console.log(`Hello, I am ${this.name}`);

}

}

const user = new Person("Bob");

user.greet(); // Abstracts object behavior

**Advantages of High Abstraction in JavaScript**

✅ **Simplifies Complex Operations** → No need to handle low-level tasks like memory management.  
✅ **Enhances Productivity** → Developers can focus on business logic instead of implementation details.  
✅ **Improves Readability & Maintainability** → Code is cleaner and easier to understand.

**Conclusion**

JavaScript’s **high-level abstraction** makes it **developer-friendly**, enabling efficient coding with minimal low-level concerns. 🚀

**JIT (Just-In-Time) in JavaScript - Brief Overview**

Just-In-Time (JIT) compilation is a technique used by JavaScript engines (like V8, SpiderMonkey) to improve execution speed by compiling JavaScript code into machine code at runtime.

How JIT Works in JavaScript

Parsing → JavaScript code is converted into an Abstract Syntax Tree (AST).

Bytecode Generation → The interpreter (e.g., Ignition in V8) converts AST into bytecode.

JIT Compilation → The JIT compiler (e.g., TurboFan in V8) detects frequently executed code (hot code) and compiles it into optimized machine code.

Advantages of JIT

✅ Faster Execution → Compiles hot code into machine code for better performance.

✅ Adaptive Optimization → Code is dynamically optimized based on execution patterns.

Disadvantages of JIT

⚠️ Warm-up Time → Initial execution might be slower due to compilation overhead.

⚠️ Higher Memory Usage → Compiled machine code consumes extra memory.

Conclusion

JIT makes JavaScript faster and more efficient by dynamically compiling frequently used code, making it crucial for modern JavaScript engines like V8 (Chrome, Node.js) and SpiderMonkey (Firefox). 🚀

**JavaScript as a Multi-Paradigm Language (Brief Overview)**

JavaScript is a versatile programming language that supports multiple programming paradigms, making it adaptable to different programming styles and project requirements.

**Key Paradigms Supported:**

1. **Procedural Programming**
   * Simple, top-down execution
   * Functions as basic building blocks
   * Example: Basic scripts with functions called in sequence
2. **Object-Oriented Programming (OOP)**
   * Prototype-based inheritance (rather than classical)
   * ES6 introduced class syntax (syntactic sugar over prototypes)
   * Encapsulation, inheritance, and polymorphism possible
3. **Functional Programming**
   * First-class functions (can be assigned, passed as arguments, returned)
   * Higher-order functions (functions that operate on other functions)
   * Pure functions encouraged (same input → same output, no side effects)
   * Immutability patterns (though not enforced)
4. **Event-Driven Programming**
   * Callbacks and event listeners
   * Asynchronous programming model
   * Promises and async/await syntax
5. **Imperative & Declarative Styles**
   * Can write both how-to (imperative) and what-to-do (declarative) code
   * Declarative examples: array methods like map(), filter()

**Why This Matters:**

* Flexibility to choose the right approach for each problem
* Can mix paradigms within a single application
* Evolved from simple scripting to supporting complex application architectures
* Functional programming features have become increasingly important in modern JS frameworks

This multi-paradigm nature contributes to JavaScript's popularity and wide applicability across different types of projects.

**JavaScript as a Multi-Paradigm Language (Brief Overview)**

JavaScript is a **multi-paradigm programming language**, meaning it supports different programming styles, allowing developers to choose the best approach for their needs.

**Main Paradigms in JavaScript**

1. **Imperative Programming** → Writing step-by-step instructions.

let sum = 0;

for (let i = 1; i <= 5; i++) {

sum += i;

}

console.log(sum); // 15

1. **Procedural Programming** → Organizing code into functions.

function greet(name) {

console.log("Hello, " + name);

}

greet("Alice");

1. **Object-Oriented Programming (OOP)** → Using objects and classes.

class Person {

constructor(name) {

this.name = name;

}

greet() {

console.log(`Hello, I am ${this.name}`);

}

}

const person = new Person("Bob");

person.greet();

1. **Functional Programming (FP)** → Using pure functions and avoiding side effects.

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

console.log(add(5, 10)); // 15

**Why Multi-Paradigm?**

✅ **Flexible** → Developers can choose **OOP, FP, or procedural** styles.  
✅ **Scalable** → Supports modular, maintainable, and reusable code.

**Conclusion**

JavaScript’s **multi-paradigm nature** makes it a **versatile** language, suitable for different programming styles, from small scripts to large applications. 🚀

**Prototype-Based JavaScript (Brief Overview)**

JavaScript is a **prototype-based** language, meaning it does not use traditional **class-based** inheritance like Java or Python. Instead, it uses **prototypes** to define properties and behaviors that objects can inherit.

**Key Concepts of Prototype-Based JavaScript**

1. **Prototype Chain** → Every JavaScript object has an internal link (\_\_proto\_\_) to another object, called its **prototype**.
2. **Inheritance** → Objects inherit properties and methods from their prototype.
3. **Prototype Property** → Functions in JavaScript have a special prototype property that is used to create inheritance.

**Example of Prototype-Based Inheritance**

// Constructor function

function Person(name) {

this.name = name;

}

// Adding a method to the prototype

Person.prototype.greet = function () {

console.log(`Hello, I am ${this.name}`);

};

// Creating an object

const person1 = new Person("Alice");

person1.greet(); // Output: Hello, I am Alice

**Prototype Chain Example**

console.log(person1.\_\_proto\_\_ === Person.prototype); // true

console.log(Person.prototype.\_\_proto\_\_ === Object.prototype); // true

console.log(Object.prototype.\_\_proto\_\_); // null (end of chain)

**Advantages of Prototype-Based Inheritance**

✅ **Memory Efficient** → Methods are shared among objects instead of being duplicated.  
✅ **Dynamic** → Prototypes can be modified at runtime.

**Conclusion**

JavaScript’s **prototype-based inheritance** allows flexible and efficient object creation, making it different from **class-based** languages. 🚀

**Dynamically Typed JavaScript (Brief Overview)**

JavaScript is a **dynamically typed** language, meaning **variable types are determined at runtime** rather than being explicitly declared.

**Key Characteristics of Dynamic Typing in JavaScript**

1. **No Type Declaration** → Variables do not have fixed types.

let x = 10; // x is a number

x = "Hello"; // Now x is a string

x = true; // Now x is a boolean

1. **Type Coercion** → JavaScript automatically converts data types when needed.

console.log("5" + 2); // "52" (string)

console.log("5" - 2); // 3 (number)

1. **Flexible Function Arguments** → Functions can accept any type of argument.

function print(value) {

console.log(value);

}

print(100); // Number

print("Hello"); // String

print(true); // Boolean

**Advantages of Dynamic Typing**

✅ **More Flexibility** → No need to declare variable types explicitly.  
✅ **Faster Development** → Quick prototyping without type constraints.

**Disadvantages of Dynamic Typing**

⚠️ **More Runtime Errors** → Bugs like undefined or NaN can occur.  
⚠️ **Harder to Debug** → Unexpected type changes may cause issues.

**Conclusion**

JavaScript’s **dynamic typing** makes it **flexible and easy to use**, but it also requires careful handling to avoid unexpected type-related bugs. 🚀

**Garbage Collection in JavaScript (JS GC - Brief Overview)**

Garbage collection (GC) in JavaScript is **an automatic memory management process** that removes unused objects from memory to free up space. This prevents memory leaks and improves performance.

**How Garbage Collection Works in JavaScript**

JavaScript engines (like V8 in Chrome and Node.js) use a **garbage collector** that automatically detects and removes objects that are **no longer accessible** in the program.

**1. Memory Allocation**

When objects are created, memory is allocated:

let obj = { name: "Alice" }; // Memory is allocated for obj

**2. Losing References**

When an object is no longer referenced, it becomes **eligible for garbage collection**:

obj = null; // The object is now unreachable and will be collected

**Garbage Collection Algorithm: Mark-and-Sweep**

Most JavaScript engines use the **Mark-and-Sweep** algorithm:

1. **Mark** → The garbage collector marks all **reachable objects** (i.e., objects still in use).
2. **Sweep** → It removes **unmarked objects** (unreachable ones) from memory.

Example:

function createUser() {

let user = { name: "Bob" }; // Allocated memory

return user; // Still referenced, so NOT collected

}

let user1 = createUser();

user1 = null; // Now eligible for garbage collection

**Types of Memory Leaks in JavaScript**

1. **Global Variables (Unintended References)**

function leak() {

leakedVar = "This is a memory leak"; // Becomes a global variable

}

leak(); // Never gets collected

✅ Always use let, const, or var to avoid accidental global variables.

1. **Dangling References in Closures**

function outer() {

let largeData = new Array(1000000); // Large memory usage

return function inner() {

console.log(largeData.length); // Closure retains reference

};

}

let leak = outer(); // `largeData` is not garbage collected

✅ Nullify references when no longer needed: leak = null;

**How to Prevent Memory Leaks in JavaScript**

✅ **Use let and const instead of implicit global variables**  
✅ **Set unused variables to null or undefined**  
✅ **Remove event listeners when no longer needed**  
✅ **Avoid unnecessary object references in closures**

**Conclusion**

JavaScript's **automatic garbage collection** improves memory management, but developers should still write efficient code to prevent **memory leaks** and optimize performance. 🚀

**35-2 JavaScript Engine V8 Internal mechanism**

[V8 JavaScript engine](https://v8.dev/)

<https://v8.dev/>

**V8 JavaScript Engine - Brief Overview**

The **V8 engine** is a high-performance, open-source JavaScript and WebAssembly engine developed by Google, used in **Chrome, Node.js, Deno, and other runtime environments**.

**Key Features of V8**

**1. Just-In-Time (JIT) Compilation**

* Uses a multi-tier compilation approach:
  + **Ignition**: Fast interpreter for initial execution.
  + **TurboFan**: Optimizing compiler for hot code (converts JS to efficient machine code).
  + **SparkPlug**: Mid-tier compiler for faster warm-up.

**2. Memory Management (Garbage Collection)**

* **Generational Garbage Collection**:
  + **Young Generation (Scavenger)** – Fast cleanup of short-lived objects.
  + **Old Generation (Mark-Sweep-Compact)** – Handles long-lived objects.
* Uses **Orinoco** (concurrent and parallel GC) to reduce pauses.

**3. Hidden Classes & Inline Caching (Optimizations)**

* **Hidden Classes**: Dynamically creates class-like structures for objects to speed up property access.
* **Inline Caching**: Caches frequent property lookups to avoid repeated searches.

**4. WebAssembly Support**

* Compiles **Wasm** to near-native speed, enabling high-performance apps (games, video editing, etc.).

**5. ECMAScript Compliance**

* Implements the latest **ES6+ features** (Promises, Async/Await, Classes, Modules, etc.).

**Where V8 is Used?**

✔ **Google Chrome & Chromium-based browsers** (Edge, Brave).  
✔ **Node.js** (server-side JavaScript runtime).  
✔ **Deno & Bun** (modern JS/TS runtimes).  
✔ **Electron** (desktop apps like VS Code, Slack).

**Performance Comparison**

| **Engine** | **Used In** | **Key Strengths** |
| --- | --- | --- |
| **V8** | Chrome, Node | Best all-round performance, JIT optimizations |
| **SpiderMonkey** | Firefox | Good balance of speed & memory |
| **JavaScriptCore** | Safari | Energy-efficient (Apple-optimized) |

**Example: How V8 Optimizes Code**

function add(a, b) {

return a + b; // TurboFan compiles to machine code if types are stable

}

add(1, 2); // Fast (numbers)

add("a", "b"); // Slower (deoptimizes if types change)

**Conclusion**

V8’s **JIT compilation, garbage collection, and hidden class optimizations** make JavaScript execution extremely fast, powering modern web apps and server-side runtimes like Node.js. 🚀

**Execution Context and Call Stack in JavaScript**

**1. Execution Context**

In JavaScript, an **Execution Context (EC)** is an environment in which JavaScript code is executed. It contains everything necessary to execute a piece of code, such as variables, functions, and the this keyword.

**Types of Execution Contexts**

1. **Global Execution Context (GEC)**
   * Created when the JavaScript file or script starts executing.
   * It creates the global this object (window in browsers, global in Node.js).
   * Only one GEC exists at a time.
2. **Function Execution Context (FEC)**
   * Created whenever a function is invoked.
   * Each function call has its own execution context.
   * Multiple FECs can exist at the same time.
3. **Eval Execution Context**
   * Created when eval() is used (not recommended due to security risks).

**Phases of Execution Context**

Every execution context has two phases:

1. **Creation Phase (Memory Allocation)**
   * Creates the **variable environment** (hoists variables with var and functions).
   * Sets up the **scope chain** and binds this.
2. **Execution Phase**
   * Executes the code line by line.
   * Assigns values to variables and executes functions.

**2. Call Stack**

The **Call Stack** is a data structure that keeps track of execution contexts in a **Last In, First Out (LIFO)** order.

**How the Call Stack Works**

1. **When the script starts** → The Global Execution Context (GEC) is pushed onto the stack.
2. **When a function is called** → A new Function Execution Context (FEC) is created and pushed onto the stack.
3. **When a function returns** → The function's execution context is popped off the stack.
4. **When the script finishes** → The GEC is removed, and the program stops.

**Example of Call Stack in Action**

function first() {

console.log("First function");

second();

}

function second() {

console.log("Second function");

third();

}

function third() {

console.log("Third function");

}

first();

**Call Stack Process**

1. first() is called → **Push first() onto the stack**
2. Inside first(), second() is called → **Push second() onto the stack**
3. Inside second(), third() is called → **Push third() onto the stack**
4. third() executes and finishes → **Pop third() off the stack**
5. second() finishes → **Pop second() off the stack**
6. first() finishes → **Pop first() off the stack**
7. The stack is empty, and execution is complete.

**Call Stack Overflow**

If a function keeps calling itself indefinitely (recursion without a base case), the call stack will overflow.

Example:

function recursive() {

recursive();

}

recursive(); // Stack Overflow Error

**Summary**

* **Execution Context**: The environment in which JavaScript code runs, including variables, functions, and this.
* **Call Stack**: A LIFO structure that tracks execution contexts during function calls.
* **Execution Phases**: Creation phase (memory allocation) and execution phase (code execution).
* **Stack Overflow**: Happens when the call stack grows too large due to deep recursion.

Let me know if you need further clarification! 🚀

**Execution Context and Call Stack in JavaScript**

**Execution Context**

An execution context is an abstract concept that holds information about the environment where JavaScript code is executed. There are three types of execution contexts:

1. **Global Execution Context**: Created when the script first starts running
2. **Function Execution Context**: Created when a function is invoked
3. **Eval Execution Context**: Created inside an eval function (rarely used)

**Components of an Execution Context**

Each execution context has two phases and contains:

1. **Creation Phase**:
   * **Variable Object (VO)** / **Activation Object (AO)**: Stores variables, functions, and arguments
   * **Scope Chain**: Contains the current variable object and all its parent variable objects
   * **this value** is determined
2. **Execution Phase**:
   * Code is executed line by line
   * Variables are assigned values
   * Functions are executed

**Call Stack**

The call stack is a mechanism JavaScript uses to keep track of execution contexts. It works as a LIFO (Last-In, First-Out) data structure.

**How the Call Stack Works**

1. When a script runs, the global execution context is pushed onto the stack
2. When a function is called, a new execution context is created and pushed onto the stack
3. When a function returns, its execution context is popped off the stack
4. The stack continues executing until it's empty

function first() {

console.log("Inside first");

second();

console.log("Back to first");

}

function second() {

console.log("Inside second");

third();

console.log("Back to second");

}

function third() {

console.log("Inside third");

}

first();

console.log("Global context");

**Call Stack Flow**:

1. Global execution context is pushed
2. first() is called - pushed to stack
3. second() is called from first() - pushed to stack
4. third() is called from second() - pushed to stack
5. third() completes - popped from stack
6. second() completes - popped from stack
7. first() completes - popped from stack
8. Global context completes - stack is empty

**Stack Overflow**

A stack overflow occurs when the call stack exceeds its maximum size, typically due to infinite recursion:

function infinite() {

infinite(); // Keeps calling itself

}

infinite(); // Causes stack overflow

* JavaScript is single-threaded, meaning it has only one call stack
* The call stack determines the order of execution
* Each function call creates a new execution context
* Understanding execution context and call stack is crucial for debugging and understanding scope, hoisting, and closures

**context.js**

let a = 10; // **global variable / global context**

function add(num1, num2) {

  const result = num1 + num2 + a; function **variable / function context**

  return result;

}

console.log(result); //ReferenceError: result is not defined

const sum = add(10, 20);

console.log(sum);

**Single -Threaded Asynchronous vs Multi-Threaded Synchronous**

Let's break down the concepts of:

* **Single-threaded, asynchronous JavaScript**
* **Multi-threaded, synchronous (and asynchronous) behavior in other languages**  
  ...and how JavaScript handles concurrency and parallelism.

**🧠 JavaScript: Single-Threaded and Asynchronous**

**✅ Single-Threaded**

JavaScript (in the browser and in Node.js) runs on a **single thread** — one operation at a time. That means:

* Only **one line of code executes at a time**.
* The **JavaScript Engine** uses an **event loop** to manage tasks and callbacks.

**✅ Asynchronous (via Event Loop)**

Although it's single-threaded, JS **handles asynchronous tasks** (like API calls, timers, file reads) using:

* **Web APIs** (in browsers)
* **libuv** (in Node.js)

These external APIs run on **separate threads**, and notify JS when tasks are done. JavaScript doesn't block execution waiting for these to finish — it continues executing other code.

**🔄 Example:**

console.log("1");

setTimeout(() => {

console.log("2"); // Runs later

}, 1000);

console.log("3");

**Output:**

1

3

2

This is possible due to the **event loop**, not because JavaScript is multithreaded.

**⚙️ Multi-Threaded, Synchronous (Other Languages)**

Languages like Java, C++, or Python can use **multiple threads** to perform **synchronous or asynchronous** operations.

**🔧 Multi-threaded, Synchronous:**

Thread t1 = new Thread(() -> task1());

Thread t2 = new Thread(() -> task2());

t1.start();

t2.start();

* Both task1 and task2 run in parallel (if CPU has multiple cores).
* JavaScript can't do this natively in the main thread.

**🔁 JavaScript and Multithreading?**

Though JS is single-threaded **by default**, it can achieve multithreaded behavior using:

**✅ Web Workers (in browsers)**

const worker = new Worker("worker.js");

worker.postMessage("Start working...");

**✅ Worker Threads / Child Processes (in Node.js)**

const { Worker } = require("worker\_threads");

new Worker("./worker-task.js");

But these are **separate threads** and communicate via **message passing** — not shared memory.

**TL;DR Table:**

| **Feature** | **JavaScript (Default)** | **Multi-threaded Languages** |
| --- | --- | --- |
| Threads | Single | Multiple |
| Async Support | Event Loop, Callbacks, Promises | Threads, async/await, Futures |
| Blocking I/O | Non-blocking (async) | Blocking unless async is used |
| True Parallelism | ❌ (without workers) | ✅ Yes |
| Use of Multiple Cores | ❌ (by default) | ✅ |

Let me know if you want visuals (like an event loop diagram) or a deeper dive into Web Workers or Node.js threads!

**35-4 Single -Threaded, Asynchronous vs Multi-Threaded, Synchronous**

console.log(1);

console.log(2);

console.log(3);

doSomething();

console.log(5);

console.log(6);

function doSomething() {

console.log(4);

}

// 1

// 2

// 3

// 4

// 5

// 6

**console.log("next part");**

console.log(7);

console.log(8);

fetch("https://jsonplaceholder.typicode.com/todos/1")

.then((response) => response.json())

.then((json) => console.log(json));

console.log(10);

console.log(11);

console.log(12);

// 7

// 8

// 10

// 11

// 12

// {userId: 1, id: 1, title: 'delectus aut autem', completed: false}

**// nb: The fetch API is asynchronous, so the code continues to run while waiting for the response. The console.log statements after the fetch call will execute immediately, while the response will be logged once it is received. This is an example of how JavaScript handles asynchronous operations using callbacks and promises.**

**console.log("next part 2");**

console.log(13);

console.log(14);

console.log(15);

// setTimeout(lazyLogged, 5000);

setTimeout(() => {

lazyLogged();

}, 5000);

console.log(16);

console.log(17);

function lazyLogged() {

console.log("lazy logged");

}

// 13

// 14

// 15

// 16

// 17

// { userId: 1, id: 1, title: 'delectus aut autem', completed: false }

// lazy logged

**// nb: The setTimeout is asynchronous. The setTimeout function is used to delay the execution of the lazyLogged function by 5 seconds. The code continues to run while waiting for the timeout, so the console.log statements after the setTimeout call will execute immediately. Once the timeout is reached, the lazyLogged function will be called and "lazy logged" will be printed to the console.**

html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Document</title>

**<script src="async.js"></script>**

</head>

<body>

</body>

</html>

**35-5 Promise, Async/Await**

const **myLoader** = () => {

return new **Promise**((resolve, reject) => {

const value = Math.random();

if (value < 0.4) {

resolve(value);

} else {

reject(value);

}

});

};

**myLoader()**

.then((data) => console.log("resolved data", data))

.catch((err) => console.log("rejected with value", err));

**fetch("")**

.then((res) => res.json())

.then((data) => console.log(data))

.catch((err) => console.error(err));

**// normal function**

**async** function dataLoader() {

const res = **await** fetch("");

const data = **await** res.json();

console.log(data);

}

dataLoader();

**// arrow function**

const loadData = **async** () => {

const res = **await** fetch("https://jsonplaceholder.typicode.com/photos");

const data = **await** res.json();

console.log(data);

};

**loadData();**

html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Async Await</title>

**<script src="async.js"></script>**

</head>

<body>

</body>

</html>

Here's a **beginner-friendly explanation** of Promise, async, and await in JavaScript:

**🧾 1. Promise**

A Promise is an object that represents the eventual **completion (or failure)** of an asynchronous operation and its resulting value.

**✅ Example:**

const promise = new Promise((resolve, reject) => {

setTimeout(() => {

resolve("Data loaded");

}, 2000);

});

promise.then((data) => {

console.log(data); // "Data loaded"

});

* resolve() means success.
* reject() means failure.
* .then() handles the resolved data.
* .catch() handles errors.

**🧾 2. async / await**

async and await are **syntactic sugar** for working with Promises. They make your code look more like regular synchronous code (easier to read!).

**✅ Basic Syntax:**

const loadData = async () => {

try {

const res = await fetch("https://jsonplaceholder.typicode.com/posts");

const data = await res.json();

console.log(data);

} catch (error) {

console.error("Error:", error);

}

};

loadData();

**Key Points:**

* You add async before a function to use await inside.
* await pauses the function until the promise resolves.
* Wrap with try/catch to handle errors.

**🔄 Comparison**

**With Promise:**

fetch("https://jsonplaceholder.typicode.com/posts")

.then((res) => res.json())

.then((data) => console.log(data))

.catch((err) => console.error(err));

**With async/await:**

const getPosts = async () => {

try {

const res = await fetch("https://jsonplaceholder.typicode.com/posts");

const data = await res.json();

console.log(data);

} catch (err) {

console.error(err);

}

};

getPosts();

**35-6 settimeout, setInterval and clearInterval**

**interval.html**

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Interval</title>

    <script src="**2.interval.js**"></script>

</head>

<body>

</body>

</html>

**interval.js**

console.log("setTimeout");

console.log(4);

**setTimeout(() => {**

**console.log(5);**

**}, 4000);**

console.log(6);

console.log("setInterval");

**setInterval(() => {**

**console.log("mahade");**

**}, 2000);**

console.log(11);

**let num = 0;**

**const clockId = setInterval(() => {**

**num++;**

**console.log(num);**

**if (num == 5) {**

**clearInterval(clockId);**

**}**

**}, 2000);**

Here’s a **concise summary** **of setTimeout, clearTimeout, setInterval, clearInterval, and clockId in JavaScript,** with proper examples:

**🕒 1. setTimeout()**

Runs a function **once** after a specified delay (in milliseconds).

**✅ Syntax:**

let timeoutId = setTimeout(function, delay);

**🔍 Example:**

let timeoutId = setTimeout(() => {

console.log("Hello after 3 seconds");

}, 3000);

**❌ 2. clearTimeout()**

Cancels a timeout set by setTimeout() using the timeoutId.

**🔍 Example:**

clearTimeout(timeoutId); // stops the timeout before it runs

**🔁 3. setInterval()**

Repeats a function **every** given time interval.

**✅ Syntax:**

let intervalId = setInterval(function, delay);

**🔍 Example:**

let intervalId = setInterval(() => {

console.log("Repeats every 2 seconds");

}, 2000);

**🛑 4. clearInterval()**

Stops a repeated action set with setInterval() using the intervalId.

**🔍 Example:**

clearInterval(intervalId); // stops the interval

**🆔 5. clockId (or ID variables like timeoutId / intervalId)**

These are **identifiers** returned by setTimeout() or setInterval() and are used to clear/cancel them.

* timeoutId → used with clearTimeout()
* intervalId → used with clearInterval()

**✅ Real-world Example: Start & Stop a Clock**

let clockId;

function startClock() {

clockId = setInterval(() => {

console.log(new Date().toLocaleTimeString());

}, 1000);

}

function stopClock() {

clearInterval(clockId);

}

startClock(); // starts showing time every second

setTimeout(stopClock, 5000); // stops after 5 seconds

Let me know if you want a live HTML example to test it out.

Sure! Here's an example using **conditional clearInterval() and clearTimeout()** in JavaScript, which stops based on certain logic.

**✅ Example: Countdown Timer with Conditional Stop**

let count = 10;

// Interval: counts down every second

let intervalId = setInterval(() => {

console.log("Countdown:", count);

count--;

// ❌ Conditionally stop the interval

if (count === 0) {

console.log("Time's up!");

clearInterval(intervalId);

}

}, 1000);

// Timeout: cancel the countdown early (e.g., after 5 seconds)

let timeoutId = setTimeout(() => {

console.log("Stopped early!");

clearInterval(intervalId); // ❌ stops the countdown

}, 5000);

// Example condition: clear the timeout if count drops below 8 before 5 seconds

let checkCountId = setInterval(() => {

if (count < 8) {

clearTimeout(timeoutId); // ✅ cancel the early stop

clearInterval(checkCountId); // stop checking

console.log("Early stop cancelled, countdown continues...");

}

}, 500);

**🔍 Breakdown:**

* setInterval runs every second to count down.
* setTimeout is set to stop the interval after 5 seconds.
* A second interval checks the value of count, and **conditionally cancels the timeout** if a rule is met (count < 8), keeping the countdown going.

This shows **conditional use** of both clearInterval() and clearTimeout() based on runtime logic. Let me know if you want a live HTML version with a button and display!

**// 35-7 JavaScript event loop and concurrency**

function a() {

  console.log("a");

**b();**

  console.log("aa");

}

function b() {

  console.log("b");

**c();**

  console.log("bb");

}

function c() {

  console.log("c");

  console.log("cc");

}

function x() {

  console.log("x");

**y();**

  console.log("xx");

}

function y() {

  console.log("y");

**z();**

  console.log("yy");

}

function z() {

  console.log("z");

  console.log("zz");

}

setTimeout(() => {

  console.log("Hello setTimeOut");

}, 1000);

fetch("https://jsonplaceholder.typicode.com/todos/1")

  .then((response) => response.json())

  .then((json) => console.log(json));

**a();**

**x();**

Great! Let's walk through this code step by step and **understand how JavaScript executes it** using the **call stack**, **Web APIs**, **task queue**, **microtask queue**, and the **event loop**.

**✅ Synchronous Code First (Call Stack)**

These functions are **synchronous**, so they’re executed immediately in a top-down manner.

a(); // called immediately

x(); // called immediately

Here's what happens when a() and x() run:

**a() calls b(), which calls c():**

a()

console.log("a")

b()

console.log("b")

c()

console.log("c")

console.log("cc")

console.log("bb")

console.log("aa")

**x() calls y(), which calls z():**

x()

console.log("x")

y()

console.log("y")

z()

console.log("z")

console.log("zz")

console.log("yy")

console.log("xx")

**📦 Web APIs (Async Code)**

setTimeout(..., 1000); // goes to Web API timer, then after 1000ms -> task queue

fetch(...) // goes to Web API, response -> microtask queue (promise)

**📋 Execution Order**

**1. Call Stack**

The synchronous part runs first:

a

b

c

cc

bb

aa

x

y

z

zz

yy

xx

**2. Microtask Queue (Promises)**

The fetch(...) Promise resolves and logs the fetched JSON **before** the setTimeout.

{ userId: 1, id: 1, title: "...", completed: false }

(The exact JSON depends on the API, but it'll be something like that.)

**3. Task Queue (setTimeout)**

After **1000ms**, the setTimeout callback runs:

Hello setTimeOut

**✅ Final Output (Likely Order)**

a

b

c

cc

bb

aa

x

y

z

zz

yy

xx

{ userId: 1, id: 1, title: "...", completed: false }<-- Microtask from fetch

Hello setTimeOut <-- Task queue after 1s

**🧠 Summary of Concepts Used**

* **Call Stack** → Runs synchronous functions (a(), b(), c(), etc.)
* **Web APIs** → setTimeout, fetch
* **Microtask Queue** → .then() of Promises (runs before task queue)
* **Task Queue** → Timer callback (setTimeout)
* **Event Loop** → Moves microtasks and tasks into the call stack when it's free

Sure! Here's a **brief summary**:

**🔁 JavaScript Event Loop Flow (Summary)**

1. **Synchronous functions** like a() and x() run first, filling the **call stack** and executing top-down.
2. **setTimeout()** and **fetch()** are sent to **Web APIs** to handle in the background.
3. After sync code finishes, the **fetch promise** resolves → its .then() callback goes to the **microtask queue**, and is executed **immediately next**.
4. Once 1000ms is up, the setTimeout callback moves to the **task queue**, and runs **after microtasks**.

**🧾 Final Output Order:**

a b c cc bb aa x y z zz yy xx

(fetch result)

Hello setTimeOut

Let me know if you want this as a handout or slide!

**JavaScript event loop and concurrency**

**function a()** {

  console.log("a");

  b();

  console.log("aa");

}

**function b()** {

  console.log("b");

  c();

  console.log("bb");

}

**🧠 Is it synchronous or asynchronous?**

✅ **It is synchronous.**

**🔍 Why?**

Because **each function is called one after another**, and each **must finish before the next line can run**.

Let’s trace what happens step by step:

1. a() is called.
2. Inside a(), console.log("a") → prints **a**.
3. Then it calls b():
   * Inside b(), console.log("b") → prints **b**.
   * Then it calls c():
     + Inside c(), console.log("c") → prints **c**.
   * After c(), console.log("bb") → prints **bb**.
4. Back in a(), after b() is done, console.log("aa") → prints **aa**.

**🧾 Output:**

a

b

c

bb

aa

No timers, no promises, no async/await — everything runs top to bottom, in order, on the **main thread**.

**⚡ Key rule:**

JavaScript is **synchronous by default**, and **asynchronous only with Web APIs, Promises, async/await**, etc.

If you want to **make it asynchronous**, you could modify c() like this:

function c() {

setTimeout(() => console.log("c"), 0);

}

Then the output becomes:

a

b

bb

aa

c

Let me know if you want a visual of the call stack or how it works under the hood!

Great! Let’s connect **asynchronous operations** and the **waiting queue** specifically in **JavaScript**—especially in the browser or Node.js environment.

**🧠 Basic Concept**

JavaScript is **single-threaded**, meaning it can execute only one piece of code at a time. But thanks to **asynchronous programming**, it can **handle multiple operations efficiently** without blocking.

**🔄 What Happens with Async Code?**

Here’s how it works:

**1. You call an async function**

Example:

setTimeout(() => {

console.log("Hello after 2 seconds");

}, 2000);

**2. The task is sent to the Web APIs (in the browser)**

* In the case of setTimeout, fetch, or DOM events, the browser handles it outside the main JS thread.
* It **does not block** the rest of the JS code.

**3. Once it's done, the callback goes to the waiting queue (event queue or callback queue)**

**4. Event Loop checks if the call stack is empty**

* If yes, it **moves the task from the queue to the call stack** and runs it.
* If no, it waits.

**🕳️ Visual Example**

**console.log("1");**

setTimeout(() => {

console.log("2");

}, 0);

console.log("3");

🔍 Output will be:

1

3

2

Even though setTimeout is 0ms, the callback goes to the **waiting queue**, and the **event loop** waits until the stack is empty.

**🧪 Real-World Use Case**

fetch('https://api.example.com/data')

.then(response => response.json())

.then(data => console.log(data));

console.log("Fetching data...");

Here:

* The fetch request is asynchronous.
* The callback to handle data goes to the **waiting queue** after the request completes.
* Meanwhile, "Fetching data..." prints immediately.

**🧩 Summary of Relation:**

| **Asynchronous Operation** | **Waiting Queue Role** |
| --- | --- |
| Does not block JS execution | Stores callbacks of finished async tasks |
| Uses Web APIs, Promises, or async/await | Event Loop pulls from this queue when ready |
| Improves efficiency | Keeps JS responsive |

Absolutely! Let’s break down **Web APIs** in the **browser context**—they're a crucial part of how JavaScript interacts with the browser and enables asynchronous behavior.

**🌐 What are Web APIs?**

**Web APIs** are **browser-provided functionalities** that JavaScript can use to perform tasks like:

* Delaying execution
* Making HTTP requests
* Handling DOM events
* Storing data locally
* Managing media, animations, etc.

JavaScript **doesn’t do these tasks directly**—instead, it **asks the browser** (via Web APIs) to handle them in the background.

**🧩 Common Web APIs (Browser)**

| **Category** | **Examples** | **Description** |
| --- | --- | --- |
| **Timers** | setTimeout(), setInterval() | Run code after a delay or repeatedly |
| **Networking** | fetch(), XMLHttpRequest | Make HTTP requests |
| **DOM** | document.querySelector(), addEventListener() | Access and manipulate page elements |
| **Storage** | localStorage, sessionStorage | Store key-value data locally |
| **Geolocation** | navigator.geolocation.getCurrentPosition() | Get the user's location |
| **Media APIs** | getUserMedia() | Access camera/microphone |
| **Canvas/WebGL** | canvas.getContext(), WebGLRenderingContext | Render 2D/3D graphics |
| **Notifications** | Notification API | Show system notifications |
| **Web Workers** | new Worker() | Run JS in a background thread |

**🔁 How Web APIs Work with Async**

Here’s a simplified flow:

console.log("Start");

setTimeout(() => {

console.log("From setTimeout");

}, 0);

console.log("End");

**🧠 What’s happening:**

1. JS engine logs **"Start"**.
2. setTimeout is handed to the **Timer Web API**.
3. JS engine logs **"End"**.
4. After 0ms, **callback goes to the event queue**.
5. Once the call stack is clear, the **event loop** pushes it back into the call stack, and it runs.

**Output:**

Start

End

From setTimeout

**🧪 When You Use Web APIs**

* **Click events**:

document.getElementById("btn").addEventListener("click", () => {

console.log("Button clicked");

});

* **Fetching data**:

fetch("https://api.example.com")

.then(response => response.json())

.then(data => console.log(data));

These are **not native to JavaScript**, but are part of the **browser’s environment**.

Ah, brilliant question! You're touching on a key concept that can seem confusing at first:

❓ **If JavaScript is synchronous and runs functions in order, why is it said to use LIFO (Last In, First Out)?**

Let’s break it down clearly. The **LIFO** concept applies to the **JavaScript Call Stack**, not the order in which you *call* functions, but the way they are *managed internally* when invoked.

**🧠 Call Stack in JavaScript**

The **Call Stack** is a data structure that follows the **LIFO (Last In, First Out)** principle.

* ✅ When a function is called, it's **pushed onto the stack**.
* ✅ When it finishes, it’s **popped off the stack**.

**🔍 Example to Understand LIFO**

function one() {

console.log("One start");

two();

console.log("One end");

}

function two() {

console.log("Two start");

three();

console.log("Two end");

}

function three() {

console.log("Three");

}

one();

**📦 What happens on the Call Stack?**

1. one() is called ➡️ push `one`

2. inside one(), two() is called ➡️ push `two`

3. inside two(), three() is called ➡️ push `three`

4. three() logs "Three" and returns ➡️ pop `three`

5. two() logs "Two end" and returns ➡️ pop `two`

6. one() logs "One end" and returns ➡️ pop `one`

**🧾 Output:**

One start

Two start

Three

Two end

One end

Even though you **called** them in a "1 ➝ 2 ➝ 3" order, the **last one to go on the stack** (three) is the **first one to finish and get removed** — that’s **LIFO** in action.

**🧩 Summary:**

| **Concept** | **Applies To** | **Behavior** |
| --- | --- | --- |
| Function Execution Order | Your code | Top-to-bottom, in sequence |
| Call Stack | Behind the scenes | LIFO (Last In, First Out) |

So LIFO isn’t about the **order you write or call functions** — it’s about how JavaScript **internally handles nested or chained function calls**.

Let me know if you want a visual diagram of this call stack in action — I can generate one!

