📜 Episode 1: What is JavaScript? (Full Notes)

🌍 Why JavaScript Matters

- JavaScript enables interactivity on the web (e.g., button clicks, form submissions, modals).
- Controls browser behavior and dynamically updates webpage content.
- Essential for modern web development:
  - Frontend (UI/UX interactivity)
  - Backend (Node.js)
  - Mobile apps, desktop apps, games, and more.
- Widely used across tech ecosystems:
  - Websites, robots, smart TVs, IoT devices.

🕰️ History of JavaScript

- Created in **1995** by **Brendan Eich**.
- Originally developed in **10 days**.
- First named **Mocha**, then **LiveScript**, and finally **JavaScript**.
- The name JavaScript was chosen to leverage the popularity of **Java** at the time — but the two languages are completely unrelated.

> ***Java ≠ JavaScript***
> ***Similar name, but entirely different languages in terms of syntax, design, and use.***

💡 What Kind of Language is JavaScript?

Scripting Language

JavaScript is classified as a **scripting language**. It was initially created to automate small tasks in the browser — like form validation or creating interactive UI components.

Interpreted and Just-In-Time (JIT) Compiled Language

JavaScript is both an **interpreted** and **Just-In-Time (JIT) compiled** language. This means:

- Code is executed line-by-line by the browser.
- The JavaScript engine optimizes performance by compiling parts of the code during execution, just before they are needed.
- No manual compilation step is required beforehand, making development faster and more dynamic.
- Combines the flexibility of interpretation with the speed of compilation.

High-Level Language

Being a **high-level language**, it abstracts away many low-level details such as memory management and hardware-level operations.

Multi-Paradigm

JavaScript supports different programming styles:

- **Procedural**
- **Object-Oriented**
- **Functional**

This flexibility makes it highly adaptable and powerful for solving a wide variety of problems.

🤔 ECMAScript vs JavaScript

ECMAScript is the standard or specification that defines how JavaScript should work. It's like the rulebook that all JavaScript engines must follow.

JavaScript is the language that follows the ECMAScript specification.

ECMAScript Evolution

The major turning point in JavaScript's evolution was **ES6 (ECMAScript 2015)**. It introduced many modern features such as:

- `let`, `const`
- Arrow functions (`=>`)
- Template literals
- Default parameters
- Destructuring
- Spread and Rest operators
- Classes
- Modules
- Promises

Since then, ECMAScript has been evolving yearly, with continuous improvements and new capabilities.

⚙️ Where Does JavaScript Run?

JavaScript runs inside web browsers. Every modern browser has a built-in **JavaScript Engine** that understands and executes JavaScript code.

Examples of JavaScript Engines:

- **V8** – used in Google Chrome and Node.js
- **SpiderMonkey** – used in Firefox
- **JavaScriptCore** – used in Safari
- **Chakra** – used in older versions of Microsoft Edge

When JavaScript code runs in the browser:

1. The engine parses the code
2. It converts the code into machine-level instructions
3. The engine then executes those instructions in the correct order

This process is seamless to the user. JavaScript code simply runs inside the browser whenever it is loaded or triggered.

📄 How JavaScript is Used in Webpages

JavaScript code can be written directly inside an HTML file using the `<script>` tag. This tag is placed either in the `<head>` or at the end of the `<body>` section.

Example:

```
<script>
    console.log("Hello from JavaScript!");
```

```
</script>
```

When the browser encounters the `<script>` tag, it executes the JavaScript code inside it.

Alternatively, JavaScript code can be written in a separate `.js` file and linked to the HTML using:

```
<script src="script.js"></script>
```

This helps keep code modular and clean.

🔄 JavaScript is Single-Threaded

JavaScript is a **single-threaded** language, meaning it can only execute **one command at a time** in a single thread of execution.

This might sound like a limitation, but JavaScript handles asynchronous tasks (e.g., network requests, timers, etc.) using mechanisms like:

- **Callback Queue**
- **Event Loop**
- **Web APIs**
- **Microtask Queue**

These concepts enable JavaScript to manage multiple operations efficiently despite being single-threaded.

💻 Simple Example of JavaScript in Action

A very basic example of JavaScript usage is using the `console.log()` statement to print output in the browser's developer console:

```
console.log("Namaste JavaScript 🙏");
```

🔚 Summary

- JavaScript is the **backbone of modern web development**
- Originally created in 1995 in just 10 days
- Runs in browsers via **JavaScript Engines**
- Follows the **ECMAScript** standard
- **Single-threaded**, but manages asynchronous tasks efficiently
- Used in almost everything: web, mobile, desktop, server, IoT, AI, and more
- Completely **different from Java**, despite the name
- Runs seamlessly inside HTML using `<script>` tags
- Evolved rapidly with major upgrades starting from **ES6 (2015)**

🧱 Episode 2: How JavaScript works & Execution Context?

- **Introduction to Execution Context:****

  Every piece of JavaScript code runs inside an “execution context”—think of it as a container that holds everything needed to run your script. Without this container, JavaScript wouldn’t know where to store variables

or how to step through your code.

- **Memory Component (Variable Environment):**

  Inside the execution context is a memory area where all variables and function declarations live as key–value pairs. For example, when you write `let a = 10` , under the hood JavaScript stores an entry like `{ a: 10 }` in this environment. Function declarations are treated similarly, allowing the engine to look them up quickly during execution.

- **Code Component (Thread of Execution):**

  Alongside memory, there's a separate code area responsible for actually running your script—one line at a time, in order. This "thread of execution" reads each statement, executes it, then moves on to the next, never skipping or rearranging unless explicitly told to do so (e.g., via callbacks or async constructs in later episodes).

- **Synchronous Single-Threaded Nature:**

  JavaScript can only do one thing at a time and does it in a strict sequence. That means if one line of code takes a while (say a heavy loop), everything else must wait until it finishes. This behavior simplifies reasoning about code order but introduces challenges when dealing with slow operations like network requests or file I/O.

- **Why Jargon Matters:**

  Terms like "execution context, " "variable environment, " and "thread of execution" might sound intimidating, but they map directly to how the engine manages your code. Understanding these will make concepts like hoisting, scope, and asynchronous behavior much clearer down the road.

- **Preview of Asynchronous Concepts:**

  Although JavaScript itself runs synchronously, we'll soon explore how features like AJAX (Asynchronous JavaScript and XML) let you perform tasks in the background. Those mechanisms won't break the single-threaded rule; they'll simply defer work until later and then feed results back into the main thread.

- **Recap of Core Components:**

  1. **Execution Context** – the overarching container
  2. **Variable Environment** – where values and function definitions are stored
  3. **Thread of Execution** – the pathway that runs your code step by step

🟫 Episode 2.1: How JavaScript Code is Executed?

🔍 Big Question: What Happens When JavaScript Code Runs?

When a JavaScript file is loaded into the browser, something happens internally that makes the code come alive. But what exactly happens?

To understand this, it's important to explore the entire journey — from code being written in a file to the output being shown in the console or browser. This is where the concept of the **JavaScript Engine** and **Execution Context** comes in.

This episode is a deep dive into **how JavaScript executes code under the hood** — starting with the concept of the **JavaScript Engine**, and how it handles everything inside.

🧠 JavaScript Engine: The Core of Execution

Every browser has a built-in component called a **JavaScript Engine**. This engine is responsible for reading, interpreting, and executing JavaScript code.

For example:

- **Chrome** uses the **V8 Engine**
- **Firefox** uses **SpiderMonkey**
- **Safari** uses **JavaScriptCore**

These engines are written in low-level programming languages like **C++** and are extremely powerful and optimized for performance.

The engine goes through the following high-level phases:

1. **Parsing** – Reading the code line by line and breaking it into tokens.
2. **Compilation/Interpretation** – Turning code into machine-level or intermediate code.
3. **Execution** – Actually running the code, line by line.

But there's more to it. JavaScript doesn't just run everything blindly — it creates something called an **Execution Context** first.

📄 Execution Context – The Stage for Code

An **Execution Context** is an environment in which JavaScript code is evaluated and executed.

Whenever any JavaScript code runs — whether it's the entire script or just a function — an execution context is created.

There are two main types:

1. **Global Execution Context (GEC)**
2. **Function Execution Context (FEC)**

🌐 Global Execution Context (GEC)

When the JavaScript file is first loaded, the engine automatically creates the **Global Execution Context**.

This is the **default context** in which the entire code initially runs. There's only **one** GEC in any program, and it is created only **once**.

The GEC does two main things:

1. Creates a `this` keyword binding

- In the global context, `this` refers to the **global object**.
- In a browser, the global object is `window`.

So inside GEC:

```
console.log(this); // window
```

2. Sets up memory for all variables and functions

But this doesn't happen in a single step — it happens in **two distinct phases**.

🔄 Two Phases of Code Execution

Whenever JavaScript code runs, execution context is created in **two phases**:

🏗️ 1. Memory Creation Phase (a.k.a Creation Phase)

- JavaScript scans the entire code.
- It allocates **memory** to variables and functions.
- During this phase:
  - **Variables** are assigned the value `undefined`
  - **Functions** are stored **as-is** (i.e., their entire definition is stored in memory)

This phase is often referred to as **Hoisting** — where all declarations are "hoisted" to the top in memory.

Example:

```
var x = 10;

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

In the memory creation phase:

- `x` is allocated memory and assigned `undefined`
- `greet` is stored with its full function code

At this point:

```
x→ undefined
greet→
function code
```

▶️ 2. Code Execution Phase

- Now JavaScript starts executing the code line by line.
- Variables are assigned their actual values.
- Functions are executed **only when they are called**.

For the above example:

- When `x = 10;` runs, `x` now becomes `10`.
- `greet()` would execute the function body if called.

🔁 Function Execution Context (FEC)

Whenever a **function is invoked**, a **new execution context** is created specifically for that function.

This function execution context goes through the **same two phases**:

1. Memory Creation Phase
2. Code Execution Phase

And it has its own:

- `this` **binding**
- **Local memory space**

This FEC is then **pushed onto the Call Stack**, and once the function finishes executing, it is **popped out** from the Call Stack.

🧱 Call Stack – Managing Execution Contexts

The **Call Stack** is a stack-based data structure used by the JavaScript engine to manage multiple execution contexts.

Here's how it works:

- When JavaScript starts, it creates the **Global Execution Context** and pushes it onto the stack.
- When a function is called, a new **Function Execution Context** is created and pushed on top.
- When a function returns, its context is popped from the stack.
- Execution returns to the previous context in the stack.

Visualizing it:

Let's say this is the code:

```
function a() {
    console.log("Inside a");
    b();
}

function b() {
    console.log("Inside b");
}

a();
```

Call Stack Timeline:

1. GEC is created → pushed to stack
2. `a()` is called → FEC of `a` is created → pushed to stack
3. Inside `a`, `b()` is called → FEC of `b` is created → pushed to stack
4. `b` finishes → popped
5. `a` finishes → popped
6. Only GEC remains → finally it also gets cleared when program ends

📦 Summary of the Execution Process

1. JavaScript Engine loads the code
2. Global Execution Context (GEC) is created

- `this` is bound to the global object
- Memory is allocated for variables and functions
- Variables are initialized to `undefined`
- Functions are stored in memory **Note**: Execution Context contains:
  - `this` binding (global object in GEC)

- Scope Chain (access to outer variables)
- Memory/Variable Environment (where variables and functions are stored)
- Thread of Execution (the order in which code runs)

3. Memory Phase:

- All variables get `undefined`
- Functions are stored

4. Execution Phase:

- Code runs line by line
- Functions are invoked as needed

5. New Execution Context is created for each function call
6. Call Stack manages the order of function execution
7. Once execution completes, stack is cleared

🔍 Why Understanding Execution Context Matters

- Helps understand **hoisting** (why variables are `undefined` before initialization)
- Explains **scoping** and **function behavior**
- Lays the foundation for deeper topics like:
  - Closures
  - Lexical environment
  - Event loop
  - Asynchronous JavaScript
  - Callbacks and Promises

Without this internal understanding, debugging JavaScript can feel confusing and inconsistent.

🛠️ Try This Example (Step-by-Step Breakdown)

```
var x = 1;

function first() {
    var y = 2;
    console.log(x + y);
    second();
}

function second() {
    var z = 3;
    console.log(x + z);
}

first();
```

Execution Order:

1. GEC created → `x` is assigned `undefined`, `first` and `second` are stored
2. Execution phase:

- `x = 1`
- `first()` is called → FEC created

3. Inside `first`:

- `y = 2`

- `console.log(x + y)` → 1 + 2 → `3`
- `second()` is called → new FEC created

4. Inside `second`:

- `z = 3`
- `console.log(x + z)` → 1 + 3 → `4`

5. `second` finishes → popped
6. `first` finishes → popped
7. GEC finishes

Console Output:

```
3
4
```

🟫 Episode 3: Execution Context – Hoisting in JavaScript

⚙️ Revisiting Execution Context

Whenever JavaScript code is executed, the engine creates an **Execution Context** (either Global or Function-level), and the execution happens in two phases:

1. Memory Creation Phase

- JavaScript scans through the code.
- All **variable declarations** are assigned the value `undefined`.
- All **function declarations** are loaded entirely (i.e., their function body is stored in memory as-is).

2. Code Execution Phase

- Code is executed line-by-line.
- Variables are assigned actual values.
- Functions are executed only when they are invoked.

This setup leads directly to the phenomenon of **Hoisting**.

🔺 What is Hoisting?

Hoisting is JavaScript's behavior of moving **declarations** to the top of their scope during the memory creation phase.

But here's the key nuance:
Only declarations are hoisted — not initializations.

This means:

- **Variable declarations** are hoisted, but initialized as `undefined`.
- **Function declarations** are hoisted with their complete definition.

So when the code starts running, the variables are already present in memory (with `undefined`), and functions can be invoked even before their actual definition appears in the code.

📦 Example: Hoisting in Action

Example 1:

```
console.log(a); // undefined
var a = 10;
```

What actually happens behind the scenes:

During the memory creation phase:

- `a` is allocated memory and assigned `undefined`.

During the code execution phase:

- `console.log(a)` outputs `undefined`
- `a = 10` is executed and `a` now becomes `10`

So the output is:

```
undefined
```

Example 2: Function Hoisting

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

Here, the entire function `greet` is hoisted into memory. So by the time `greet()` is called, it already exists.

So the output is:

```
Hello
```

This works **only** with **function declarations**, not **function expressions** (more on that soon).

❌ Not All Functions Are Hoisted Equally

There's a difference between:

1. Function Declaration

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

- Fully hoisted
- Can be called **before** its definition

2. Function Expression

```
var greet = function() {
    console.log("Hi");
};
```

- `greet` is hoisted as a variable → `undefined`
- The function itself is not hoisted

So calling it before its definition:

```
greet(); // TypeError: greet is not a function
```

Because at that point:

```
greet→ undefined
```

And `undefined()` is not a valid function call.

💡 Deep Dive: Why Does Hoisting Exist?

Hoisting is not a feature for developers — it's just a **byproduct of how the JavaScript engine creates memory**.

Because of how Execution Context is built in two phases, the declarations (variables, functions) are naturally lifted to the top of their scope — even though they're written lower in the code.

It's not that JavaScript is literally moving the lines of code around. It's just that **declarations exist in memory before code runs**.

🤯 What Gets Hoisted and What Doesn't?

Statement Type	Hoisted?	Value Assigned During Memory Phase
`var` variable	Yes	`undefined`
`let` / `const`	Yes	No value assigned (in TDZ)
Function declaration	Yes	Entire function definition
Function expression	Partially	Variable is `undefined`
Arrow function	Partially	Variable is `undefined`

⛔ Temporal Dead Zone (Preview)

Variables declared with `let` and `const` are also hoisted, **but they are not initialized** in the memory phase.

This means if you try to access them before their declaration, you'll get a **ReferenceError** — not `undefined` .

This concept is known as the **Temporal Dead Zone (TDZ)** and will be explored in detail later.

🧪 Practice Example: Predict the Output

```
console.log(one); // ?
one = 1;
var one;
console.log(one); // ?
```

Memory Phase:

- `one` is hoisted and initialized with `undefined`

Execution Phase:

1. `console.log(one)` → `undefined`
2. `one = 1`
3. `console.log(one)` → `1`

Output:

```
undefined
1
```

🔬 Another Common Interview Example

```
getName(); // ?
console.log(x); // ?

var x = 7;

function getName() {
    console.log("Namaste JavaScript");
}
```

Memory Creation Phase:

- `x → undefined`
- `getName → function code`

Execution Phase:

- `getName()` → logs: `Namaste JavaScript`
- `console.log(x)` → logs: `undefined`

Output:

```
Namaste JavaScript
undefined
```

🔚 Final Takeaways

- Hoisting happens during the **Memory Creation Phase**
- **Variables declared with** `var` are hoisted and initialized with `undefined`
- **Function declarations** are hoisted completely (can be invoked before definition)
- **Function expressions** and **arrow functions** are hoisted as variables (value is `undefined`)
- This explains common bugs like: “why is my variable `undefined`?” or “why is this function not a function?”

📓 Episode 4: How functions work in JavaScript 💛 & Variable Environment

🏵️ JavaScript is a Synchronous Single-threaded Language

Before diving into how functions work, let's recall:

- JavaScript runs **one line of code at a time**
- It follows **a top-to-bottom, line-by-line execution**
- It can only do **one task at a time**
- This behavior is due to its **single-threaded** and **synchronous** nature

But functions allow us to **encapsulate logic** and **reuse** it, so JavaScript has an internal mechanism to handle them in the **Execution Context Stack**

🔁 Functions in JavaScript

When we declare a function, it doesn’t execute immediately — it is **stored in memory**. When we **invoke** or **call** the function, JavaScript creates a **new Execution Context** for it.

Let’s take a simple example:

```
function a() {
    var b = 10;
    console.log(b);
}
a();
```

Here’s what happens step-by-step:

1. JavaScript parses the code
2. It stores the function `a` in memory
3. When `a()` is invoked:

- A new **Execution Context** is created for `a`
- It gets pushed to the **Call Stack**
- It goes through two phases: **Memory Creation Phase** and **Code Execution Phase**

🧰 Inside a Function: How Does It Actually Work?

Each time a function is invoked, **a new Execution Context is created**.

Let’s break it down again with this function:

```
function a() {
    var b = 10;
    console.log(b);
```

```
}
a();
```

When `a()` is called:

1. A New Execution Context is Created for Function `a`

- It's like a new environment (bubble) is created.
- It contains its own **Variable Environment**, **Lexical Environment**, and `this` reference.

2. Memory Creation Phase (inside the function context):

- JavaScript scans the function line-by-line
- All variables inside this function are **allocated memory**
- But variables are not assigned their actual values yet
- So `b` is allocated with value `undefined` initially

```
Memory Environment:
b → undefined
```

3. Code Execution Phase:

- The actual value is assigned to `b`
- So now: `b → 10`
- Then, `console.log(b)` is executed → prints `10`

🧱 Understanding the Call Stack with Functions

JavaScript uses a **Call Stack** to manage function calls.

Think of it like a pile of plates:

- The main execution context is created first (Global EC)
- When you call a function, a new context is pushed onto the stack
- When a function finishes executing, its context is **popped** off the stack

Let's visualize:

```
function a() {
    console.log("inside a");
    b();
}

function b() {
    console.log("inside b");
}
a();
```

Step-by-step execution:

1. **Global Execution Context** is created
2. Global code starts executing, sees `a()` call

3. Creates **Execution Context for** `a` → pushed on stack
4. Inside `a`, it sees `b()` call
5. Creates **Execution Context for** `b` → pushed on stack
6. `b` executes, logs "inside b"
7. `b` completes → context is popped
8. `a` resumes → logs "inside a"
9. `a` completes → context is popped
10. Global context resumes

📦 Variable Environment – What Is It?

The **Variable Environment** is part of the Execution Context and consists of:

- **Memory** space where variables are stored (like `b = 10`)
- **Function arguments** (if any)
- It is what helps the JS engine know what variables exist **inside the function** at that moment

So every function, when invoked, gets its **own variable environment** — totally separate from other contexts.

That's why you can have:

```
function a() {
    var x = 10;
}

function b() {
    var x = 100;
}
```

Both `a` and `b` have their **own versions of `x` ** because each gets a fresh variable environment.

🤯 A Function Inside Another Function

This leads us to a more interesting scenario:

```
function a() {
    var x = 10;

    function b() {
        var y = 20;
        console.log(x, y);
    }
    b();
}
a();
```

What Happens Here?

- `a()` is invoked → new Execution Context for `a`
- Inside `a`, `x = 10` and function `b` is stored in memory
- Then `b()` is called → new Execution Context for `b`

- But wait — `b()` is **inside** `a`, so it has access to `a`'s variable `x`
- `b` has its own variable `y = 20`
- So `console.log(x, y)` logs `10 20`

This is because of **Lexical Scoping** — a function has access to variables from the **environment in which it was defined**, not where it was called

🔍 Recap: What Happens When a Function is Invoked?

Let's summarize the process with exact steps:

1. **A new Execution Context is created**
2. Inside it, JavaScript creates:

- A **Variable Environment** (memory space)
- A **Lexical Environment**
- A `this` keyword reference

3. The Execution Context goes through:

- **Memory Creation Phase** (allocates variables, functions → `undefined`)
- **Code Execution Phase** (assign values, execute code)

4. The Execution Context is **pushed to the Call Stack**
5. When execution finishes, it is **popped off**

🔚 Final Takeaways

- **Each time a function is called**, a brand-new Execution Context is created
- This includes a **Variable Environment**: memory to store that function's variables
- These environments are **completely isolated** — they don't leak into each other
- But inner functions can access variables of outer functions because of **Lexical Scope**

📄 Episode 5: SHORTEST JS Program 🔥 | window & this keyword

🎯 What is the "Shortest JavaScript Program"?

> ❓ *What's the shortest valid JavaScript program?*

☑️ Answer:

```
// An empty file
```

Yes! An empty file is a perfectly valid JavaScript program.

Even if there's **no code written at all**, JavaScript can still run it.
But how?

🔍 JavaScript Still Does Something Internally

Even when the code is empty, the **JavaScript Engine** still performs the following:

1. Creates a **Global Execution Context**
2. Initializes the **Global Object**
3. Binds the `this` keyword
4. Executes the code (which, in this case, is nothing)

So under the hood, things are still happening, and we get some default values.

🌐 What is the Global Object?

Every JavaScript environment (like browsers, Node.js, etc.) provides a **global object** — a central object that holds globally accessible functions, variables, and properties.

In Browsers:

The global object is called:

```
window;
```

So in the browser, the following is true:

```
console.log(window); // outputs the global window object
```

🧠 What is `this` in the Global Context?

Another big point:
When JavaScript is running in the **global context**, the value of `this` is **equal to the global object**.

So in a browser:

```
console.log(this); // window
console.log(window); // window
console.log(this === window); // true ☑
```

This means:
The keyword `this` , when used **outside any function or object**, refers to the `window` object in browsers.

🔁 Understanding with an Example:

```
var a = 10;
console.log(window.a); // 10
console.log(this.a); // 10
```

Here's what's happening:

- You declared `a` using `var` in the global scope.
- When you use `var` globally, JavaScript **attaches it to the global object**.
- So `window.a` and `this.a` both return `10` .

⚠ This only applies to `var` .
Variables declared with `let` or `const` are **not attached** to the `window` object.

🔍 Testing with `let` and `const`

```
let b = 20;
const c = 30;

console.log(window.b); // undefined ❌
console.log(window.c); // undefined ❌
```

☑ `let` and `const` create variables in the **Script scope**, not on the `window` object.
Only `var` attaches to `window` in the global scope.

🧠 Summary of Concepts Introduced

📦 1. Global Execution Context is always created

Even if the file is empty, JavaScript:

- Creates the **Global Execution Context**
- Sets up the **Global Object** (`window` in browsers)
- Sets up the `this` keyword
- Initializes Memory and Execution phases

🌐 2. The Global Object

- In browsers → it's the `window` object
- In Node.js → it's called `global`

🧠 3. `this` in Global Context

- `this === window` ☑
- `this` refers to the global object in global scope

🔗 4. Global Variables

- `var` creates properties on the `window` object
- `let` and `const` do **not**

💡 Bonus Thought

> ***Even if you write no code at all, JavaScript does a lot of setup behind the scenes.***

🔚 Final Takeaways

- The **shortest JS program is an empty file**, but it still runs!
- Every JavaScript file runs inside an automatically created **Global Execution Context**
- The `window` **object** is the global object in browsers, and it's automatically available
- The `this` **keyword** refers to the `window` object in the global scope
- Only `var` attaches variables to `window` ; `let` and `const` don't

📓 Episode 6: undefined vs not defined in JavaScript 😳

🧠 What Will You Learn in This Episode?

In this, explore two terms that sound similar but mean **very different** things in JavaScript:

- `undefined`

- `not defined`

📦 Let's Begin with `undefined`

What is `undefined` ?

A variable is `undefined` when:

- It has been **declared** (exists in memory),
- But has **not been assigned a value**.

Example:

```
var a;
console.log(a); // undefined
```

Explanation:

- The variable `a` is declared.
- No value is assigned.
- During the **Memory Creation Phase**, JavaScript assigns `undefined` to `a`.
- So when we log it, we get `undefined`.

Internal Working:

When JavaScript executes:

1. It scans the code and **allocates memory** for `a`.
2. It assigns the value `undefined` to `a` by default.

So `undefined` is a **placeholder** value that means:

> ***"I know the variable exists, but no value has been assigned yet."***

Important:

JavaScript does **not throw an error** when accessing a declared variable that is still `undefined`.
It simply logs: `undefined`.

🚫 What is `not defined` ?

A variable is `not defined` when:

- You try to access a variable
- That has **never been declared** anywhere in the code

Example:

```
console.log(b); // ❌ ReferenceError: b is not defined
```

Explanation:

- JavaScript cannot find any memory location for `b`

- It throws a **ReferenceError**
- Execution stops there

This means the variable was **never declared**, so it does not exist in any memory space.

💡 Summary of Differences

Concept	`undefined`	`not defined`
Declared?	☑ Yes	❌ No
Assigned Value?	❌ No (default is `undefined`)	❌ No (doesn't even exist)
Error?	❌ No error, logs `undefined`	☑ Throws `ReferenceError`
Memory?	☑ Memory is allocated	❌ No memory allocation

🧪 Let's Do a Code Walkthrough Together

```
console.log(a); // undefined
var a = 10;
```

Internally:

1. **Memory Creation Phase**:

- `a` is declared and stored in memory
- JavaScript assigns `undefined` as a placeholder

2. **Code Execution Phase**:

- `console.log(a)` logs `undefined`
- Then, `a` is assigned the value `10`

Now contrast that with:

```
console.log(b); // ❌ ReferenceError
```

No memory was ever allocated for `b` .
JavaScript doesn't know what `b` is → it throws a `ReferenceError` .

🤯 Why Does JavaScript Assign `undefined` by Default?

Because of the **Memory Creation Phase** during Execution Context creation.

When JavaScript scans through code:

- It allocates memory for all variables declared with `var`
- But doesn't execute them yet
- So those variables exist in memory with `undefined` as the default value

That's why accessing them before actual assignment won't crash the program — you'll just get `undefined` .

🛠 Advanced Example to Solidify the Concept

```
function test() {
    var x;
    console.log(x); // undefined
    console.log(y); // ❌ ReferenceError
    x = 5;
}
test();
```

- `x` is declared, so memory is allocated → `undefined`
- `y` is never declared → no memory → `ReferenceError`

⚠️ Important Note on `let` and `const`

```
console.log(a); // ❌ ReferenceError: Cannot access 'a' before initialization
let a = 10;
```

Unlike `var`, variables declared with `let` and `const`:

- Are **not initialized** to `undefined` immediately
- They are in a **"temporal dead zone"** from the start of the block until the line where they are declared

So while they are technically in memory, you **can't access them** until after their declaration line.

This is different from being `not defined`, but also doesn't log `undefined`.

🔚 Final Takeaways

`undefined`

- A variable is **declared**, but value is not yet assigned
- JavaScript assigns `undefined` during memory creation
- Safe to access — logs `undefined`

`not defined`

- A variable was **never declared**
- JavaScript doesn't find it in memory
- Accessing it throws a **ReferenceError**

🟫 Episode 7: Scope Chain, Lexical Environment & Let vs Var

This explains how JavaScript **manages access to variables** through **Scope**, **Lexical Environment**, and the **Scope Chain**.

Everything starts from the foundational idea of **where a variable is accessible**, and why JavaScript can or can't "find" a variable in a certain place.

🧿 Scope – The Visibility of Variables

Scope defines **where in the code you can access a variable**.

There are two main types of scope in JavaScript:

1. **Global Scope**
2. **Local (Function) Scope**

1. Global Scope

Variables declared **outside any function** are part of the **Global Scope**. These can be accessed from **anywhere** in the program.

```
var name = "Akshay";

function sayName() {
    console.log(name); // Accessible
}
```

2. Local Scope (Function Scope)

Variables declared **inside a function** are **only accessible within that function**.

```
function test() {
    var a = 10;
    console.log(a); // OK
}
console.log(a); // ReferenceError
```

So, a variable's **scope determines its lifetime and visibility**.

📦 Lexical Environment

Every execution context has a **Lexical Environment** associated with it.

What is a Lexical Environment?

Think of it as a **local memory** where all the variables and functions of a particular context are stored. It also has access to its **outer lexical environment**, which refers to the place where it was **physically written** in the code.

- "Lexical" refers to the **position of code during writing time**, not runtime.
- In JavaScript, **scope is lexical**, not dynamic — meaning it's determined by **where you write your code**, not where it's called from.
- **In simple terms:** Every function in JavaScript remembers where it was physically written, and carries that environment with it. That remembered environment is the Lexical Environment.

Structure of a Lexical Environment:

A Lexical Environment has two parts:

1. **Environment Record** – local memory: stores variables and function declarations
2. **Outer Environment Reference** – a link to the outer lexical environment (where this function was defined)

☑️ Lexical Scope:

> *"Lexical Scope in JavaScript means that the scope of a variable is determined by its position in the source code. A function can access variables from its own scope and from the scopes of its parent functions, based on where it was defined — not where it was called. This is resolved during the compile phase, not at runtime."*

🤔 Example :

```
function outer() {
    let a = 10;

    function inner() {
        console.log(a); // inner can access 'a' due to lexical scope
    }

    inner();
}

outer();
```

🗣️ Then explain:

> *"Here, `inner()` is defined inside `outer()`, so it has lexical access to `outer`'s variables, like `a`. This access exists even if `inner()` is called from somewhere else."*

🚫 Q: "What if `inner()` is called from outside `outer()`?"

> *"That doesn't matter. Lexical scope is based on where the function is defined, not where it is called. If `inner` is defined outside `outer`, it won't have access to `outer`'s variables."*

🧪 Example:

```
function outer() {
    let a = 10;
    inner(); // 🍃 inner is called here, inside outer
}

function inner() {
    console.log(a); // ❌ ReferenceError: a is not defined
}

outer();
```

💥 Output:

```
ReferenceError: a is not defined
```

🤔 Why This Happens:

- Even though `inner()` is **called inside** `outer()`, it was **defined outside**.

- Because it was defined globally, `inner()` doesn't have **lexical access** to `outer()`'s variables.
- Lexical scope is **decided at the time of function declaration**, not invocation.

🗣️ What You Can Say in the Interview:

> *"Even if you call a function inside another function, it can't access the outer function's variables unless it was lexically defined inside it. In the example I just shared, `inner()` was defined globally, so it couldn't access `a` from `outer()`. Lexical scope is about where a function is written, not where it's called."*

🔁 Bonus — Scope Chain:

> *"The scope chain is how JavaScript resolves variables. If a variable isn't found in the current function's scope, JavaScript looks into the outer lexical environment — step by step — until it finds the variable or reaches the global scope."*

🔗 Scope Chain – Accessing Variables Across Scopes

When JavaScript tries to access a variable:

- It **first looks in the local memory** (current lexical environment).
- If it doesn't find it, it goes to the **outer environment** (the parent).
- This continues until it reaches the **global environment**.

This chain of lexical environments linked together is called the **Scope Chain**.

Example:

```
function a() {
    var b = 10;
    c();

    function c() {
        console.log(b); // Will this work?
    }
}
a();
```

Even though `c` is not inside `b`, it can access `b` — **because of lexical scoping**. The function `c` is **physically written inside** function `a`, so it has access to `a`'s variables through the **scope chain**.

🧠 How the Scope Chain Actually Works

Imagine a box inside a box inside a box:

- Each function has its **own local scope (inner box)**
- If a variable is not found inside that box, JavaScript looks into the **outer box**
- And so on — until it reaches the outermost box: **global scope**

If the variable is found at any level, it is used.
If not found even in the global scope, JavaScript throws a **ReferenceError**.

🔬 Deep Scope Example

```
var x = 1;

function outer() {
    var y = 2;

    function inner() {
        var z = 3;
        console.log(x, y, z);
    }

    inner();
}
outer();
```

What Happens Here?

- `x` is global
- `y` is in `outer`
- `z` is in `inner`

When `inner()` is invoked:

- It has access to:
  - its own variable: `z`
  - outer variable: `y`
  - global variable: `x`

So the output is:

```
1 2 3
```

Because of the **scope chain**, JavaScript finds all these variables by walking up through the nested lexical environments.

⚠ Reference Error – When Variable Is Not Found

If a variable is not present in any lexical environment (local → outer → global), JavaScript throws a **ReferenceError**.

```
function greet() {
    console.log(a); // ReferenceError
}
greet();
```

Here, `a` is not declared anywhere — not in `greet()` , not globally — so JavaScript fails to find it and throws:

```
ReferenceError: a is not defined
```

VS let vs var (Introduction)

`var` :

- **Function-scoped** – its scope is the entire function where it is defined.
- Hoisted and initialized as `undefined`.

`let` and `const` :

- **Block-scoped** – their scope is limited to the nearest `{ }` block.
- Hoisted but **not initialized** – they stay in the **Temporal Dead Zone (TDZ)** until declared.

This changes how variables behave inside blocks and nested functions.

Example: let vs var Scoping

```
{
    var a = 10;
    let b = 20;
    const c = 30;
}

console.log(a); // 10
console.log(b); // ReferenceError
console.log(c); // ReferenceError
```

- `var a` is **function-scoped** (or global if outside a function), so it is accessible outside the block
- `let b` and `const c` are **block-scoped** — they only exist inside the `{ }` block
- So accessing them outside throws a ReferenceError

Lexical Scoping and Function Creation

Functions carry their lexical scope with them, **even when passed around or executed outside their original environment**.

```
function outer() {
    var x = 10;
    return function inner() {
        console.log(x);
    };
}

var fn = outer(); // outer() executes, returns inner
fn(); // inner() executes → can still access x
```

Even though `outer()` has finished executing, `inner()` still remembers `x` — because it was **lexically defined inside outer**. This concept is the **foundation of closures**, which will be covered in later.

END Final Takeaways

- JavaScript uses **lexical scoping** — variable access is determined by the location where functions are defined

- Every execution context has a **Lexical Environment** with:
  - Its own local memory (environment record)
  - A reference to its outer lexical environment
- These references form the **Scope Chain**
- If a variable isn't found in the current context, JavaScript moves outward step-by-step
- `var` is function-scoped; `let` and `const` are block-scoped and stay in TDZ until declaration

📓 Episode 8:let & const in JavaScript 🔥 | Temporal Dead Zone

`var` behavior:

- **Function scoped**
- **Hoisted** and initialized with `undefined`
- Can be **re-declared** and **updated**
- Attaches to **global object** (like `window` in browsers)

```
console.log(a); // undefined
var a = 10;
```

⚠️ The Confusing Part with `let` and `const`

Let's try this:

```
console.log(b); // ❌ ReferenceError: Cannot access 'b' before initialization
let b = 20;
```

Even though `b` is hoisted internally, **we can't access it before its declaration**. Why?

Because of something called the **Temporal Dead Zone**.

⏳ What is the Temporal Dead Zone (TDZ)?

> ***TDZ is the time between when a `let` or `const` variable is hoisted and when it is actually initialized.***

During this phase:

- The variable exists in memory
- But accessing it will throw a `ReferenceError`

Visual Timeline:

```
Creation Phase:
---------------------------
'let b' is hoisted -> ☑ memory allocated (no value assigned)
---------------------------
TDZ begins (start of scope) ➡ Ends at the line where 'b' is defined

Execution Phase:
---------------------------
At line: let b = 20; -> ☑ b is initialized
```

Trying to access `b` **before** its declaration line is an error.

🔍 Internal Behavior of `let` and `const`

When a variable is declared with `let` or `const` :

- Memory is allocated during the **Memory Creation Phase**
- But they are **not initialized with undefined**
- They remain **uninitialized**
- You must **explicitly assign** a value before accessing them

```
let x;
console.log(x); // ☑ undefined (after declaration line)
```

But:

```
console.log(x); // ✖ ReferenceError
let x = 10;
```

❄ Difference Between `let` and `const`

Feature	`let`	`const`
Re-assignable	☑ Yes	✖ No
Must initialize at declaration	✖ No	☑ Yes
Hoisted	☑ Yes	☑ Yes
Initialized to `undefined` ?	✖ No (TDZ)	✖ No (TDZ)
Scope	Block Scoped	Block Scoped

`const` needs assignment immediately:

```
const y; // ✖ SyntaxError: Missing initializer
```

```
const y = 10; // ☑ Works
```

📦 Block Scope – Key for `let` & `const`

Unlike `var` , `let` and `const` are **block scoped**.

```
{
    let a = 10;
    const b = 20;
    console.log(a); // 10
```

```
}
console.log(a); // ❌ ReferenceError
```

Block = anything inside `{ }` (like inside functions, `if` , `for` , etc.)

🧨 Example: Temporal Dead Zone in Action

```
function demo() {
    console.log(x); // ❌ ReferenceError
    let x = 5;
}
demo();
```

Even though `x` is hoisted, it's in **TDZ** until the line `let x = 5` .

So accessing it **before the initialization** line causes a `ReferenceError` .

💣 The Trap of Shadowing

```
let a = 100; {
    let a = 10;
    console.log(a); // 10
}
console.log(a); // 100
```

The inner `a` is **shadowing** the outer `a` .
Each `let` exists in its **own block scope**. They are **independent**.

🤯 Common Interview Trick:

```
console.log(a); // ?
let a = 10;
```

💥 Answer: `ReferenceError`

🧠 Reason: `a` is in the TDZ, so JS won't allow access even though it's hoisted

🚨 Extra Gotcha: `typeof` Doesn't Save You Anymore

```
console.log(typeof a); // ❌ ReferenceError with let/const
let a = 10;
```

With `var` , `typeof` returns `"undefined"` .

But with `let` and `const` , it throws an error because the variable is still in the **TDZ** — not safely accessible.

🔁 Another TDZ Example with Nested Scopes

```
let a = 100;

function test() {
    console.log(a); // ❌ ReferenceError (TDZ)
    let a = 200;
}
test();
```

Even though there's a global `a`, the inner `let a` **shadows** it.

During `test()`'s execution, JS creates a new memory for `a` in that local scope, but since it's in **TDZ**, accessing it before declaration results in an error.

🧠 Final Takeaways

Keyword	Hoisted?	Initialized in Memory Phase?	Scope	Can Reassign?	Can Re-declare?
`var`	☑ Yes	☑ Yes, with `undefined`	Function	☑ Yes	☑ Yes
`let`	☑ Yes	❌ No (TDZ)	Block	☑ Yes	❌ No
`const`	☑ Yes	❌ No (TDZ)	Block	❌ No	❌ No

🟫🔥 Episode 9: BLOCK SCOPE & Shadowing in JS

🧠 What Will You Learn in This Episode?

- What exactly is **Block Scope**?
- How do `var`, `let`, and `const` behave inside blocks `{}`?
- What is **Shadowing** in JavaScript?
- Can shadowing cause bugs?
- How does shadowing behave in strict mode?
- What about **illegal shadowing**?

🧩 What is a Block?

> *A block in JavaScript is everything inside a pair of **curly braces `{}` **.*

Examples of blocks:

- **Control structures**: `if`, `for`, `while`, etc.
- **Standalone blocks**: Just `{}` in the global or local scope

Example:

```
{
    // This is a block
}
```

You can write anything inside a block — it's just a set of instructions grouped together.

📦 What is Block Scope?

> ***Block Scope means that variables declared inside a block cannot be accessed from outside that block.***

Important:

- `let` and `const` are **block-scoped**.
- `var` is **NOT block-scoped** (it ignores the block).

Example:

```
{
    var a = 10;
    let b = 20;
    const c = 30;
}
console.log(a); // ☑ 10
console.log(b); // ✗ ReferenceError
console.log(c); // ✗ ReferenceError
```

Explanation:

- `var a` gets hoisted to the function/global scope. It **does not care about the block**.
- `let b` and `const c` are block-scoped — they **exist only inside** the `{}` block.

🚨 Why is `var` NOT Block Scoped?

When you declare `var` inside a block:

- It is still available **outside the block** because it gets hoisted to the nearest **function** or **global** scope.

This can cause **unexpected bugs**, especially in loops or conditionals!

🔍 Real-World Example

Imagine you are writing an `if` condition:

```
if (true) {
    var x = 10;
}
console.log(x); // ☑ 10
```

Even though `x` was declared **inside the if block**, it is still accessible outside.
That's because `var` is NOT block-scoped.

But if you use `let` :

```
if (true) {
    let y = 20;
}
console.log(y); // ✗ ReferenceError
```

☑ `let` respects the block.

Shadowing in JavaScript

> ***Shadowing happens when a variable declared in a local/block scope has the same name as a variable in an outer scope.***

The inner variable **shadows** the outer one.

Example:

```
var a = 100; {
    var a = 10;
    console.log(a); // 10 (inner 'a' shadows outer 'a')
}
console.log(a); // 10 (still 10 because 'var' overrides globally)
```

With `let` or `const` :

```
let b = 100; {
    let b = 20;
    console.log(b); // 20
}
console.log(b); // 100
```

Here:

- Inside the block: `b` is 20
- Outside the block: `b` is still 100

Because `let` is **block-scoped**, the shadowed version disappears outside the block!

Shadowing Behavior for `var` , `let` , and `const`

Behavior	`var`	`let` and `const`
Scope	Function / Global	Block
Shadowing effect	Affects outside variable	Only inside the block
After block access	Outer variable overwritten	Outer variable preserved

Illegal Shadowing

There are **rules** about what you can and cannot shadow!

> ***If you try to declare a `var` variable inside a block where a `let` or `const` variable already exists with the same name — that is illegal shadowing.***

It will throw an error!

Example:

```
let c = 10; {
    var c = 20; // ❌ SyntaxError: Identifier 'c' has already been declared
}
```

Why?

- `let c` is block-scoped
- Trying to declare `var c` conflicts because `var` tries to hoist it into the function/global scope — but it clashes with the existing `let c`.

☑ You **can** shadow `var` with `let` / `const`, but not the other way around.

Example that works:

```
var d = 10; {
    let d = 20; // ☑ OK
    console.log(d); // 20
}
console.log(d); // 10
```

🔥 Shadowing with Different Scopes

If you're inside a function, the behavior still applies.

```
function test() {
    var a = 100; {
        let a = 200; // ☑ New 'a' inside block
        console.log(a); // 200
    }
    console.log(a); // 100
}
test();
```

☑ The outer `a` and block `a` are independent due to `let`.

📏 Summary Table

Concept	`var`	`let` / `const`
Block Scope	❌ No	☑ Yes
Shadowing allowed	☑ Yes (but overrides globally)	☑ Yes (only within block)
Illegal Shadowing	❌ No error if overwriting itself	❌ Error if var shadows let/const

🔥 Final Takeaways

- `let` and `const` respect **block scope** strictly. * `var` **ignores block scope** — behaves according to **function/global** scope.
- Shadowing is allowed carefully:

- **Shadowing** `var` **with** `let/const` is OK.
- **Shadowing** `let/const` **with** `var` is illegal.

- Illegal shadowing will cause **SyntaxErrors**.
- Always use `let` and `const` instead of `var` for **safer scoping** and avoiding hidden bugs.

📓 Episode 10: Closures in JavaScript 🔥

🚀 What Will You Learn?

- What exactly is a **Closure**?
- How and **when** is a Closure created?
- Real-life examples where Closures are used
- Why are Closures considered powerful in JavaScript?
- How JavaScript remembers variables even after a function finishes execution

Let's go **deep into the JavaScript engine** and explore one of its most magical concepts!

🧠 First, Understand the Basic Behavior

Let's start with a simple example:

```
function outer() {
    var a = 10;

    function inner() {
        console.log(a);
    }
    inner();
}
outer();
```

☑ Output: **10**

Nothing surprising here.

- `outer()` is called.
- `outer()` declares `a` and defines `inner()`.
- `inner()` accesses and prints `a` from its parent function.

But now... let's tweak it slightly.

🎩 The Real Magic Begins: Returning a Function

```
function outer() {
    var a = 10;

    function inner() {
        console.log(a);
    }
    return inner;
}
```

```
var innerFunc = outer();
innerFunc();
```

☑ Output: **10**

Wait, **how?**

- `outer()` finishes execution.
- Normally, variables (`a`) should be **gone** (memory should be cleared).
- But `innerFunc()` still **remembers** `a = 10` !

➡ This is because of **Closures**.

🔒 What is a Closure?

> ***A closure is a function that has access to its own scope, the outer function's scope, and the global scope — even after the outer function has returned.***

In Simple Words:

- Even when a function is executed **outside its original lexical scope**, it **remembers** the variables from where it was originally created.

📦 How Closures Work Internally

When `outer()` is called:

- JavaScript creates a **lexical environment** for it.
- `a = 10` is part of that environment.

When `inner()` is returned:

- It **closes over** the variables it needs.
- JS does **not destroy** the `outer` function's variables if they are being used by an inner function.

Instead of clearing memory, JavaScript **preserves** the necessary variables in memory.

☑ **That preserved environment is Closure.**

🔥 Another Example: Multiple Closures

```
function outer() {
    var a = 10;

    function inner1() {
        console.log(a);
    }

    function inner2() {
        console.log(a + 5);
    }
    return {
        inner1,
        inner2,
    };
```

```
}

const {
    inner1,
    inner2
} = outer();
inner1(); // 10
inner2(); // 15
```

Both `inner1` and `inner2` have **independent closures** but they **share** access to `a` from `outer()` .

🏗️ Visualizing the Closure

Imagine a backpack 🎒 attached to every function.

When a function is created, it carries a hidden backpack containing:

- Variables from its **outer scope** that it might need later.

Even when the original environment is **gone**, the backpack (Closure) remains.

📜 Closure Rules

- Functions **lexically** remember where they were created.
- They **carry their lexical environment** even when invoked elsewhere.
- Closures are created **at function definition time**, not at function call time.

💡 Common Real-World Use Cases of Closures

1. Data Hiding and Encapsulation

```
function counter() {
    let count = 0;
    return function() {
        count++;
        console.log(count);
    };
}

const increment = counter();
increment(); // 1
increment(); // 2
increment(); // 3
```

- `count` is **private** — outside code **can't access** it directly.
- Only the returned function can modify it.

☑ **Closures are used for creating private variables!**

2. Event Listeners

Closures are heavily used in event handlers:

```
function attachEventListener() {
    let count = 0;
    document.getElementById("clickMe").addEventListener("click", function() {
        console.log("Button clicked", ++count);
    });
}
attachEventListener();
```

Even after `attachEventListener` execution is done, the click handler **remembers** `count` .

☑ Each click increments the preserved `count` variable.

3. SetTimeout Problems and Closures

```
function x() {
    var i = 1;
    setTimeout(function() {
        console.log(i);
    }, 1000);
    console.log("hi");
}
x();
```

☑ Output:

```
hi
1
```

Even after 1 second delay, the callback **remembers** `i` .

Closures preserve variables even across **async operations** like `setTimeout` .

🔥 Closure and Loops – Common Interview Trap

```
for (var i = 1; i <= 5; i++) {
    setTimeout(function() {
        console.log(i);
    }, i * 1000);
}
```

What will it print?

✵ Output:

```
6
6
```

```
6
6
6
```

Why?

Because `var` is **function scoped**, not block scoped — every callback shares the same `i` (which becomes 6 after loop ends).

☑ To fix it:

Using `let` :

```
for (let i = 1; i <= 5; i++) {
    setTimeout(function() {
        console.log(i);
    }, i * 1000);
}
```

Now output:

```
1
2
3
4
5
```

Or using Closures manually:

```
for (var i = 1; i <= 5; i++) {
    (function(i) {
        setTimeout(function() {
            console.log(i);
        }, i * 1000);
    })(i);
}
```

Passing `i` into an **Immediately Invoked Function Expression (IIFE)** creates a **separate closure** for each iteration.

🧠 Important Interview FAQs on Closures

Question	Short Answer
When are Closures created?	At **function creation** time
Do Closures retain references to outer variables?	☑ Yes, not copies

Question	Short Answer
Can Closures prevent garbage collection?	☑ Yes, if references persist
Are Closures expensive?	Sometimes, can lead to memory leaks if not handled carefully

Episode 11: 🚀 setTimeout + Closures Interview Question 🔥

👋 Introduction

This session focuses on a **very popular interview question** involving `setTimeout` and **Closures**.

🎭 The Interview Question

```
function x() {
    for (var i = 1; i <= 5; i++) {
        setTimeout(function() {
            console.log(i);
        }, i * 1000);
    }
    console.log("Hello JavaScript");
}
x();
```

🎾 Step-by-Step Analysis

- A function `x` is defined.
- Inside `x`, there is a `for` loop where:
  - `i` starts from 1, goes up to 5.
  - For each value of `i`, a `setTimeout` is scheduled.
  - Delay is `i * 1000` milliseconds (i.e., 1s, 2s, 3s, 4s, 5s).
- Immediately after the loop, `Hello JavaScript` is logged.

Now, **what will be the output?**

🤔 Understanding Output Carefully

Immediately:

- The console logs: `Hello JavaScript`

After delays:

- After 1 second, 2 seconds, ..., 5 seconds, what will be printed?

☑ It prints:

```
6
6
6
```

```
6
6
```

Not `1` , `2` , `3` , `4` , `5` as expected.

❓ Why does this happen?

Let's break it down:

- `var` is **function-scoped**.
- When `setTimeout` runs its callback function after delay, the entire loop is already completed.
- By the time callbacks execute, the loop has finished and `i` has become **6** (since the last increment after `i = 5`).
- All the functions inside `setTimeout` **reference** the **same** `i` from the outer scope.
- They don't have their **own copy** of `i` .

Thus, every callback prints `6` .

📦 Key Concept: Closures + Event Loop

- When `setTimeout` is called, the callback function is registered and waits in the Web API environment.
- Meanwhile, JavaScript keeps running the synchronous code (the loop and console.log).
- After the specified time, the callback moves to the **callback queue**, waits for the **call stack** to be empty, and then executes.
- During execution, the callback **looks up** the value of `i` — and finds `6` (not the value at the time of setTimeout call).

🔥 How to Fix It?

1️⃣ Solution using IIFE (Immediately Invoked Function Expression)

```
function x() {
    for (var i = 1; i <= 5; i++) {
        (function(i) {
            setTimeout(function() {
                console.log(i);
            }, i * 1000);
        })(i);
    }
}
x();
```

☑ Output now:

```
1
2
3
4
5
```

How does it work?

- The IIFE creates a **new execution context** for each iteration.
- Each IIFE gets its own **copy of** `i` passed as a parameter.
- The closure formed now locks the **correct value of** `i` for each timeout.

IIFE traps the current value of `i` **inside a new scope**, solving the problem.

🤔 What is an IIFE?

- **Immediately Invoked Function Expression**: a function that runs as soon as it is defined.
- Syntax: `(function(params){...})(arguments);`
- Purpose here: create a new local scope to "freeze" the value of `i` during each iteration.

2️⃣ Solution using let

```
function x() {
    for (let i = 1; i <= 5; i++) {
        setTimeout(function() {
            console.log(i);
        }, i * 1000);
    }
}
x();
```

☑ Output:

```
1
2
3
4
5
```

How does it work?

- `let` is **block-scoped**.
- Each iteration of the loop creates a **new binding** of `i`.
- Each `setTimeout` closure gets a separate, correctly captured value of `i`.

Thus, no problem.

💬 Which method is better?

- If you can use ES6+, using `let` is much **cleaner and simpler**.
- If working in older JS environments (without `let`), using **IIFE** is the traditional solution.

🎯 Important Interview Concepts Covered Here:

Concept	Explanation
`var` is function-scoped	`i` remains same across iterations inside same function scope
`let` is block-scoped	Each iteration has its own copy of `i`

Concept	Explanation
Closures	Functions remember their surrounding lexical scope
Event Loop	`setTimeout` callbacks execute after the stack is clear
IIFE	Creates new scopes to capture variable values

Episode 12: CRAZY JS INTERVIEW 🐵 ft. Closures | Namaste 🙏

🕐 What is a Closure in JavaScript?

Closure is a concept in JavaScript where a **function "remembers" its outer environment** (variables, functions) even after the outer function has finished executing.

In simple terms:

- When a function is defined inside another function, it "captures" the variables around it.
- Even when the outer function is gone from the call stack, the inner function still has access to those variables.

➔ Technical Definition

A closure is the combination of a function **and** the lexical environment **within which** that function was declared.

🕐 Can you give examples of a Closure in JS?

Example:

```
function outerFunction() {
    let outerVariable = 10;

    function innerFunction() {
        console.log(outerVariable);
    }
    return innerFunction;
}
const closureFunc = outerFunction();
closureFunc(); // Output: 10
```

➔ Explanation:

- `outerFunction` creates a variable `outerVariable`.
- `innerFunction` uses `outerVariable` inside it.
- When `outerFunction` is executed, it **returns** `innerFunction`.
- Even though `outerFunction` has completed, `closureFunc` (which is `innerFunction`) still **remembers** `outerVariable` because of Closure. **Example 2:**

```
function x() {
    var a = 7;

    function y() {
        console.log(a);
    }
```

```
    return y;
}

var z = x();
console.log(z); // prints the function body
z(); // prints 7
```

➔ Explanation:

- x() is a function that defines a variable a = 7.
- Inside x, there's another function y() that simply logs a.
- x() returns y (the function itself, not executing it).
- var z = x(); — now z holds the returned function y.
- console.log(z); — prints the function code.
- z(); — when we invoke z, it still remembers a = 7 because of closure.

☑ Closure ensures that a doesn't get garbage collected and is still accessible to y(). **Key Concept:** Closures keep variables **alive** even after their original scope would otherwise have been destroyed.

🕐 Use of double parenthesis ()() in JS

➔ Double parenthesis syntax is used for Immediately Invoked Function Expressions (IIFE).

Example:

```
function x() {
    var a = 7;

    function y() {
        console.log(a);
    }
    return y;
}
x()();
```

Explanation:

- `x()` is called, which returns the function `y`.
- Then immediately `()` invokes `y`.
- So `x()()` means: first call `x`, get `y`, then immediately invoke `y`. ☑ This is an example of **immediate execution** pattern.

🕐 Are Let declarations closed over?

Answer:
Yes, `let` and `const` declarations are **also closed over**.

➔ Explanation:

When a closure captures variables, it doesn't care whether the variables were declared using `var`, `let`, or `const`.
All variables, regardless of how they are declared, can be closed over.

However:

- `let` and `const` are **block scoped**, unlike `var` (function scoped).
- This is why in loops (like `for (let i=0; i<5; i++)`), each iteration captures a fresh new `i`.

Example with `let`:

```
function x() {
    let a = 7;

    function y() {
        console.log(a);
    }
    a = 100;
    return y;
}

var z = x();
z(); // prints 100
```

Explanation:

- `let` declarations are **also closed over** by functions just like `var`.
- `a` was modified after it was declared (`a = 100`).
- When `z()` (which is `y`) is executed, it prints the updated value `100`.

☑ Closures capture variables by **reference**, not by value.

🕐 Are function parameters closed over?

Answer:
Yes, **function parameters are closed over** exactly like normal variables.

Explanation:

- When a function is created, the function's parameters are part of its **lexical environment**.
- Hence, if an inner function refers to a parameter, that reference is maintained via closure.

Example:

```
function z() {
    var b = 900;

    function x() {
        var a = 7;

        function y() {
            console.log(a, b);
        }
```

```
        y();
    }
    x();
}
z();
```

Explanation:

- `b` is a **parameter** of function `z`.
- Even though `b` is passed as an argument (`100`), it is available inside `y` due to closure.
- `y` prints both `a` (7) and `b` (100).

☑ **Function parameters are part of the closure** like normal variables.

🕐 Relation of Scope Chain and Closures

Scope Chain is the **hierarchical structure** JavaScript uses to resolve variables.

Closure is deeply connected to Scope Chain:

- When a function is executed, it **looks outward** through its Scope Chain for any variables it needs.
- The Scope Chain is **created lexically** — meaning based on where functions are physically written in code, **not where they are called from**.

Thus:

- A closure **captures** the Scope Chain at the moment of function creation.
- Even after outer functions are removed from the stack, the Scope Chain is kept alive via closure.

Example:

```
function z() {
    var b = 900;

    function x() {
        var a = 7;

        function y() {
            console.log(a, b);
        }
        y();
    }
    x();
}
z();
```

Explanation:

- `y` needs `a` and `b`.
- It first checks its own scope for variables, then goes outward if not found (scope chain).
- `y` has closure over `a` and `b` due to lexical scoping.

☑ Closures rely on scope chain to access variables.

🕐 Conflicting name Global variables in JS

When there is a **conflict** between local variables and global variables having the **same name**, **JavaScript gives preference to the innermost scope**. **Example:**

```
var a = 100;

function z() {
    var a = 200;

    function x() {
        var a = 7;

        function y() {
            console.log(a);
        }
        y();
    }
    x();
}
z();
```

Explanation:

- There are multiple `a` variables at different levels: global (100), `z` level (200), `x` level (7).
- `y` finds the nearest `a`, which is `7`.

☑ Always the closest scope wins in variable lookup.

🕐 Advantages of Closure

➔ Major Advantages:

1. **Data Privacy / Encapsulation**
2. **Maintaining State between function calls**
3. **Currying and Partial Applications**
4. **Functional Programming Techniques**
5. **Memory Efficiency (with proper use)**

🕐 Data Hiding & Encapsulation in JavaScript?

Data Hiding:
Keeping variables **private**, not accessible from outside code.

Encapsulation:
Wrapping data (variables) and methods (functions) together, and restricting access to the internal data.

In JavaScript, Closures help achieve Data Hiding by allowing access to variables only through **controlled functions**.

🕐 Example of Data Privacy using Closures

Example:

```
function Counter() {
    let count = 0;

    return {
        increment: function() {
            count++;
            console.log(count);
        },
        decrement: function() {
            count--;
            console.log(count);
        },
    };
}

const counter1 = Counter();
counter1.increment(); // 1
counter1.increment(); // 2
counter1.decrement(); // 1
```

Here:

- `count` is **private**.
- It cannot be accessed or modified directly outside `Counter()`.
- Only `increment` and `decrement` can change it.

☑ This achieves **Data Privacy** and **Encapsulation**.

🕐 Function Constructor in JavaScript

Function Constructor is another way to create functions dynamically:

```
var sum = new Function("a", "b", "return a+b");
console.log(sum(2, 3)); // 5
```

However, **function constructors do not form closures** over the local environment where they are created — they form closures only over the global scope.

Thus:

- Functions created via `new Function` are **less secure**.
- They don't capture local variables.
- Rarely used today — modern JavaScript uses normal function expressions.

🕐 Disadvantages of Closures?

➔ Disadvantages:

1. **Memory leaks** — Variables captured by closures won't be garbage collected as long as closure exists.
2. **Overhead** — Extra memory retention can cause performance problems if not managed.
3. **Complexity** — Nested closures can make code difficult to understand and debug.

4. **Unintentional Sharing** — Mistakenly sharing same closure between different parts of code.

🕐 What is a Garbage Collector in JavaScript?

Garbage Collector is part of the JavaScript Engine responsible for **automatically freeing up memory** by removing objects that are no longer reachable.

JavaScript uses algorithms like **Mark and Sweep**:

- Variables that can be **reached** (from the root, global object) are kept.
- Variables that are **unreachable** are marked for deletion.

🕐 Relation between Garbage Collection, Memory Leaks, and Closures?

Closures can cause memory leaks when:

- Functions close over large objects or data.
- If the function is kept alive unnecessarily (e.g., attached to a long-living event listener).

Thus:

- Unused closures can **keep memory alive** even when it's no longer needed.

Good practice:
Explicitly nullify references when no longer needed.

Example:

```
function setup() {
    let hugeData = new Array(10000).fill("Memory");
    return function() {
        console.log(hugeData[0]);
    };
}

let call = setup();
// Later:
call = null; // allow hugeData to be garbage collected
```

Example 2:

```
function a() {
    var x = 0,
        z = 100;
    return function b() {
        console.log(x);
    };
}

var c = a();
```

Explanation:

- `b` only uses `x`, not `z`.
- Smart Garbage Collectors (like V8) **remove** `z` **from memory** because it's not needed.

☑ Closures only retain necessary variables if the engine is optimized.

🕐 Example of Smart Garbage collection by V8 JS Engine in Chrome

Modern JS engines like **V8** are smarter:

- They can detect closures.
- If an inner function does not **use** some outer variables, it won't unnecessarily keep them alive.

Thus:

- If a closure doesn't *actually use* a variable, it can be garbage collected.
- Improves memory efficiency.

Episode 13: — "First-Class Functions 🔥 ft. Anonymous Functions"

What is a Function Statement in JS?

The concept of **Function Statement** is introduced.

Definition:

> ***Function Statement is simply the normal way of declaring a function in JavaScript.***

Example shown:

```
function a() {
    console.log("Function Statement called");
}
```

Explanation:

- `function a() {}` is a **Function Statement**.
- It's also called a **Function Declaration** (more on this shortly).
- This way of creating a function **gives it a name** (`a`) and **hoists** it to the top of the scope (explained soon).
- It can be called even **before** its definition appears in code because of **hoisting**.

What is a Function Expression?

After understanding Function Statement, **Function Expression** is introduced.

Definition:

> ***When a function is assigned to a variable, it becomes a Function Expression.***

Example shown:

```
var b = function() {
    console.log("Function Expression called");
};
```

Explanation:

- A **function** is created **without a name** (anonymous) and **assigned to a variable** `b`.
- Now, the variable `b` holds a function.
- To call this function, you use `b()`.

Important:

- Unlike function statements, **function expressions are not hoisted** in the same way.
- Calling `b()` before this line would give an **error**.

Difference between Function Statement and Function Expression

Here, the **difference** between the two is explained clearly:

Feature	Function Statement	Function Expression
Definition	Normal function declaration	Function assigned to a variable
Hoisting	Fully hoisted	Variable is hoisted, but not function
Calling before definition	Works	Error (undefined or TypeError)

Example for clarity:

```
// Function Statement
a(); // ☑ Works
function a() {
    console.log("Function Statement");
}

// Function Expression
b(); // ✗ Error: Cannot access 'b' before initialization
var b = function() {
    console.log("Function Expression");
};
```

Explanation:

- **Function Statements** are **hoisted completely**, so you can call them even before defining.
- **Function Expressions** only hoist the **variable declaration** (not assignment), so before the assignment line, `b` is `undefined`, and calling it gives an error.

Remember:
In interviews, mentioning hoisting difference makes a **great impression**!

What is a Function Declaration?

Here it's clarified that:

- **Function Declaration** is just **another name** for **Function Statement**.
- Both mean **declaring a function normally** like:

```
function a() {
    console.log("Function Declaration/Statement");
}
```

Thus:

- **Function Statement = Function Declaration**.
  (They are just two names for the same thing.)

What is an Anonymous Function in JavaScript?

Definition:

> ***Anonymous Functions are functions without a name.***

Example:

```
function() {
    console.log("Anonymous Function");
}
```

Problem:

- **Syntax Error** ❌
- In JavaScript, you **cannot** declare a standalone function **without a name**.

Thus, an **anonymous function** must be **used as part of an expression**, e.g., assigned to a variable, passed as an argument.

Syntax Error & Anonymous Functions

When you try to write just:

```
function() {}
```

It will immediately throw an error:

* SyntaxError:* Function statements require a function name

Thus, anonymous functions must be:

- Assigned to variables
- Passed directly into other functions (as callbacks)

Example (correct):

```
var x = function() {
    console.log("Anonymous assigned to variable");
};
```

or

```
setTimeout(function() {
    console.log("Anonymous function used as callback");
}, 1000);
```

Use/Advantages of Anonymous Functions

Why do we use Anonymous Functions?
Three main reasons:

1. **Assign them to variables** easily:

```
var hello = function() {
    console.log("Hello Anonymous");
};
```

2. **Pass them directly as arguments** (callbacks):

```
setTimeout(function() {
    console.log("Delay done");
}, 2000);
```

3. **Return them from other functions**:

```
function outer() {
    return function() {
        console.log("Returned Anonymous Function");
    };
}
var inner = outer();
inner();
```

Thus, anonymous functions make code:

- Shorter
- More dynamic
- Great for cases where functions are **needed only once**

What are Named Function Expressions in JS?

Now comes **Named Function Expressions**.

Definition:

> ***A Function Expression where the function has a name.***

Example:

```
var y = function xyz() {
    console.log("Named Function Expression");
};
```

Here:

- `xyz` is the **name** of the function inside the expression.
- `y` is the **variable name** holding the function.

Important:

- `xyz` **is only available inside the function itself**, not outside.
- You can **only call** `y()`, not `xyz()` from outside.

Corner Case Gotcha using Named Function Expression

Important gotcha:

Example:

```
var z = function pqr() {
    console.log(pqr); // ☑ Works inside
};

z(); // Calls the function
pqr(); // ✗ ReferenceError: pqr is not defined
```

Explanation:

- `pqr` is only defined **inside the function's local scope**.
- Outside, it **does not exist**.
- Thus, calling `pqr()` directly throws **ReferenceError**.

Interview Tip:

- Explaining this gotcha shows deep knowledge!

What is the difference between Parameters & Arguments?

Clear distinction made:

Word	Meaning
Parameters	Names listed when defining a function
Arguments	Values passed when calling the function

Example:

```
function sum(a, b) {
    // a, b are PARAMETERS
    console.log(a + b);
```

```
}

sum(2, 3); // 2, 3 are ARGUMENTS
```

Summary:

- **Parameters** = placeholders inside the function
- **Arguments** = actual data you supply when invoking the function

First-Class Functions in JavaScript

Finally, introduction to **First-Class Functions**.

Meaning:

> ***In JavaScript, functions can be:***
>
> - ***Passed as arguments***
> - ***Returned from functions***
> - ***Assigned to variables***
>
> ***i.e., functions are treated like values.***

Thus, functions behave like **first-class citizens**.

Functions are First-Class Citizens

Explanation of **"Functions are First-Class Citizens"**:

Meaning:

- Functions have the **same rights** as any other value (like strings, numbers).
- You can store them, return them, pass them, treat them as data.

Examples already shown earlier:

- Passing functions (callback)
- Returning functions
- Assigning functions

Thus:

- In JavaScript, **functions are first-class citizens** because they are **first-class objects**.

Episode 14: 🎧 Callback Functions ft. Event Listeners

What is a Callback Function in JavaScript?

A **Callback Function** is nothing but:

- A function that is **passed as an argument** into another function.
- Later, this passed function is **executed inside** the outer function.

In simple words:
One function gives **control** to another function by passing a function to it.
And when the time comes, the outer function **calls back** the passed function.

Functions in JavaScript behave like **normal values**.

- Just like numbers, strings, arrays can be passed to functions —
- Functions too can be **passed as arguments**.

JavaScript treats functions as **first-class citizens** — this ability is what makes callback functions possible.

Advantages of Callback Functions

Callback functions are extremely important because they provide:

- **Powerful flexibility** — you can decide what should happen after a task completes.
- **Decoupling** — the main function doesn't have to know what the callback does; it just calls it.
- **Modularity** — you can break large functionalities into smaller, independent units.
- **Control over when to run something** — callback ensures certain code runs **after** something else is done.

Imagine a scenario:

- You ordered a pizza .
- You gave your phone number to the restaurant.
- They will **call you back** when your pizza is ready.

You don't have to wait at the restaurant. You continue your life.
When the pizza is ready — **they call you**. This is exactly how **callback functions** work in JavaScript.

Demo of Callback Functions in DevTools (Browser Demo)

Now let’s open the browser console (DevTools) and see it practically.

First, define two simple functions:

```
function x(y) {
    console.log("Inside function x");
    y();
}

function y() {
    console.log("Inside function y");
}
```

Now pass function `y` as an argument to function `x` and call:

```
x(y);
```

Output:

```
Inside function x
Inside function y
```

Here:

- `y` is passed into `x`.
- Inside `x`, the function `y` is **executed**.

This simple code shows the working of a **callback function**.

You can even pass an **anonymous function** directly:

```
x(function() {
    console.log("Anonymous function called!");
});
```

Output:

```
Inside function x
Anonymous function called!
```

Thus, **functions can be passed directly** — they don't even need a name.

Blocking the Main Thread in JavaScript

Now it's important to understand:
JavaScript is single-threaded — meaning, it has only **one call stack**.

So if one heavy task (e.g., heavy computation or network request) is running, it **blocks** the entire thread.

Example:

```
console.log("Start");

setTimeout(function() {
    console.log("Inside setTimeout");
}, 5000);

console.log("End");
```

Output:

```
Start
End
Inside setTimeout
```

Even though the timer is for 5 seconds, the thread is **free** after initiating the timer.
That's why `End` is logged immediately after `Start`.

In heavy operations, **callbacks help** — they make JavaScript **non-blocking**.
They allow JavaScript to **continue doing other things** while waiting for some operation to complete.

Thus, callbacks are key for **asynchronous behavior**.

Creating an Event Listener in JavaScript

Event listeners are a very real-world, daily-life example where **callback functions** are used.

Example:

```
<button id="clickMe">Click Me</button>
```

```
document.getElementById("clickMe").addEventListener("click", function xyz() {
    console.log("Button Clicked!");
});
```

Explanation:

- `addEventListener` takes two arguments:
  1. Type of event — `"click"`
  2. Callback function — `xyz`
- When the user clicks the button, the callback function `xyz` is **executed**.

This is how **Event Listeners** are built upon **callback functions**.

Whenever the event happens, **JavaScript calls the callback function** provided.

You can even pass an **anonymous function** instead of naming it:

```
document.getElementById("clickMe").addEventListener("click", function() {
    console.log("Button Clicked Again!");
});
```

Notice:

- Here we pass an unnamed function directly.
- It will still behave exactly the same.

Thus, **event listeners and callbacks go hand in hand**.

Closures along with Event Listeners

Now the story gets a little deeper —
when you attach an Event Listener, and the callback function **uses a variable** from its outer environment, **closures** come into play.

Example:

```
function attachEventListener() {
    let count = 0;

    document.getElementById("clickMe").addEventListener("click", function() {
        console.log("Button clicked", ++count);
```

```
    });
}

attachEventListener();
```

Explanation:

- `count` is a variable inside the `attachEventListener` function.
- The callback inside `addEventListener` **closes over** (remembers) the `count` variable.
- Every time the button is clicked, it **remembers** the updated `count`.

This happens because of **closures**:

- Even after `attachEventListener` has finished executing,
- The callback still **retains access** to the `count` variable.

Thus, Event Listeners and Closures are **deeply connected**.

Without closures, `count` would be lost after the function ends.

Closures allow **preserving the scope** and **retaining variables** even after the parent function is gone.

Garbage Collection and Removing Event Listeners

In JavaScript, unused memory is automatically cleaned up by the **Garbage Collector**.

However, if you **don't remove Event Listeners**, they **hold references** to DOM elements and can cause:

- **Memory leaks**
- **Performance issues**

Thus, it's important to **manually remove event listeners** when they are no longer needed.

How to remove an event listener:

```
const button = document.getElementById("clickMe");

function onClick() {
    console.log("Button Clicked!");
}

button.addEventListener("click", onClick);

// Later when you want to remove it
button.removeEventListener("click", onClick);
```

Important points:

- You need to **pass the exact same reference** to `removeEventListener` that you passed in `addEventListener`.
- Otherwise, it won't work.

If you use **anonymous functions** directly while adding the listener, you **cannot remove** them later, because you don't have a reference.

Thus, **named functions are recommended** when you might need to remove event listeners later.

Garbage collection becomes easier and memory management improves if event listeners are properly cleaned up.

The Event Loop will explain the true behind-the-scenes working of JavaScript's async behavior.

✨ Final Summary

Topic	Insights
Callback Functions	Functions passed into another function to be executed later
Advantages	Modularity, Asynchronous handling, Control, Flexibility
Event Listeners	Practical real-world usage of callback functions
Closures + Event Listeners	Callback functions can access variables from parent scopes
Garbage Collection	Remove event listeners to prevent memory leaks

Episode 15: 🔁 Asynchronous JavaScript & Event Loop from Scratch — Full Detailed Notes

How JavaScript Engine Executes the Code using Call Stack

JavaScript is a **synchronous**, **single-threaded** language.

When a JavaScript program runs, it follows a simple model:

- There is a **Call Stack**.
- It picks code **line by line** and **executes it**.
- If a function is called, it is **pushed onto the Call Stack**.
- When the function finishes execution, it is **popped out** of the Call Stack.

The main job of the Call Stack is to keep track of:

- **Where** we are in the code.
- **Which functions** are running.
- **What to execute next**.

Thus, the **Call Stack** is like a manager — it controls the flow of code execution in a synchronous manner.

Main Job of the Call Stack

The Call Stack's **primary job** is:

☑ Managing function invocations.
☑ Keeping track of function calls.
☑ Running synchronous code in a **last-in, first-out (LIFO)** way.

If everything was synchronous, JavaScript would simply execute everything **one by one**.
However, **real-world applications** are not purely synchronous.
We have to deal with:

- Timers
- User interactions
- API calls
- File reading

- Animations
- And much more...

These are **asynchronous tasks**.

How does JavaScript perform async tasks

Since JavaScript is single-threaded, how does it handle async tasks without blocking the code?

The trick is: **JavaScript engine itself cannot handle async operations**.
Instead, it **outsources** them to the **Browser environment**.

When async code is encountered, JavaScript hands off the task to the **Browser** (or environment like Node.js).

Example:
When `setTimeout` , `fetch` , `DOM events` , etc., are called:

- They are handled by the browser.
- Meanwhile, JavaScript engine **keeps executing** the rest of the code.

Thus, JavaScript itself remains synchronous and single-threaded, but it collaborates with the browser to achieve asynchronous behavior.

Behind the Scenes in Browser

Behind the scenes, the Browser provides a huge infrastructure:

- **Web APIs** (SetTimeout, DOM API, Fetch API, etc.)
- **Callback Queue** (for storing callbacks waiting to be executed)
- **Event Loop** (constantly monitoring the stack and queue)

So when an async task like `setTimeout` or `fetch` is triggered:

- The Browser's Web API section **takes responsibility**.
- Once the async task finishes (e.g., timer expires or HTTP response comes), the **callback function** is pushed into the **Callback Queue**.

But it doesn't immediately execute!

Execution only happens when the **Call Stack is empty** — thanks to the **Event Loop**.

Thus, the browser acts like a smart assistant, handling heavy work and signaling JavaScript when to continue.

Web APIs in JS

The Web APIs provided by the browser include:

- **DOM APIs** (document, events, etc.)
- **Timers** (`setTimeout` , `setInterval`)
- **HTTP Requests** (`fetch` , `XMLHttpRequest`)
- **Geolocation API**
- **Web Storage APIs** (`localStorage` , `sessionStorage`)

These APIs **work separately** from the JavaScript Call Stack.

When we trigger a timer or make an HTTP request, the Web API:

- Starts the timer or request in the background.
- After completion, registers a **callback function** into the Callback Queue.

JavaScript doesn't do the waiting —
it hands over the job and **continues execution immediately**.

How setTimeout Works Behind the Scenes in Browsers

`setTimeout` is one of the simplest examples to understand asynchronous behavior.

Suppose we write:

```
console.log("Start");

setTimeout(function() {
    console.log("Timeout Callback");
}, 5000);

console.log("End");
```

Here's what happens:

1. "Start" is printed immediately.
2. `setTimeout` is picked by the **Web API**. A 5-second timer is started **outside** JavaScript engine.
3. "End" is printed immediately (synchronous).
4. After 5 seconds, the callback (`console.log("Timeout Callback")`) is **moved into the Callback Queue**.
5. **Event Loop** checks:

- If Call Stack is empty,
- Then it **pushes the callback onto the Call Stack**.

6. Finally, the callback executes, printing "Timeout Callback".

Important:
Even if `setTimeout` is 0 milliseconds, it **does not guarantee immediate execution**.
It still waits for the Call Stack to be empty.

Event Loop & Callback Queue in JS

The **Event Loop** plays a **vital role** in async behavior.

Its job:

- Keep checking whether the **Call Stack** is empty.
- If empty, pick the first function from the **Callback Queue** and **push it onto the Call Stack** for execution.

Thus, the Event Loop acts like a **traffic controller** —
ensuring that callbacks don't interrupt running synchronous code but are executed at the right time.

Without Event Loop, there would be chaos — callbacks could jump in randomly.

How Event Listeners Work in JS

When we add an Event Listener, e.g.,

```
button.addEventListener("click", function() {
    console.log("Button Clicked");
});
```

Here's what happens:

- The **Browser's Web API** registers the listener.
- It **listens** to the click event in the background.
- It does **NOT** block the Call Stack.
- When the button is clicked:
  - The callback (`console.log('Button Clicked')`) is pushed into the **Callback Queue**.
  - Event Loop picks it up when Call Stack is empty and executes it.

Thus, Event Listeners are another great example of **asynchronous collaboration** between JavaScript and the Browser.

More About Event Loop

The Event Loop's **core principle** is:

> ***Never let anything from Callback Queue enter the Call Stack until the Stack is empty.***

Thus:

- All synchronous code gets **priority**.
- Async callbacks **wait** their turn patiently in the queue.

This behavior guarantees the **predictability** of JavaScript execution.

Why do we need Event Loop

Imagine if asynchronous tasks directly jumped into the Call Stack.
The running synchronous code would be interrupted at random points — causing bugs, unexpected behavior, and a lot of instability.

The Event Loop solves this by:

- **Queueing async callbacks**
- **Executing them only when safe**

Thus, the Event Loop **preserves** the synchronous nature of JavaScript, while still allowing asynchronous operations to exist harmoniously.

How fetch() function works

When you use `fetch()` :

```
fetch("https://api.example.com/data").then(function(response) {
    console.log("Data received");
});
```

Here's the sequence:

1. `fetch` is a Web API.
2. It initiates a **network request** in the background (Browser or Node APIs).
3. JavaScript **moves on** without waiting.
4. When response is received:

- The `.then()` callback is **not** sent to Callback Queue.
- It is sent to a **special queue called the Microtask Queue**.

Thus, Promises (`.then` , `.catch` , `.finally`) behave slightly differently than simple setTimeout callbacks.

They use a **Microtask Queue**, which has **higher priority** over Callback Queue.

MicroTask Queue in JS

MicroTask Queue is a special queue for:

- `.then` handlers of Promises
- `MutationObservers`
- `queueMicrotask()`

Rules of MicroTask Queue:

- After every task from Call Stack finishes, **before** picking from Callback Queue, **Microtask Queue is emptied completely**.

Thus, Promises (Microtasks) are always processed **before** setTimeout callbacks (Macro tasks).

What are MicroTasks in JS?

MicroTasks are small tasks that need to be executed **immediately after** the currently executing function finishes and **before** any other macrotask (like timers, events, etc.)

Example:

```
console.log("Script Start");

setTimeout(function() {
    console.log("Timeout");
}, 0);

Promise.resolve().then(function() {
    console.log("Promise 1");
});

console.log("Script End");
```

Output:

```
Script Start
Script End
Promise 1
Timeout
```

Here:

- Promise callback executes **before** setTimeout, because Microtasks (Promises) have higher priority over Macrotasks (Timers).

Starvation of Functions in Callback Queue

If Microtasks keep getting created endlessly inside each other (like chained Promises):

- The Microtask Queue will **never get empty**.
- The Callback Queue (timers, events) will **starve** — meaning they never get a chance to execute.

Thus, excessive use of Promises without breaks can cause **starvation** of Macrotasks.

Real-world example:

- Huge infinite Promise chains can block even basic events like clicks, typing, UI updates.

Thus, **balance is important** between Microtasks and Macrotasks.

✨ In Summary

🟢 JavaScript engine = Single-threaded, synchronous.

🟢 Browser = Provides Web APIs, Callback Queue, MicroTask Queue, Event Loop.

🟢 Asynchronous tasks = Handed over to Web APIs, completed in background.

🟢 Event Loop = Coordinates when tasks can safely execute.

🟢 Microtask Queue = Promises and high-priority tasks.

🟢 Callback Queue = Timers, events, and normal async tasks.

🟢 Microtasks always execute before Macrotasks.

🟢 Excessive Microtasks can cause starvation of Macrotasks.

Thus, the Event Loop + Callback Queue + Microtask Queue =
The heart of Asynchronous JavaScript!

Episode 16: JS Engine EXPOSED 🔥 | Namaste JavaScript Ep.16 (with updated, complete coverage):

What is JavaScript Runtime Environment (JSRE)?

JavaScript Runtime Environment (JSRE) is the environment where JavaScript code is executed.
It is a **bundle of components** that helps JavaScript code:

- **Run**
- **Interact with APIs**
- **Handle asynchronous tasks**
- **Manage memory**

JavaScript itself only knows how to execute the logic.
But the environment around it (JSRE) provides a lot more — like access to timers, the DOM (in browsers), and filesystem APIs (in Node.js).

Thus:

- JS Engine → Only responsible for running JS code.
- JS Runtime Environment → Adds things like Web APIs, Callback Queues, Event Loop, Microtask Queue, etc.

Browser vs Node.js — Different JavaScript Runtime Environments

JavaScript is used in two major environments:

1. **Browsers** (like Chrome, Firefox, Safari)

In the browser:

- The Runtime Environment includes things like:
  - DOM APIs (`document` , `window`)
  - Web APIs (`setTimeout` , `fetch`)
  - Event Loop
  - Callback Queue

2. **Node.js** (server-side)

In Node.js:

- The Runtime Environment includes things like:
  - File System APIs
  - Networking APIs
  - Timers
  - Event Loop
  - Microtask Queue

Thus, depending on where JavaScript is running, the **available APIs and features differ**, but the **core JS Engine** (like V8) remains the same.

The diagram explained visually:

- At the center → **JavaScript Engine (V8, SpiderMonkey, etc.)**
- Around it → Different APIs (provided by Browser or Node.js)
- Managed by → Event Loop, Callback Queue, Microtask Queue

List of Popular JavaScript Engines

There are different JavaScript Engines used in different environments:

- **V8** → Chrome, Node.js
- **SpiderMonkey** → Firefox
- **JavaScriptCore** → Safari
- **Chakra** → Old Microsoft Edge
- **Hermes** → React Native

Each engine is responsible for understanding and executing JavaScript code.

First JavaScript Engine Ever Created

The **first JavaScript Engine** was created by **Brendan Eich** in 1995 for **Netscape Navigator** browser.

It was called **SpiderMonkey**.

- Brendan Eich wrote it in just **10 days**.
- The focus was **quick scripting** to enhance websites (simple validation, interaction).
- At that time, performance and scalability were **not priorities**.

Modern engines like V8 have come a long way since then.

Myths about JavaScript Engines

There are several **myths** about JS Engines:

- People think JS Engines handle everything (DOM, timers, networking, etc.)
  → **Wrong.**
  Only the JS code execution part is handled by the JS Engine.
- Some think V8 Engine is only for Chrome.
  → **Wrong.**
  Node.js also uses V8.

The main job of the JS Engine is simple but powerful:

- Parse JS code
- Execute JS code
- Manage memory allocation (Heap, Stack)
- Optimize code during runtime

JavaScript Engine Architecture

All JS Engines, including V8, share a common architecture with two important components:

1. **Memory Heap**

- An unorganized space to allocate memory.
- Objects, functions, arrays live here.

2. **Call Stack**

- Manages the execution of functions.
- Follows LIFO (Last In, First Out) structure.

Together:

- Heap → For storing data.
- Stack → For managing execution flow.

Syntax Parsers and Abstract Syntax Tree (AST)

When JavaScript code is loaded into the JS Engine, it first goes through **Parsing**.

- **Syntax Parser** reads the code and converts it into tokens.
- These tokens are analyzed and a **tree-like structure** is created → the **Abstract Syntax Tree (AST)**.

Example:
Code:

```
let a = 5;
```

AST Representation:

- `VariableDeclaration`
- `Identifier` (`a`)
- `Literal` (`5`)

The AST is the blueprint that tells the Engine what needs to be executed and in what order.

Compilation and Execution of JS Code

After parsing:

- The AST is either **interpreted** (line-by-line execution)
- Or **compiled** (transformed into optimized machine code)

In modern engines, it is a combination of **interpretation + compilation**.

Thus, JavaScript achieves:

- **Faster startup time** (using interpreter)
- **Faster execution** (using compiler)

Just In Time Compilation (JIT)

Modern engines like V8 use **JIT Compilation**.

- Code is initially **interpreted** into **Bytecode** (via Ignition in V8).
- As code runs, the engine **monitors** which parts of the code are used most often (hot code).
- Those parts are then **recompiled** and **optimized** into machine code (via TurboFan in V8).

Thus:

- First load → Fast because of quick interpretation.
- Long running → Fast because of smart compilation and optimization.

This combination allows JavaScript to be **dynamic, flexible, and fast**.

Is JavaScript Interpreted or Compiled?

Historically, JavaScript was considered **interpreted**.
But today, **JavaScript is both interpreted and compiled**.

In modern engines:

- Initial parsing + interpretation for quick execution.
- Later compilation for heavy optimization.

Thus, technically:

> ***JavaScript is interpreted at first, but compiled Just-In-Time (JIT) for performance optimization.***

Garbage Collector — Mark & Sweep Algorithm

Managing memory is extremely important.

The **Heap** can grow indefinitely if unused memory is not cleared.
To manage this, JS Engines use **Garbage Collection**.

The **Mark and Sweep Algorithm** is widely used:

- **Mark Phase**:
  - Find and mark all variables and objects that are still reachable (being used).
- **Sweep Phase**:
  - All unmarked memory is considered garbage and is **freed**.

Thus:

- Memory that's no longer referenced → Cleared.
- Memory that's still reachable → Retained.

Garbage Collection happens periodically and automatically.

You don't have to manually free memory in JavaScript (unlike C++).

Fastest JavaScript Engine

As of today, **Google's V8 Engine** is considered the fastest JavaScript Engine.

- Extremely optimized.
- Powerful JIT compilation.
- Continuously updated.
- Used in Chrome and Node.js.

Other engines are also fast (like SpiderMonkey and JavaScriptCore), but V8 often leads benchmarks.

Google's V8 JavaScript Engine Architecture

Let's specifically focus on V8.

The V8 Architecture includes:

1. **Parser** → Converts code to AST.
2. **Ignition (Interpreter)** → Converts AST to Bytecode for quick execution.
3. **TurboFan (Compiler)** → Optimizes hot code into super-fast machine code.
4. **Memory Heap** → For dynamic allocation.
5. **Call Stack** → For execution management.
6. **Garbage Collector** → For cleaning up unused memory.

Thus, V8 is a beautifully engineered combination of:

- **Fast startup** (thanks to Ignition).
- **Fast long-term performance** (thanks to TurboFan).
- **Memory management** (thanks to Garbage Collector).

Quick Revision of JavaScript Runtime Environment and JS Engine

To summarize:

🟤 **JS Engine** → Executes JavaScript code
🟤 **Memory Heap** → Dynamic memory allocation
🟤 **Call Stack** → Function execution
🟤 **Web APIs** → Provided by Browser or Node.js, not JS Engine
🟤 **Callback Queue / Microtask Queue** → Manage async code
🟤 **Event Loop** → Coordinates between Engine and API tasks

And in V8 specifically:

- AST → Bytecode (Ignition) → Machine Code (TurboFan)

All this ensures JavaScript remains:

- Single-threaded.
- Non-blocking.
- Super fast.
- Highly flexible.

Episode 17: Trust Issues with setTimeout()

Opening

When we use `setTimeout()` , we often **trust** that if we write:

```
setTimeout(callback, 2000);
```

the callback will definitely execute **after exactly 2 seconds**.

But this trust is **not fully true**.

The **JavaScript engine** does not guarantee **execution exactly after 2 seconds**.
There are **trust issues** with `setTimeout()` .

Understanding How setTimeout Really Works

When we use `setTimeout(callback, 2000)` :

- The browser sets a **timer** for 2 seconds.
- After 2 seconds, the browser **moves the callback function** into the **Task Queue**.
- Now, the **Event Loop** monitors:
  - If the **Call Stack** is **empty**, then the Event Loop will **push** the callback into the Call Stack for execution.

Thus:

- Even though 2 seconds have passed, **if the Call Stack is busy**, the callback will **have to wait** in the Task Queue.

This is the **main trust issue** with `setTimeout()` .

Simple Example

```
console.log("Start");

setTimeout(function cb() {
    console.log("Callback");
}, 5000);

console.log("End");
```

Execution:

1. "Start" is printed.
2. `setTimeout()` is encountered:

- Browser sets a 5-second timer.
- Registers the callback to be executed after 5 seconds.

3. "End" is printed immediately.
4. After 5 seconds:

- The callback moves to the **Task Queue**.
- If Call Stack is empty → Event Loop moves callback to Call Stack → "Callback" is printed.

Thus, output:

```
Start
End
(wait 5 sec)
Callback
```

Heavy Code Blocking Execution

Suppose there is heavy synchronous code after `setTimeout()` .
Example:

```
console.log("Start");

setTimeout(function cb() {
    console.log("Callback");
}, 2000);

for (let i = 0; i < 10000000000; i++) {
    // Heavy synchronous task
}

console.log("End");
```

- "Start" gets printed immediately.
- `setTimeout()` is set for 2 seconds.
- Then the **huge loop** starts running.
- **Even after 2 seconds**, callback moves to Task Queue but **cannot execute** because the Call Stack is busy running the loop.
- Only after loop and "End" are printed and Call Stack is empty, the callback is picked from Task Queue and executed.

Thus:

Real output order:

```
Start
(wait heavy time)
End
Callback
```

Main Point About setTimeout()

> `setTimeout()` ***ensures that the callback will be queued after at least that time (2 seconds, 5 seconds, etc.)***
>
> ***It does not guarantee that the callback will be executed immediately after the timer ends.***

Execution can be **delayed** if the **Call Stack** is busy.

Another Example with 0ms setTimeout

```
setTimeout(function() {
    console.log("Timeout 1");
}, 0);

setTimeout(function() {
    console.log("Timeout 2");
}, 0);

console.log("Synchronous Code");
```

Even though both `setTimeout()` functions are given `0` milliseconds:

- Their callbacks are still **moved to Task Queue**.
- **Synchronous code** (`console.log("Synchronous Code")`) is executed first.

Thus, the output will be:

```
Synchronous Code
Timeout 1
Timeout 2
```

Explanation

- Even `0ms` does not guarantee that the callback will execute immediately.
- Synchronous code always has **higher priority** than asynchronous code.
- Callbacks registered by `setTimeout()` will only be executed when:
  - The minimum timer is done.
  - The Call Stack is empty.

Thus, **timers are minimum time guarantees, not execution guarantees**.

Summary from the Video

- **setTimeout() trust issues** happen because the JavaScript engine cannot guarantee exact timing of execution.
- It can guarantee **only when** to **move** the callback to the Task Queue, not when to **execute** it.
- **Heavy synchronous tasks** can **block the execution** of setTimeout callbacks.
- Even with **0ms delay**, callbacks are placed in the Task Queue and will execute **after** all synchronous code.

Episode 18: Higher Order Functions and Functional Programming

What are Higher Order Functions?

A **Higher Order Function** is a function that either:

- Takes another function as an argument
- **OR** returns another function as its output

Any function that deals with other functions this way is called a Higher Order Function.

Example of Higher Order Function

```
function x() {
    console.log("Inside x");
}

function y(x) {
    console.log("Inside y");
    x();
}

y(x);
```

Explanation:

- Here, `y` is a Higher Order Function because it **accepts** another function `x` as an argument and **calls** it.

Common Real-Life Examples of Higher Order Functions

- `setTimeout`
- `setInterval`
- `map`
- `filter`
- `reduce`
- `then` and `catch` in Promises

Introduction to Functional Programming

Functional Programming is a way of writing code where:

- Functions are treated as first-class citizens (passed, returned, stored like variables).
- The focus is on **pure functions** and **immutability**.
- The goal is to avoid shared state and side-effects.

Key Concepts of Functional Programming

1. Pure Functions

- A pure function always returns the same output for the same input.
- It has **no side effects** (doesn't modify global variables, files, database, etc.).

2. Immutability

- Instead of changing existing data, new copies of data are created and modified.

3. First-Class and Higher-Order Functions

- Functions can be passed around just like normal variables.
- Higher-order functions either take functions as arguments or return functions.

Code Example Showing Functional Approach

```
const radius = [1, 2, 3, 4];
```

```
const calculateArea = function(radius) {
    return Math.PI * radius * radius;
};

const calculateDiameter = function(radius) {
    return 2 * radius;
};

const calculateCircumference = function(radius) {
    return 2 * Math.PI * radius;
};

const calculate = function(radius, logic) {
    const output = [];
    for (let i = 0; i < radius.length; i++) {
        output.push(logic(radius[i]));
    }
    return output;
};

console.log(calculate(radius, calculateArea));
console.log(calculate(radius, calculateDiameter));
console.log(calculate(radius, calculateCircumference));
```

Explanation:

- `calculate` is a Higher Order Function.
- It accepts a function (`logic`) that defines what calculation to perform on each radius.
- This **avoids code repetition** and **makes the code more modular**.

Mistakes People Make in Coding Interviews

- Writing repetitive, copy-paste style code.
- Not abstracting repeated logic into reusable functions.
- Forgetting about modularization and reusability.

DRY Principle (Don't Repeat Yourself)

The **DRY Principle** says:

> ***"Every piece of knowledge must have a single, unambiguous, authoritative representation within a system."***

In simple words:

- Don’t write the same code multiple times.
- If something is repeating, abstract it into a function.
- Write generic logic and reuse it with different behaviors.

How to Optimize Our Code

- Find repeating patterns in logic.
- Write Higher Order Functions that accept behavior as an argument.
- Keep functions **small**, **pure**, and **focused on a single task**.
- **Avoid side effects** wherever possible.

Beauty of Functional Programming

Functional Programming makes code:

- Shorter
- Cleaner
- Reusable
- Easier to debug
- Easier to test
- More predictable

By building everything with pure functions and small compositions, code becomes easier to maintain and scale over time.

Polyfill for `map()` Method in JavaScript

Understanding how `map()` works internally helps in deepening JavaScript skills.

Creating a custom `calculate()` function on arrays:

```
Array.prototype.calculate = function(logic) {
    const output = [];
    for (let i = 0; i < this.length; i++) {
        output.push(logic(this[i]));
    }
    return output;
};

const radius = [1, 2, 3, 4];

const area = function(radius) {
    return Math.PI * radius * radius;
};

console.log(radius.calculate(area));
```

Explanation:

- `Array.prototype.calculate` adds a new function to all arrays.
- `this` inside the function refers to the array.
- It iterates through each element, applies the logic, and returns a new array — just like `map()` does!

Recap

- **Higher Order Functions** either take functions as arguments or return functions.
- **Functional Programming** focuses on pure functions, immutability, and no side effects.
- Always follow the **DRY Principle**.
- Identify repeating logic and abstract it into reusable higher-order functions.
- Polyfilling methods like `map()` internally helps master JavaScript fundamentals deeply.

**Episode 19: map, filter & reduce **

Array.map() function in JavaScript

What is `map()` ?

- `map()` is used when you want to **transform an array**.
- It **creates a new array** by applying a function to each element of the original array.
- It **does not modify** the original array.
- It **returns a new array**.

➔ Syntax:

```
const newArray = arr.map(callbackFunction);
```

- `callbackFunction` is executed **on each element**.
- The result of `callbackFunction` is pushed into the new array.

Example 1 - map()

Suppose we have:

```
const arr = [5, 1, 3, 2, 6];
```

If we want to **double** each element:

```
const output = arr.map(function(x) {
    return x * 2;
});

console.log(output); // [10, 2, 6, 4, 12]
```

- For each element, the function is called.
- Each element is multiplied by 2 and returned into a new array.

Example 2 - map()

If we want to **square** each element:

```
const output = arr.map(function(x) {
    return x * x;
});

console.log(output); // [25, 1, 9, 4, 36]
```

- Each number is squared individually.
- A **new array** with the squares is created.

Example 3 - map()

If we want to **convert numbers into binary**:

```
const output = arr.map(function(x) {
    return x.toString(2);
});

console.log(output); // ['101', '1', '11', '10', '110']
```

- `toString(2)` converts a number into its **binary form**.
- Again, a **new array** with binary strings is returned.

Array.filter() function in JavaScript

What is `filter()` ?

- `filter()` is used to **filter elements** based on a **condition**.
- It will **return a new array** of elements that pass the given condition.
- It **does not modify** the original array.

➔ Syntax:

```
const newArray = arr.filter(callbackFunction);
```

- `callbackFunction` should return **true** to keep the element, or **false** to discard it.

Example 4 - filter()

Let's find all elements **greater than 4**:

```
const output = arr.filter(function(x) {
    return x > 4;
});

console.log(output); // [5, 6]
```

☑ Only elements greater than 4 are kept.

Example 5 - filter()

Find **odd numbers**:

```
const output = arr.filter(function(x) {
    return x % 2;
});

console.log(output); // [5, 1, 3]
```

- Odd numbers return **true** for `x % 2 !== 0`.

Example 6 - filter()

Find **even numbers**:

```
const output = arr.filter(function(x) {
    return x % 2 === 0;
});

console.log(output); // [2, 6]
```

- Even numbers return **true** for `x % 2 === 0`.

Array.reduce() function in JavaScript

What is `reduce()` ?

- `reduce()` takes all the elements and **reduces them to a single value**.
- It uses a **callback function** and **an accumulator**.
- It can be used for **sum**, **Find the maximum**, **Find the minimum**, **flattening arrays**, **counting**, etc.

➔ Syntax:

```
const result = arr.reduce(callbackFunction, initialValue);
```

- `callbackFunction` takes two arguments:
  - **accumulator**: accumulates the callback's return values.
  - **currentValue**: current element being processed.
- `initialValue` is optional but good practice to provide.

Example 7 - reduce()

Find the **sum** of all elements:

```
const output = arr.reduce(function(acc, curr) {
    acc = acc + curr;
    return acc;
}, 0);

console.log(output); // 17
```

☑ `acc` starts from `0` . ☑ We add each `curr` to `acc` . ☑ Final sum is 17.

Find the **maximum** element:

```
const output = arr.reduce(function(max, curr) {
    if (curr > max) {
        max = curr;
    }
    return max;
}, 0);
```

```
console.log(output); // 6
```

☑ At each step, we check if `curr` is greater than `acc` . ☑ If yes, update `acc` . ☑ At the end, we get the maximum value.

Example 8 - reduce()

Given:

```
const users = [{
        firstName: "Akshay",
        lastName: "Saini",
        age: 26
    },
    {
        firstName: "Donald",
        lastName: "Trump",
        age: 75
    },
    {
        firstName: "Narendra",
        lastName: "Modi",
        age: 71
    },
    {
        firstName: "Elon",
        lastName: "Musk",
        age: 50
    },
];
```

We want to find **how many users per age**:

```
const output = users.reduce(function(acc, curr) {
    if (acc[curr.age]) {
        acc[curr.age]++;
    } else {
        acc[curr.age] = 1;
    }
    return acc;
}, {});

console.log(output);
// {26: 1, 75: 1, 71: 1, 50: 1}
```

☑ Counting how many times each `age` appears.

Example 9 - tricky map()

If we want to get **first names** of all users:

```
const output = users.map(function(x) {
    return x.firstName;
});

console.log(output);
// ["Akshay", "Donald", "Narendra", "Elon"]
```

☑ Extracting a specific property from objects.

Example 10 - tricky reduce()

Find **full names of users whose age is less than 60** using **reduce**:

```
const output = users.reduce(function(acc, curr) {
    if (curr.age < 60) {
        acc.push(curr.firstName + " " + curr.lastName);
    }
    return acc;
}, []);

console.log(output);
// ["Akshay Saini", "Elon Musk"]
```

☑ Using `reduce` to **filter** and **map** in one go.

. Create full names of users who are younger than 60

Combining filter + map:

```
const fullNames = users
    .filter((user) => user.age < 60)
    .map((user) => user.firstName + " " + user.lastName);

console.log(fullNames);
// Output: ["Akshay Saini", "Elon Musk"]
```

☑ First **filter**, then **transform**.

Example 11 - Chaining map, filter & reduce

Imagine you want to **square the numbers**, **filter numbers greater than 10**, and **sum them**.

Using chaining:

```
const arr = [5, 1, 3, 2, 6];
```

```
const output = arr
    .map((x) => x * x)
    .filter((x) => x > 10)
    .reduce((acc, x) => acc + x, 0);

console.log(output);
// 61
```

Step-by-step:

- **map**: [25, 1, 9, 4, 36]
- **filter**: [25, 36]
- **reduce**: 25 + 36 = 61

☑ Chaining leads to **beautiful**, **clean**, and **functional** code.

Homework - Challenge

Challenge for you:

- Try to **chain map, filter, and reduce** for different problems.
- Example: Pick users aged below 60, create their full names, and find the total characters across all full names.

Think in a **functional way**! 🚀