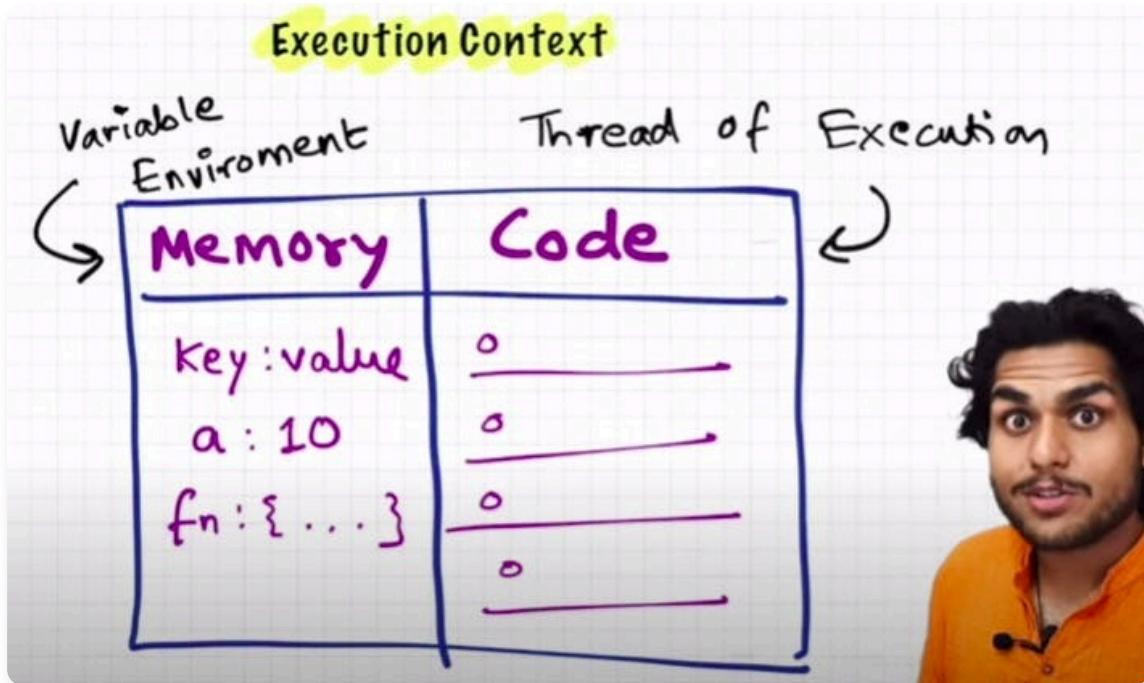


Episode 1 : Execution Context

- Everything in JS happens inside the execution context. Imagine a sealed-off container inside which JS runs. It is an abstract concept that holds info about the env. within the current code is being executed.



- In the container the first component is **memory component** and the 2nd one is **code component**
- Memory component has all the variables and functions in key value pairs. It is also called **Variable environment**.
- Code component is the place where code is executed one line at a time. It is also called the **Thread of Execution**.
- JS is a **synchronous, single-threaded** language
 - Synchronous:- In a specific synchronous order.
 - Single-threaded:- One command at a time.

Watch Live On Youtube below:



Episode 2: How JavaScript is Executed & Call Stack

What You'll Learn

- How JavaScript creates and manages execution contexts
 - The two phases of execution context creation
 - Call Stack and how it manages execution order
-

Understanding Execution Context



When a JavaScript program runs, a **Global Execution Context** is created automatically.

How Execution Context Works

Phase	Description	What Happens
1. Memory Creation Phase	Allocates memory to variables and functions	<ul style="list-style-type: none"> Variables get <code>undefined</code> Functions get their complete code
2. Code Execution Phase	Executes code line by line	<ul style="list-style-type: none"> Variables get their actual values Functions are invoked

Let's See This in Action



Let's trace through this code step by step:

```
var n = 2;
function square(num) {
  var ans = num * num;
  return ans;
}
```

```
var square2 = square(n);
var square4 = square(4);
```

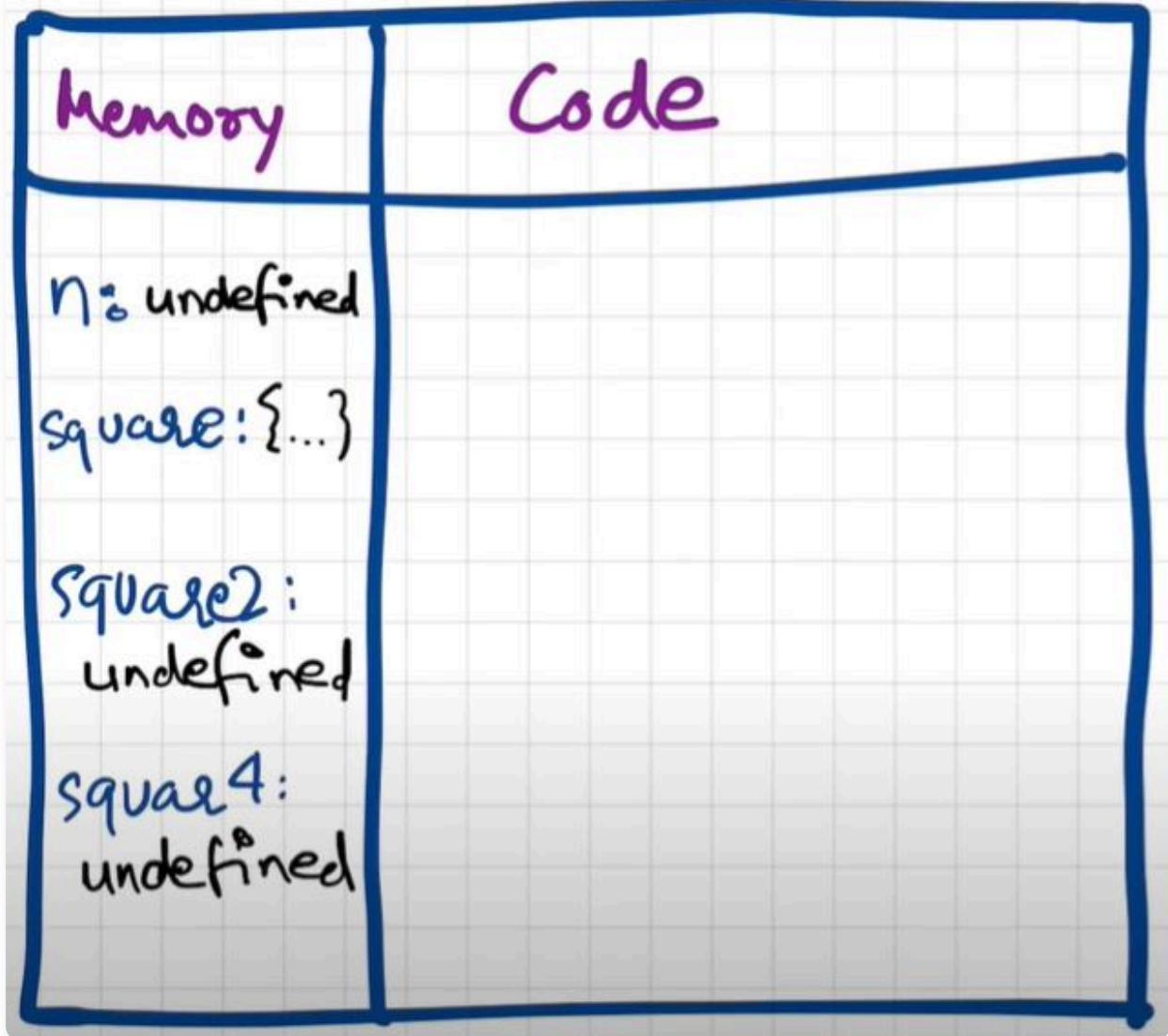
Breaking Down the Execution 🔎

Phase 1: Memory Creation Phase 🧠

JavaScript first scans through the entire code and allocates memory:

Line	Variable/Function	Memory Allocation
<code>var n = 2;</code>	<code>n</code>	<code>undefined</code>
<code>function square(num)</code> <code>{...}</code>	<code>square</code>	Complete function code
<code>var square2 =</code> <code>square(n);</code>	<code>square2</code>	<code>undefined</code>
<code>var square4 =</code> <code>square(4);</code>	<code>square4</code>	<code>undefined</code>

Memory State After Phase 1:



Phase 2: Code Execution Phase ⚡

Now JavaScript executes the code line by line:

Line 1: `var n = 2;`

- Assigns value `2` to variable `n` (replaces `undefined`)

Line 2-5: `function square(num) {...}`

- Nothing to execute (function already stored in memory)

Line 6: `var square2 = square(n);`

Something Important Happens Here!

When a function is called, JavaScript creates a **new Local Execution Context** (also called Function Execution Context). **Functions are a bit different than any other language** - a new execution context is created altogether:

Memory Creation Phase (for square function):

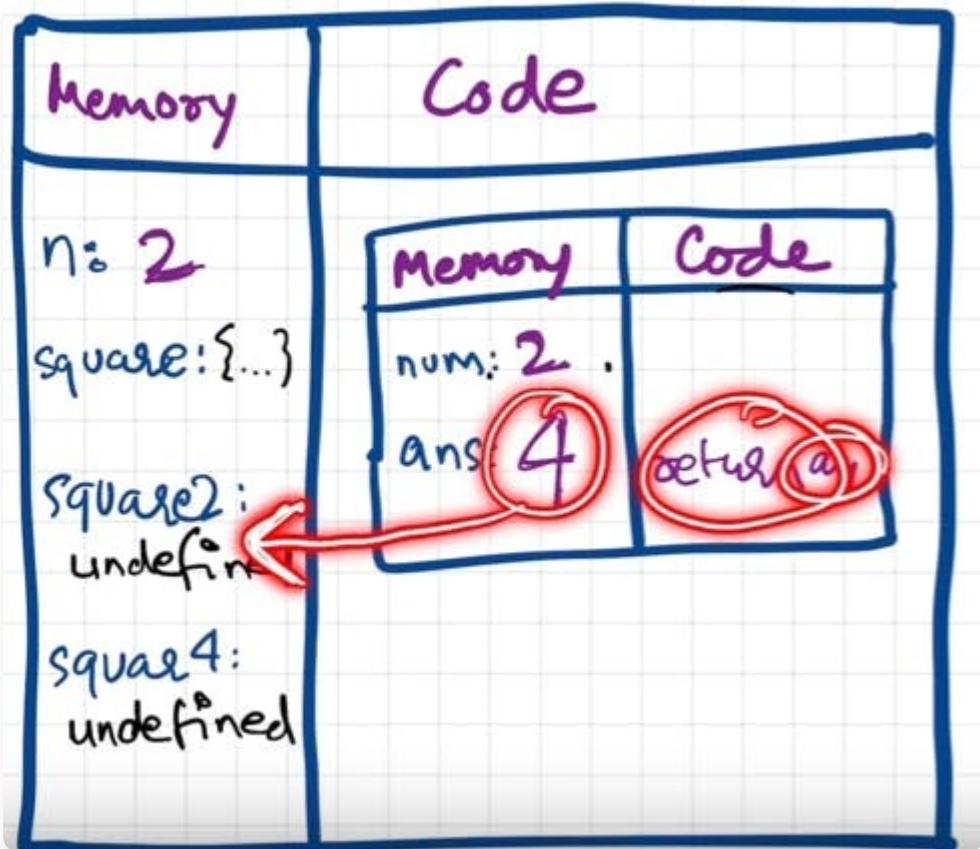
- `num` → `undefined`
- `ans` → `undefined`

Code Execution Phase (for square function):

1. `num` gets value `2` (from parameter `n`)
2. `ans = num * num` → `ans = 2 * 2` → `ans = 4`
3. `return ans` → returns `4` and **destroys** this execution context

Remember: When the `return` keyword is encountered, it returns the control to the called line and also the function execution context is deleted.

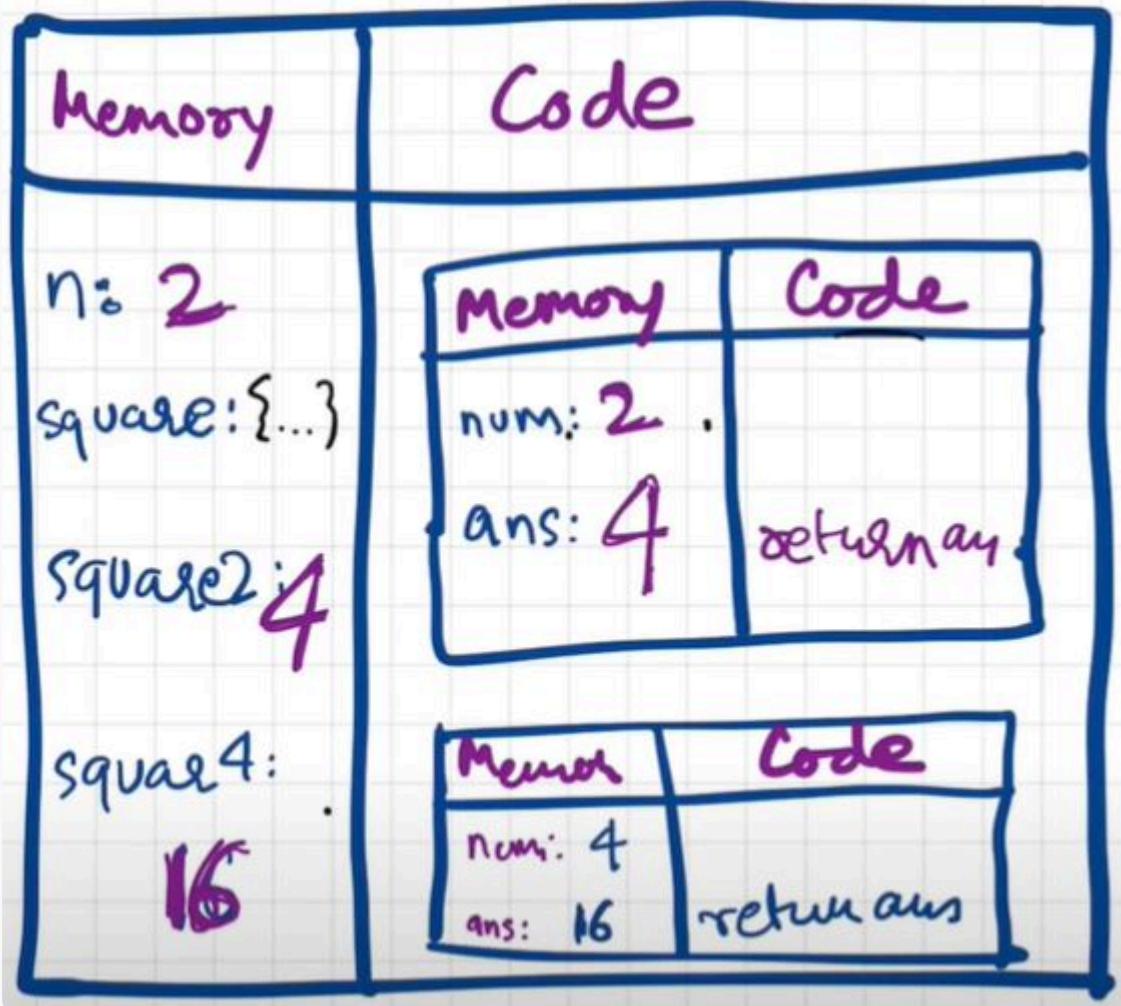
Memory State During Function Execution:



Line 7: `var square4 = square(4);`

- Same process repeats with parameter value 4
- Result: `square4 = 16`

Final Memory State:



Understanding Call Stack

🤔 What is Call Stack?

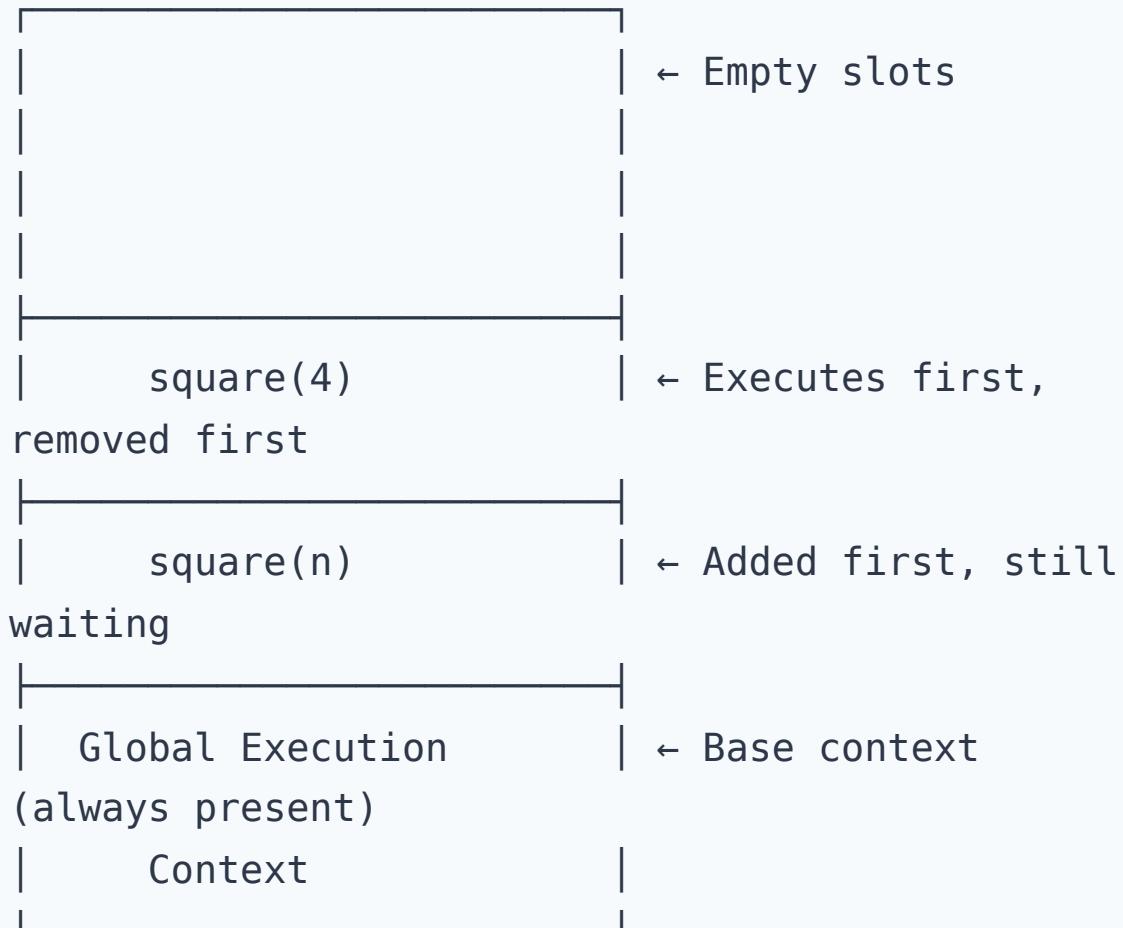
The **Call Stack** is JavaScript's way of managing execution contexts and keeping track of function calls.

⚡ Key Things to Know:

- LIFO (Last In, First Out) principle
- Tracks execution order
- Manages memory allocation and cleanup
- Prevents infinite recursion

How Call Stack Works

Call Stack (LIFO Structure)



Execution Flow:

1. **Global Execution Context** is pushed to stack
2. When `square(n)` is called → **New context** pushed to stack
3. When `square(n)` returns → **Context is popped** from stack
4. When `square(4)` is called → **New context** pushed to stack
5. When `square(4)` returns → **Context is popped** from stack
6. Finally, **Global context** is destroyed when program ends

Why LIFO Matters - Practical Implications:

LIFO (Last In, First Out) is crucial for:

Debugging:

```
function a() {
    b(); // If error occurs in b(), stack trace
  shows: b() → a() → global
}
function b() {
    c(); // If error occurs in c(), stack trace
  shows: c() → b() → a() → global
}
function c() {
    throw new Error("Something broke!");
}
```

Function Return Order:

- Nested functions must complete before their parent functions can continue
- This ensures proper cleanup and variable scope management
- Prevents memory leaks and maintains execution integrity

Memory Management:

- Each function's variables are cleaned up when it's removed from stack
- LIFO ensures proper memory deallocation order
- Parent functions wait for child functions to complete before cleaning up



Other Names for Call Stack

- Program Stack
 - Control Stack
 - Runtime Stack
 - Machine Stack
 - Execution Context Stack
-

Quick Summary

What We Learned:

1. Execution Context Creation

- **Global Execution Context** is created when JS program runs
- Every function call creates a new **Local Execution Context**
- Each context has **2 phases**: Memory Creation → Code Execution

2. Memory Creation Phase

- Variables are allocated memory and assigned `undefined`
- Functions are stored with their complete code
- No code execution happens in this phase

3. Code Execution Phase

- Variables get their actual values
- Functions create new execution contexts when called
- `return` statement destroys the function's execution context

4. Call Stack Management

- **LIFO (Last In, First Out)** structure
- Tracks execution order of all contexts

- Global context stays at bottom, function contexts stack on top
- Contexts are **pushed** when called, **popped** when returned

Quick Memory Aid:

JavaScript Execution = Global Context + Function Contexts

Each Context = Memory Phase + Execution Phase

Call Stack = Context Manager (LIFO)

Return = Context Destruction

Where You'll Use This:

Understanding execution contexts and call stack helps with:

- **Debugging** code execution flow
 - **Understanding** variable scope and hoisting
 - **Preventing** stack overflow errors
 - **Optimizing** function performance
-

Watch the Video

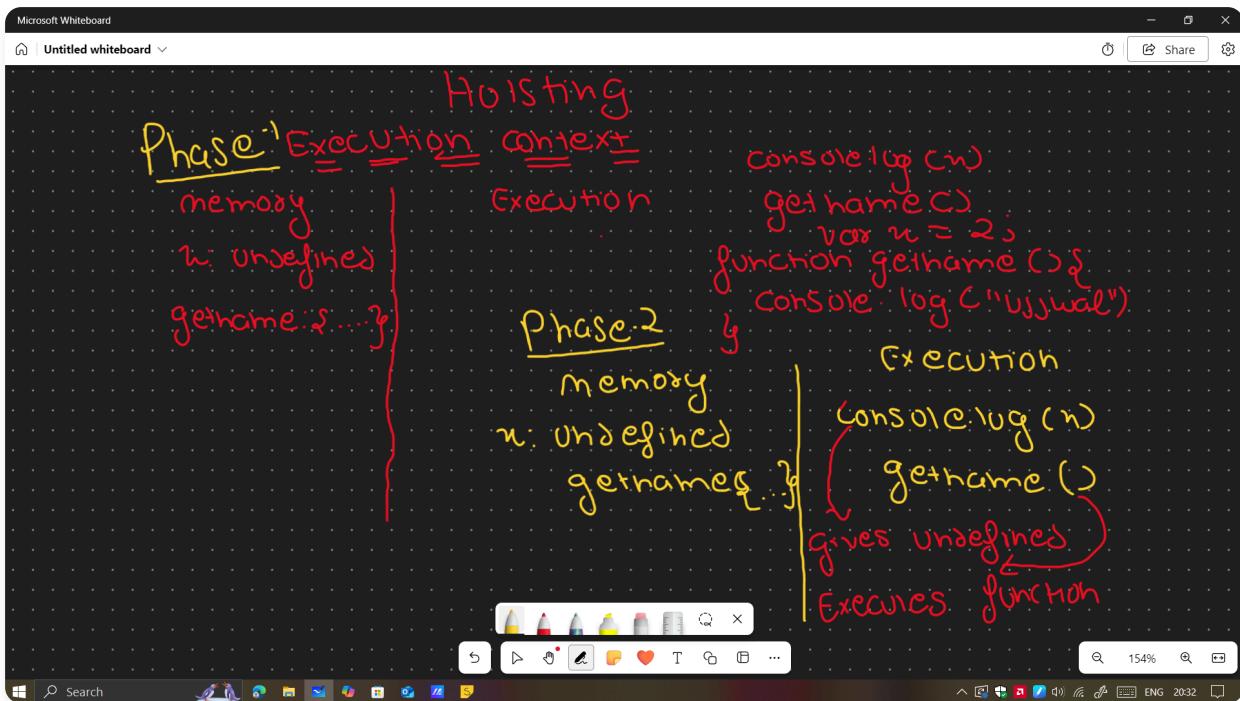


Episode 3: Hoisting in JavaScript (Variables & Functions)

🎯 What You'll Learn

- Understanding Hoisting behavior in JavaScript
 - How variables and functions are treated differently during hoisting
 - Difference between `undefined` and `not defined`
 - Why arrow functions and function expressions behave differently
-

🔍 Let's Start with This Surprising Code



```
getName(); // Namaste Javascript
console.log(x); // undefined
var x = 7;
function getName() {
  console.log("Namaste Javascript");
}
```

🤔 What Just Happened?

In most other languages, this code would throw an error because you're trying to access things before they're created. But JavaScript works differently!

Why this works:

- During the **memory creation phase**, JS assigns `undefined` to variables and stores the complete function code in memory

- During **execution phase**, it executes line by line
- So `getName()` works because the function is already in memory
- `console.log(x)` prints `undefined` because that's what was initially assigned

 **Important:** If we remove `var x = 7;` completely, then we get:

Uncaught ReferenceError: x is not defined



What is Hoisting?

Hoisting is a concept that allows us to access variables and functions even before initializing/assigning values without getting errors. This happens because of the **memory creation phase** of the Execution Context.

CRUCIAL CONCEPT: Memory Phase Happens BEFORE Any Execution

This is extremely important to understand:

```
debugger; // ← Even if debugger is on the
FIRST line
console.log(x); // x is already in memory
with 'undefined'
var x = 5;
function test() {
    return "hello";
}
```

What happens:

1. Code is loaded/parsed
2. Memory Creation Phase completes (happens BEFORE any code execution)
 - `X` → `undefined`
 - `test` → complete function code stored
3. Execution Phase starts
4. Debugger hits and stops execution
5. But memory allocation is ALREADY DONE!

 *Key Insight: Memory Creation Phase is NOT part of execution - it's a preparation phase that happens before the first line of code even starts executing!*

Proof: When debugger stops on line 1, check the browser's Variables/Scope panel - you'll see all variables and functions already exist in memory!

Function Expressions vs Function Declarations

❖ Key Point:

Arrow functions and function expressions are treated like variables during hoisting - they don't get the special treatment that regular function declarations get.

❖ Arrow Functions Don't Get Hoisted Like Regular Functions

When you assign a function to a variable (arrow function or function expression), it will **NOT** be assigned the entire code structure during memory creation phase. It will be assigned as `undefined`.

Arrow Function Example:

```
getName(); // Uncaught TypeError: getName is  
not a function  
console.log(getName); // undefined  
var getName = () => {  
    console.log("Namaste JavaScript");  
};
```

Function Expression Example:

```
getName(); // Uncaught TypeError: getName is  
not a function  
console.log(getName); // undefined  
var getName = function() {  
    console.log("Namaste JavaScript");  
};
```

⌚ When to Use Function Declarations vs Expressions

🤔 Why Does This Difference Matter?

Understanding when to use each type is crucial for writing predictable code:

✓ Use Function Declarations When:

1. You Need Hoisting (Call Before Declaration):

```
// ✓ Works: Function available throughout entire scope
calculateTotal(); // Works fine!

function calculateTotal() {
    return price * quantity;
}
```

2. Creating Main/Core Functions:

```
// ✓ Good for main application functions
function initApp() { /* startup logic */ }
function handleUserLogin() { /* login logic */ }
function processPayment() { /* payment logic */ }
```

3. Recursive Functions:

```
// ✓ Function can call itself by name
function factorial(n) {
    if (n <= 1) return 1;
```

```

    return n * factorial(n - 1); // Self-
reference works
}

```

✓ Use Function Expressions When:

1. Conditional Function Creation:

```

// ✓ Create functions based on conditions
let calculator;
if (advancedMode) {
  calculator = function(a, b) {
    return a * b + Math.pow(a, 2);
  };
} else {
  calculator = function(a, b) {
    return a * b;
};
}

```

2. Callback Functions:

```

// ✓ Perfect for event handlers and
callbacks
button.addEventListener('click', function() {
  console.log('Button clicked!');
});

// ✓ Array methods

```

```
numbers.map(function(num) { return num * 2;  
});
```

3. Avoiding Global Scope Pollution:

```
// ✅ Function expression doesn't pollute  
global scope  
const utils = {  
    format: function(text) { return  
text.toUpperCase(); },  
    validate: function(email) { return  
email.includes('@'); }  
};
```

4. Modules and Encapsulation:

```
// ✅ Creating private functions  
const myModule = (function() {  
    const privateFunction = function() {  
        console.log("This is private");  
    };  
  
    return {  
        publicMethod: function() {  
            privateFunction(); // Can call private  
function  
        }  
    }  
});
```

```
};  
})();
```

Quick Decision Guide:

Need	Use	Why
Call before declaration	Function Declaration	Hoisting works
Conditional creation	Function Expression	More flexible
Callbacks/Event handlers	Function Expression	Cleaner syntax
Main app functions	Function Declaration	More readable
Private functions	Function Expression	Better encapsulation

Memory Allocation Example

Let's see how memory is allocated in the execution context:

```
getName(); // Namaste JavaScript
console.log(x); // Uncaught ReferenceError: x
is not defined
console.log(getName); // f getName(){
console.log("Namaste JavaScript"); }
function getName() {
  console.log("Namaste JavaScript");
}
```

Memory allocation looks like this:

```
{  
  getName: function getName() {  
    console.log("Namaste JavaScript");  
  },  
  // x is not present because it's not  
  declared anywhere  
}
```



Variable Shadowing in Hoisting



What is Variable Shadowing?

Variable Shadowing occurs when a local variable has the same name as a variable in an outer scope, effectively "hiding" or "shadowing" the outer variable.

```
var name = "Global John";  
  
function greetUser() {  
  console.log(name); // undefined (NOT  
  "Global John")  
  var name = "Local Jane";  
  console.log(name); // "Local Jane"  
}
```

```
greetUser();
console.log(name); // "Global John"
```

🤔 Why Does This Happen?

During Memory Creation Phase:

```
// Global Execution Context
{
  name: undefined,
  greetUser: function
}

// greetUser() Execution Context
{
  name: undefined, // ← This shadows the
  global 'name'
}
```

During Execution Phase:

1. `console.log(name)` finds local `name` (which is `undefined`)
2. Local `name` gets assigned `"Local Jane"`
3. `console.log(name)` prints `"Local Jane"`

💡 Key Insights:

- **Hoisting happens per scope** - each execution context has its own memory creation phase

- Local variables shadow global ones even during hoisting
- JavaScript always looks in local scope first

⚠ Common Gotcha:

```
var x = 1;
function test() {
  console.log(x); // undefined (not 1!)
  var x = 2;
}
test();
```

Even though `var x = 2` comes after `console.log(x)`, the hoisting of `var x` in the function scope shadows the global `x`.

⚡ Understanding `undefined` vs `not defined`

Term	Meaning	Example
<code>undefined</code>	Variable exists in memory but no value assigned yet	<code>var x; console.log(x); // undefined</code>
<code>not defined</code>	Variable doesn't exist in memory at all	<code>console.log(y); // ReferenceError: y is not defined</code>



Simple Explanation:

- **undefined** = "Hey, I know this variable exists, but it doesn't have a value yet"
- **not defined** = "Sorry, I've never heard of this variable"



Type Coercion with **undefined** :

```
console.log(undefined + 5); // NaN
console.log(undefined == null); // true
console.log(undefined === null); // false
console.log(typeof undefined); // "undefined"
```

Why This Matters:

- **undefined + 5** = **NaN** because **undefined** can't be converted to a number
- **undefined == null** is **true** (special case in JavaScript)
- **undefined === null** is **false** (different types)
- Always use **typeof** to safely check for **undefined**



Let's See Another Example

```
getName(); // Uncaught TypeError: getName is
not a function
console.log(getName); // undefined
```

```
// Way 1: Arrow Function
var getName = () => {
    console.log("Namaste JavaScript");
};

// Way 2: Function Expression
var getName = function() {
    console.log("Namaste JavaScript");
};

// Both ways behave exactly the same!
```

🎯 Why This Happens:

Arrow functions and function expressions are **treated like variables** during hoisting. They don't get the special treatment that regular function declarations get.

During Memory Creation Phase:

- `getName` gets `undefined` (like any other variable)
- The actual function code is **NOT** stored in memory yet

During Execution Phase:

- When we try to call `getName()`, it's still `undefined`
- Calling `undefined` as a function gives: `TypeError: getName is not a function`



Quick Summary



What We Learned:

1. Hoisting Basics

- Hoisting allows access to variables and functions before they're declared
- Happens because of memory creation phase in execution context

2. Function Declaration vs Function Expression

- **Function Declaration:** Gets complete function code in memory during hoisting
- **Function Expression/Arrow Function:** Gets `undefined` like variables

3. `undefined` vs `not defined`

- **`undefined`:** Variable exists in memory but no value assigned
- **`not defined`:** Variable doesn't exist in memory at all



Quick Memory Aid:

JavaScript Execution Timeline:

1. Code Loading/Parsing
2. Memory Creation Phase (BEFORE any execution)
3. Execution Phase (line by line, synchronous)

Function Declaration = Fully Hoisted (complete code)

Function Expression/Arrow = Partially Hoisted (`undefined`)

Variables = Partially Hoisted (undefined)
Not Declared = Not Hoisted (ReferenceError)

🎯 Where You'll Use This:

Understanding hoisting helps with:

- Debugging unexpected `undefined` values
- Writing better, more predictable code
- Avoiding common JavaScript pitfalls
- Understanding execution flow

🎥 Watch the Video



Episode 4: Functions and Variable Environments

What You'll Learn

- How functions create their own execution contexts
 - Understanding variable environments and scope
 - How the same variable name can have different values in different contexts
 - Call stack behavior with multiple function calls
-

Let's Start with This Interesting Code

```
var x = 1;
a();
b(); // we are calling the functions before
      defining them. This will work properly, as
      seen in Hoisting.
console.log(x); // 1

function a() {
    var x = 10; // localscope because of
    separate execution context
    console.log(x); // 10
}

function b() {
```

```
var x = 100;
console.log(x); // 100
}
```

Actual Output:

```
> 10
> 100
> 1
```

Understanding Variable Environments

Key Concept:

Each function creates its **own execution context** with its **own variable environment**. This means:

- Variables with the same name can exist independently in different functions
- Each `X` is a completely separate variable in its own scope
- Functions don't interfere with each other's variables

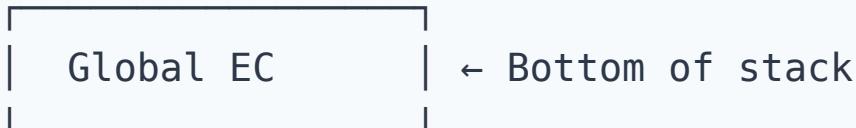
Step-by-Step Execution Flow

Phase 1: Memory Creation (Global Execution Context)

Global Memory:

```
{
  x: undefined,
  a: function a() { var x = 10;
    console.log(x); },
  b: function b() { var x = 100;
    console.log(x); }
}
```

Call Stack:



⚡ Phase 2: Code Execution

Step 1: `var x = 1;`

- Global `x` gets value `1`

Step 2: `a()` is called

- New Local Execution Context created for function `a`
- Call Stack becomes:



Global EC	← Bottom of stack
-----------	-------------------

Memory Creation in `a()` :

```
{
  x: undefined // This is a DIFFERENT x than
global x
}
```

Code Execution in `a()` :

- `var x = 10;` → Local `x` becomes `10`
- `console.log(x);` → Prints `10` (local `x`, not global)
- Function `a()` completes → Local EC is destroyed

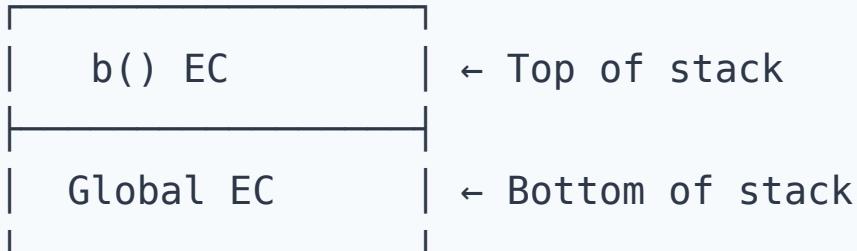
Call Stack after `a()` completes:

Global EC	← Back to global context
-----------	--------------------------

Step 3: `b()` is called

- New Local Execution Context created for function `b`
- Same process as `a()` but with different values

Call Stack during `b()` :



- Local `x` in `b()` becomes `100`
- `console.log(x);` → Prints `100`
- Function `b()` completes → **Local EC is destroyed**

Step 4: `console.log(x);` (Global)

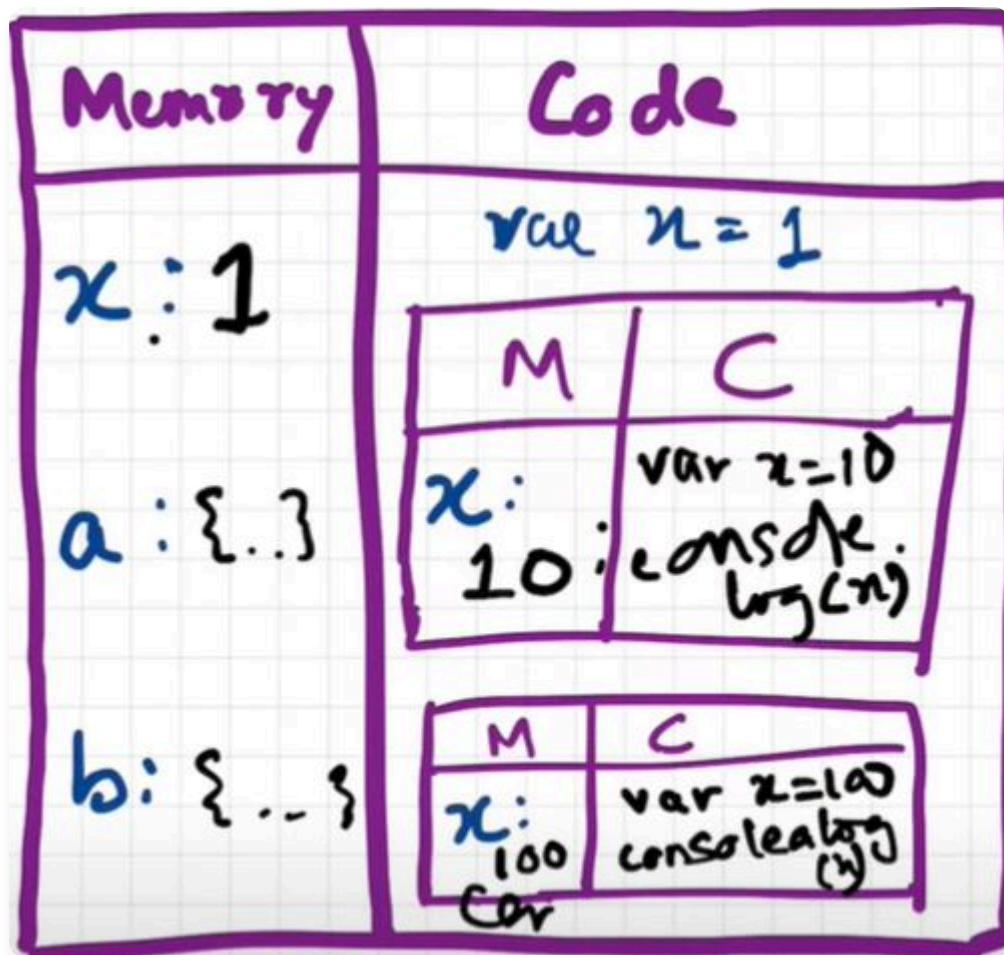
- Back in global context
- Prints global `x` which is still `1`

Step 5: Program Ends

- Global Execution Context is destroyed
- Call stack becomes empty



Visual Representation



⌚ Call Stack Timeline:

Step	Call Stack	Action
1	[Global EC]	Global context created
2	[Global EC, a()]	Function <code>a()</code> called
3	[Global EC]	Function <code>a()</code> completed
4	[Global EC, b()]	Function <code>b()</code> called
5	[Global EC]	Function <code>b()</code> completed
6	[]	Program ends



Quick Summary

What We Learned:

1. Variable Environments

- Each function has its own separate variable environment
- Variables with same names are independent in different scopes
- Local variables don't affect global variables

2. Execution Context Creation

- Every function call creates a new Local Execution Context
- Each context has its own memory space
- Contexts are destroyed when functions complete

3. Call Stack Management

- Functions are added to call stack when called
- Removed from call stack when they complete
- LIFO (Last In, First Out) principle applies

Quick Memory Aid:

Each Function = New Execution Context = New Variable Environment
 Same Variable Name ≠ Same Variable (if in different functions)
 Call Stack = Function Call Manager (LIFO)
 Local Context Destroyed = Local Variables Gone

Where You'll Use This:

Understanding variable environments helps with:

- Debugging scope-related issues
 - Understanding why variables don't interfere with each other
 - Writing cleaner, more predictable functions
 - Avoiding variable naming conflicts
-

🎥 Watch the Video



Episode 5: Shortest JS Program, Window & This Keyword

🎯 What You'll Learn

- What happens when JavaScript runs an empty file
 - Understanding the global `window` object
 - How `this` keyword works at global level
 - How global variables attach to the window object
 - Difference between browser and Node.js global objects
-

🎉 The Shortest JavaScript Program

What is it?

An empty file!

```
// This empty file is a complete JavaScript  
program
```

🚀 What Happens Even in an Empty File?

Even with no code, the JavaScript engine does A LOT:

1. 🚧 Creates Global Execution Context (GEC)

- Memory space allocated
- Execution context set up

2. 🌎 Creates `window` object

- Global object created in global space

- Contains lots of built-in functions and variables
- Accessible from anywhere in the program

3. Creates `this` keyword

- Points to the `window` object at global level
- `this === window` (in browsers)

 *Quick Tip: Anything that is not inside a function is what we call **global space!** and everything that you declare globally get attached to the global object (`window` in case of browsers). and you can access them by `console.log(window.varname)` .*



Understanding the Window Object

What is Window?

The `window` object is the **global object** in browsers. It contains:

- Built-in JavaScript functions (like `setTimeout` , `console.log`)
- Built-in variables
- DOM-related functions and properties
- Your global variables and functions

Key Point:

```
this === window // true (at global level in
browsers)
```

Different Environments:

Environment	Global Object Name
Browser	window
Node.js	global
Web Workers	self



Global Variables and Window Object

Global Variables Get Attached to Window

When you create variables in global scope, they automatically become properties of the window object:

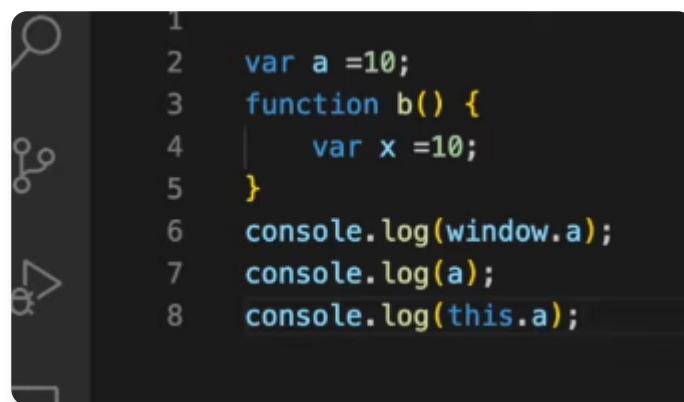
```
var x = 10;
console.log(x);          // 10
console.log(this.x);     // 10
console.log(window.x);   // 10
```

Why All Three Give Same Result?

- `x` → Direct access to global variable
- `this.x` → Access via `this` (which points to `window`)
- `window.x` → Direct access via `window` object

All three are accessing the same variable!

🔥 Important Concept: Default Global Object Reference



```

1
2  var a =10;
3  function b() {
4    var x =10;
5  }
6  console.log(window.a);
7  console.log(a);
8  console.log(this.a);

```

By default, when you use `console.log(varName)` in the JavaScript environment, it **automatically assumes** that you are referring to the `window` (or global object). Since `this` points to the global object at the global level:

```

var name = "JavaScript";

// All of these are exactly the same!
console.log(name);          // "JavaScript" -
                             assumes window.name
console.log(this.name);     // "JavaScript" -
                             this points to window
console.log(window.name); // "JavaScript" -
                           direct window access

```

Key Insight: `console.log(varName)` is essentially
`console.log(window.varName)` in the browser!



How This Works Behind the Scenes

Step-by-Step Process:

1. JavaScript Engine Starts

Creates Global Execution Context

↓

Creates window object (global object)

↓

Creates this keyword (points to window)

↓

Sets up: `this === window`

2. When You Declare Global Variables

```
var x = 10;
```

What happens:

- Variable `X` is created in global scope
- `X` automatically becomes `window.X`
- Can be accessed as `x`, `this.x`, or `window.x`

3. Memory Structure

```

window = {
    // Built-in properties
    console: {...},
    setTimeout: function() {...},
    // ... many more built-ins

    // Your global variables
    x: 10
}

this = window; // this points to window
object

```



Quick Summary

What We Learned:

1. Empty File Behavior

- Even empty file creates Global Execution Context
- JavaScript engine sets up global environment automatically
- Minimum setup includes GEC, window object, and this keyword

2. Window Object

- Global object in browsers (called differently in other environments)
- Contains built-in functions and variables
- Accessible from anywhere in the program

3. This Keyword at Global Level

- `this` points to `window` object in browsers
- `this === window` is true at global level
- Provides another way to access global variables

4. Global Variable Attachment

- Global variables automatically become window properties
- Can be accessed multiple ways: `variable`, `this.variable`, `window.variable`
- All three methods access the same memory location

5. Default Global Object Reference

- `console.log(varName)` automatically assumes `window.varName`
- JavaScript implicitly refers to global object when no context specified
- This is why global variables are so easily accessible

Quick Memory Aid:

```
Empty File = GEC + window + this
Global Variable = window Property
this === window (in browsers)
Global Scope = Window Scope
console.log(varName) =
console.log(window.varName)
```

Where You'll Use This:

Understanding global objects helps with:

- **Debugging** global variable issues
- **Understanding** how JavaScript sets up its environment
- **Working** with different JavaScript environments
- **Avoiding** global namespace pollution

Important: Avoiding Global Namespace Pollution

What is Global Namespace Pollution?

- Adding too many variables to the global scope (window object)
- Can cause naming conflicts with other scripts/libraries
- Makes code harder to debug and maintain

Example of Pollution:

```
//  Bad: Polluting global scope
var userName = "John";
var userAge = 25;
var userEmail = "john@example.com";
var calculateTax = function() { /* logic */ };
var formatDate = function() { /* logic */ };

// Now window has: window.userName,
// window.userAge, etc.
// Risk of conflicts with other scripts!
```

Better Approach:

```
//  Good: Using objects/modules to contain variables  
const App = {  
    user: {  
        name: "John",  
        age: 25,  
        email: "john@example.com"  
    },  
    utils: {  
        calculateTax: function() { /* logic */ },  
        formatDate: function() { /* logic */ }  
    }  
};  
  
// Only one global variable: App
```





Episode 6: Undefined vs Not Defined in JavaScript

🎯 What You'll Learn

- The difference between `undefined` and `not defined`
 - How JavaScript assigns `undefined` during memory allocation
 - Understanding JavaScript as a loosely typed language
 - Best practices for handling `undefined`
-

The Core Difference

Memory Allocation Phase

In the **first phase (memory allocation)**, JavaScript assigns each variable a placeholder called **undefined**.

Quick Comparison

Term	Meaning	When it happens
undefined	Memory allocated, but no value assigned yet	Variable declared but not initialized
not defined	Variable not declared/found in memory allocation phase	Variable never declared but accessed

 *Key Point: Not Defined !== Undefined*



Understanding **undefined**

What is **undefined** ?

- **undefined** is when memory is allocated for the variable, but no value is assigned yet
- It's just a **placeholder** assigned to the variable temporarily until a value is assigned
- It's a **special value** in JavaScript, not an error

🎯 Simple Rule:

*When a variable is **declared** but not assigned a value, its current value is **undefined**. But when the variable itself is **not declared** but called in code, then it is **not defined**.*

🔍 Let's See This in Action

```
console.log(x); // undefined
var x = 25;
console.log(x); // 25
console.log(a); // Uncaught ReferenceError: a
is not defined
```

🧠 What's Happening Here?

Line 1: `console.log(x);` → `undefined`

- Variable `X` is **declared** (on line 2) but not yet **initialized**
- During memory creation phase, `X` got `undefined` as placeholder
- So accessing `X` before assignment gives `undefined`

Line 2: `var x = 25;`

- Now `X` gets the actual value `25`

Line 3: `console.log(x);` → `25`

- `x` now has a real value, so it prints `25`

Line 4: `console.log(a);` → **ReferenceError**

- Variable `a` was **never declared** anywhere
 - JavaScript engine never allocated memory for `a`
 - Result: **Uncaught ReferenceError: a is not defined**
-



JavaScript: The Flexible Language

Loosely Typed / Weakly Typed Language

JavaScript is a **loosely typed** (or **weakly typed**) language, which means:

- **No fixed data types** for variables
- Variables can change types during runtime
- Same variable can hold different types of values



Example of Type Flexibility:

```

var myVariable = 5;          // Number
myVariable = true;           // Boolean
myVariable = 'hello';        // String
myVariable = [1, 2, 3];       // Array
myVariable = {name: 'JS'};    // Object

// All of these are perfectly valid!

```

⚠️ Best Practices with `undefined`

🚫 What NOT to Do:

```
var x = undefined; // ❌ Never do this  
manually!
```

✓ What TO Do:

```
var x; // ✓ Let JavaScript assign undefined  
naturally  
// or  
var x = null; // ✓ Use null if you want to  
explicitly indicate "no value"
```

⌚ Why Avoid Manual `undefined` Assignment?

- You **can** assign `undefined` to any variable manually
- But you should **NEVER** do it
- It's a **bad practice** that leads to:
 - Inconsistencies in code
 - Difficulty in debugging
 - Confusion about whether it's natural hoisting or manual assignment

 **Best Practice:** Let `undefined` happen naturally through JavaScript's hoisting mechanism!

Quick Summary

What We Learned:

1. `undefined` vs `not defined`

- **undefined:** Variable declared but no value assigned (memory allocated)
- **not defined:** Variable never declared (no memory allocated)
- These are completely different concepts!

2. Memory Allocation Behavior

- JavaScript assigns `undefined` as placeholder during memory creation phase
- This happens automatically through hoisting mechanism
- Don't interfere with this natural process

3. JavaScript's Type System

- Loosely typed language - no fixed data types
- Variables can change types during runtime
- Provides flexibility but requires careful handling

4. Best Practices

- Never manually assign `undefined` to variables
- Let JavaScript handle `undefined` naturally

- Use `null` if you need to explicitly indicate "no value"

Quick Memory Aid:

`undefined` = Declared but not assigned
(placeholder)
`not defined` = Never declared (doesn't exist)
Loosely typed = Variables can change types
Manual `undefined` = Bad practice (avoid it!)

Where You'll Use This:

Understanding `undefined` helps with:

- **Debugging** variable initialization issues
 - **Writing** more predictable code
 - **Avoiding** common JavaScript pitfalls
 - **Understanding** hoisting behavior
-

Watch the Video



Episode 7: The Scope Chain, Scope & Lexical Environment

🎯 What You'll Learn

- Understanding Scope and its relationship with Lexical Environment
 - How JavaScript resolves variable access through scope chain
 - The concept of Lexical Environment and its hierarchy
 - How inner functions can access outer function variables
 - Physical location vs accessibility in JavaScript
-



Core Concept

What is Scope?

Scope means where you can access a function or a variable in your code.

second perspective

if a particular variable is in the scope of a function

Scope in JavaScript is directly related to Lexical Environment.



Let's Explore Through Examples

Case 1: Function Accessing Global Variable

```
// CASE 1
function a() {
    console.log(b); // 10
    // Instead of printing undefined it prints
    // 10, So somehow this a function could access
    // the variable b outside the function scope.
}
var b = 10;
a();
```

Case 2: Nested Function Accessing Global Variable

```
// CASE 2
function a() {
  c();
  function c() {
    console.log(b); // 10
  }
}
var b = 10;
a();
```

🔍 Case 3: Local Variable Takes Precedence

```
// CASE 3
function a() {
  c();
  function c() {
    var b = 100;
    console.log(b); // 100
  }
}
var b = 10;
a();
```

🔍 Case 4: Global Can't Access Local Variables

```
// CASE 4
function a() {
    var b = 10;
    c();
    function c() {
        console.log(b); // 10
    }
}
a();
console.log(b); // Error, Not Defined
```



Understanding Each Case



Analysis of Results

Case	Result	Explanation
Case 1	10	Function <code>a</code> can access global variable <code>b</code>
Case 2	10	Nested function <code>c</code> can also access global variable <code>b</code>
Case 3	100	Local variable <code>b</code> takes precedence over global <code>b</code>
Case 4	10 then Error	Function can access parent scope, but global can't access local

Key Insights:

- Inner functions can access outer function variables
 - Local variables take precedence over global ones
 - Global scope cannot access local variables
 - Nested functions can access variables from any parent scope
-



Execution Context Memory Structure

Let's understand Case 4 in terms of execution context:



Call Stack:

```
call_stack = [GEC, a(), c()]
```



Memory Allocation:

```
c() = [
    [lexical environment pointer → a()]
]

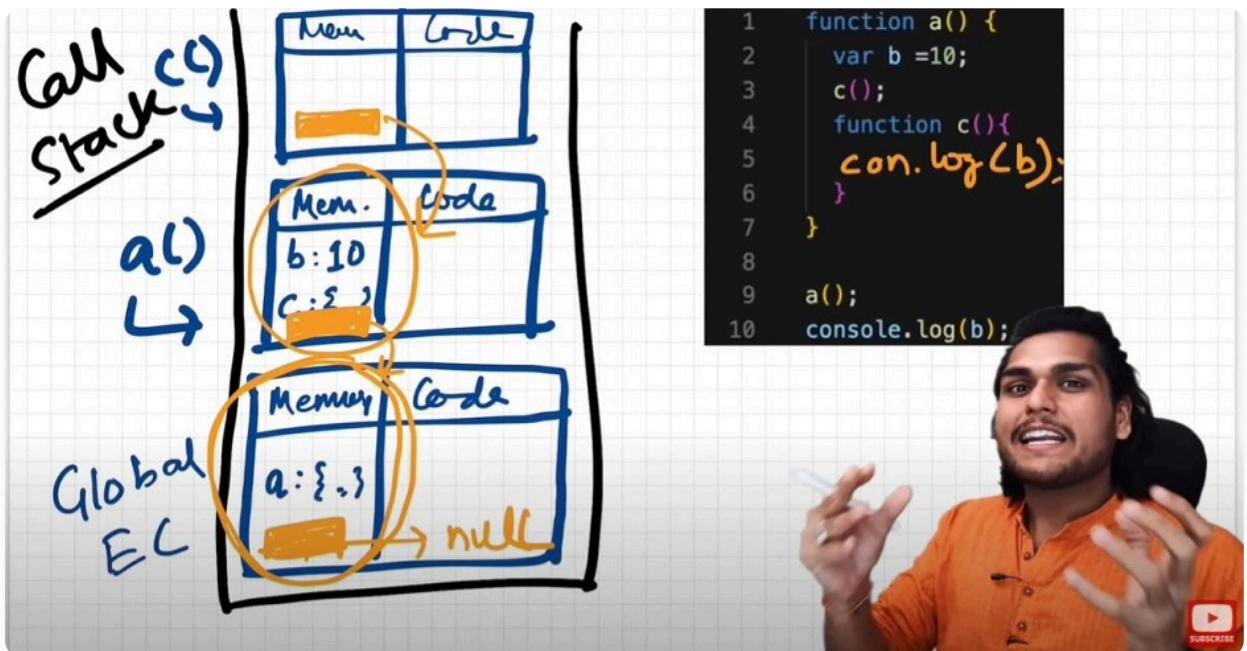
a() = [
    b: 10,
    c: function,
    [lexical environment pointer → GEC]
]

GEC = [
```

```
a: function,  
[lexical environment pointer → null]  
]
```

Visual Representation:

The orange thing in the diagrams is the reference to the lexical environment of its parent.



(index) index.js x

```

1 function a() {
2   var b = 10; b = 10
3   c();
4   function c(){
5     console.log(b);
6   }
7 }
8
9 a();
10

```

{ } Line 3, Column 3 Coverage: n/a

Scope Watch

Paused on breakpoint

Call Stack

- c index.js:5
- a index.js:3
- (anonymous) index.js:9

Local

- b: 10
- c: f c()
- this: Window
- Global

Breakpoints

index.js:5

Console What's New



Understanding Lexical Environment

⚡ Lexical Environment Creation

Whenever an Execution Context is created, a Lexical Environment is also created along with it.

🔗 What is Lexical Environment?

Lexical Environment = Local Memory + Reference to lexical Environment of its Parent

🏢 Understanding "Lexically Inside"

If you look at the code example:

```

function a() {
    function c() {
        // c is lexically inside function a
    }
}
// function a is lexically inside global
scope

```

- Function `c` is lexically inside function `a`
- Function `a` is lexically inside the global scope
- This physical nesting determines the scope chain

Key Points:

- Lexical means "in hierarchy, in order"
- Every Execution Context has its own Lexical Environment
- Lexical Environment is referenced in the local Execution Context's memory space
- Each Lexical Environment has a pointer to its parent's Lexical Environment

The Scope Chain Process:

When JavaScript looks for a variable:

1. Check local memory first
2. If not found, check parent's lexical environment
3. Continue up the chain until found or reach global
4. If not found anywhere, throw `ReferenceError`

This process is called the **Scope Chain** or **Lexical Environment Chain**.



Lexical Scope Explained

🎯 What is Lexical Scope?

Lexical Scope (or **Static Scope**) refers to the accessibility of variables, functions, and objects based on their **physical location** in the source code.

💻 Visual Representation:

```
function a() {  
    function c() {  
        // logic here  
    }  
    c(); // c is lexically inside a  
} // a is lexically inside global execution
```

🏢 Scope Hierarchy:

```
Global {  
    Outer {  
        Inner {  
            // Inner is surrounded by lexical  
            // scope of Outer  
            // Inner can access variables  
            // from Outer and Global  
        }  
    }  
}
```

```
    }  
}
```

🔑 Key Rule:

Physical location in code determines variable accessibility



Quick Summary



What We Learned:

1. Scope and Lexical Environment

- Scope is directly related to Lexical Environment
- Lexical Environment = Local Memory + Parent's Lexical Environment
- Every execution context gets its own Lexical Environment

2. Scope Chain Mechanism

- JavaScript searches for variables through scope chain
- Starts from local scope, moves up to parent scopes
- Process continues until variable is found or global scope is reached

3. Variable Access Rules

- Inner functions can access outer function variables
- Local variables take precedence over global ones
- Global scope cannot access local variables
- Access is determined by physical location in code

4. Lexical Scope Concept

- Based on where variables are declared in the code
- Inner scope has access to outer scope
- Outer scope cannot access inner scope

Quick Memory Aid:

Lexical Environment = Local Memory + Parent's Environment

Scope Chain = Variable lookup process (inner → outer)

Lexical Scope = Physical location determines accessibility

Inner can access Outer, but Outer cannot access Inner

Where You'll Use This:

Understanding scope helps with:

- Debugging variable access issues
- Writing cleaner, more organized code
- Understanding closures and advanced JavaScript concepts
- Avoiding variable naming conflicts



TL;DR

An inner function can access variables from outer functions even if nested deep. In any other case, a function cannot access variables not in its scope.

🎥 Watch the Video



Episode 8: Let & Const in JavaScript, Temporal Dead Zone

🎯 What You'll Learn

- How `let` and `const` declarations are hoisted differently from `var`
- Understanding the Temporal Dead Zone (TDZ)
- Different types of JavaScript errors and when they occur

- Best practices for variable declarations
 - Memory allocation differences between `var`, `let`, and `const`
-



Let's Start with This Surprising Code

```
console.log(a); // ReferenceError: Cannot  
access 'a' before initialization  
console.log(b); // prints undefined as  
expected  
let a = 10;  
console.log(a); // 10  
var b = 15;  
console.log(window.a); // undefined  
console.log(window.b); // 15
```



What Just Happened?

It looks like `let` isn't hoisted, but it actually is! Let's understand the difference:

- Both `a` and `b` are initialized as `undefined` in the hoisting stage
 - But `var b` is stored in the **GLOBAL** memory space
 - `let a` is stored in a **separate memory object** called `script`
 - You can access `a` only **after** it's assigned a value
 - This is why it throws an error when accessed before initialization
-



Complete Hoisting Behavior Explained

 **IMPORTANT:** Let and Const ARE Hoisted (Just Differently!)

All three (`var`, `let`, `const`) are hoisted, but they behave differently:



Memory Allocation Phase (Hoisting):

Variable Type	Memory Allocation	Storage Location	Accessibility
<code>var</code>	undefined	Global scope	 Immediately accessible
<code>let</code>	undefined	Script scope	 Cannot access until assigned
<code>const</code>	undefined	Script scope	 Cannot access until assigned



Step-by-Step Process:

1. Memory Initialization Phase:

```
// During hoisting, JavaScript does this
internally:
var b = undefined;      //  Attached to
global scope (window.b)
let a = undefined;      //  In script scope,
cannot access
const c = undefined;    //  In script scope,
cannot access
```

2. After Assignment:

```
var b = 15;      // ✓ Now b is accessible and
attached to global (window.b = 15)
let a = 10;      // ✓ Now a is accessible but
NOT attached to global (window.a = undefined)
const c = 20;    // ✓ Now c is accessible but
NOT attached to global (window.c = undefined)
```

⌚ Temporal Dead Zone Explained:

TDZ = Time between variable hoisting (undefined assignment) and actual value assignment

```
// Start of TDZ for 'a'
console.log(a); // ✗ ReferenceError: Cannot
access 'a' before initialization
console.log(b); // ✓ undefined (no TDZ for
var)

let a = 10;      // End of TDZ for 'a'
var b = 15;
// From here, both are accessible
```

⌚ Key Insight:

- var: Hoisted + Immediately accessible + Global scope attachment

- **let/const:** Hoisted + Temporal Dead Zone + Script scope (no global attachment)

Practical Examples of the Differences:

```
// Example 1: Global scope attachment
var globalVar = "I'm attached to window";
let scriptVar = "I'm in script scope";
const scriptConst = "I'm also in script
scope";

console.log(window.globalVar); // ✓ "I'm
attached to window"
console.log(window.scriptVar); // ✓
undefined (not attached)
console.log(window.scriptConst); // ✓
undefined (not attached)
```

```
// Example 2: Hoisting behavior comparison
console.log(hoistedVar); // ✓ undefined
console.log(hoistedLet); // ✗
ReferenceError: Cannot access 'hoistedLet'
before initialization
console.log(hoistedConst); // ✗
ReferenceError: Cannot access 'hoistedConst'
before initialization

var hoistedVar = "var value";
```

```
let hoistedLet = "let value";
const hoistedConst = "const value";
```



Understanding Temporal Dead Zone



What is Temporal Dead Zone?

Temporal Dead Zone (TDZ): The time period from when a `let` variable is hoisted until it is initialized with some value.

🎯 In Our Example:

- Any line before `let a = 10` is the **TDZ for variable a**
- During TDZ, the variable exists in memory but cannot be accessed
- Accessing it during TDZ throws a **ReferenceError**



More TDZ Examples:

```
// Example 1: Basic TDZ
console.log(name); // ✗ ReferenceError:
                    Cannot access 'name' before initialization
let name = "John";
console.log(name); // ✓ "John"
```

```
// Example 2: TDZ with function
function checkTDZ() {
    console.log(age); // ✗ ReferenceError:
    Cannot access 'age' before initialization
    let age = 25;
    return age;
}
```

```
// Example 3: No TDZ with var
console.log(city); // ✓ undefined (no error,
just undefined)
var city = "Mumbai";
console.log(city); // ✓ "Mumbai"
```

Global Object Access:

- `window.b` or `this.b` → 15 (var variables attach to global object)
 - `window.a` or `this.a` → `undefined` (let variables don't attach to global object)
-



Understanding JavaScript Error Types

Types of Errors in JavaScript

Error Type	When it occurs	Code execution

Reference Error	Variables in Temporal Dead Zone	Code runs but stops at error
Syntax Error	Invalid syntax detected	Code doesn't run at all
Type Error	Wrong operation on a value	Code runs but stops at error



Quick Examples of Each Error Type:

```
// Reference Error Examples:
console.log(undeclaredVar);      // ✗
ReferenceError: undeclaredVar is not defined
console.log(letVar);            // ✗
ReferenceError: Cannot access 'letVar' before
initialization
let letVar = 10;
```

```
// Syntax Error Examples:
let name = "John";
let name = "Jane";           // ✗
SyntaxError: Identifier 'name' has already
been declared
// Note: Code won't even start running!
```

```
// Type Error Examples:
const obj = { name: "John" };
```

```
obj = { name: "Jane" };           // ✗  
TypeError: Assignment to constant variable
```

```
const func = null;  
func();                      // ✗  
TypeError: func is not a function
```

🔴 Reference Error Examples:

```
let a = 10;  
let a = 100; // SyntaxError: Identifier 'a'  
has already been declared
```

```
let a = 10;  
var a = 100; // SyntaxError: Identifier 'a'  
has already been declared
```

 **Important:** *Syntax Errors don't even let us run a single line of code - they're caught before execution starts!*

 **Variable Declaration Strictness: var → let
→ const**

↳ Let: Stricter than var

```
let a;          // ✓ Declaration without  
initialization is allowed  
a = 10;        // ✓ Assignment later is  
allowed  
console.log(a); // 10
```



More Let Examples:

```
// Example 1: Let allows reassignment  
let score = 100;  
score = 200;          // ✓ Allowed  
console.log(score); // 200
```

```
// Example 2: Let has block scope  
if (true) {  
    let blockVar = "I'm in block";  
    console.log(blockVar); // ✓ "I'm in  
block"  
}  
console.log(blockVar); // ✗ ReferenceError:  
blockVar is not defined
```

```
// Example 3: Let redeclaration not allowed
let username = "user1";
let username = "user2"; // ✗ SyntaxError:
Identifier 'username' has already been
declared
```

Const: Stricter than let

```
const b;      // ✗ SyntaxError: Missing
initializer in const declaration
b = 10;
console.log(b);
```

```
const b = 100; // ✓ Declaration with
initialization
b = 1000;     // ✗ TypeError: Assignment
to constant variable
```

 **Important:** `const` variables must be initialized on the same line where they are declared. You cannot declare a `const` variable and assign a value to it later - this is a fundamental requirement of `const` declarations.



More Const Examples:

```
// Example 1: Const must be initialized
immediately
const PI = 3.14159; // ✅ Correct way
console.log(PI); // 3.14159
```

```
// Example 2: Const with objects (reference
is constant, not content)
const user = { name: "John", age: 30 };
user.age = 31; // ✅ Allowed -
modifying object properties
user.city = "NYC"; // ✅ Allowed - adding
new properties
console.log(user); // { name: "John", age:
31, city: "NYC" }

user = { name: "Jane" }; // ❌ TypeError:
Assignment to constant variable
```

```
// Example 3: Const with arrays (similar
behavior)
const fruits = ["apple", "banana"];
fruits.push("orange"); // ✅ Allowed -
modifying array content
console.log(fruits); // ["apple",
```

```
"banana", "orange"]
```

```
fruits = ["grape"];      // ✗ TypeError:  
Assignment to constant variable
```

Comparison Table:

Feature	var	let	const
Hoisting	✓ (undefined)	✓ (TDZ)	✓ (TDZ)
Global object attachment	✓	✗	✗
Redeclaration	✓	✗	✗
Reassignment	✓	✓	✗
Must initialize	✗	✗	✓



Complete Error Reference Guide

ReferenceError: x is not defined

```
console.log(x); // x was never declared  
anywhere
```

Meaning: Variable was never defined/declared and is being accessed.

ReferenceError: Cannot access 'a' before initialization

```
console.log(a); // a is in TDZ  
let a = 10;
```

Meaning: Variable is declared as `let / const` but not assigned a value yet (in TDZ).

🔴 SyntaxError: Identifier 'a' has already been declared

```
let a = 10;  
let a = 20; // Duplicate declaration
```

Meaning: Trying to redeclare a `let / const` variable. No execution happens.

🔴 SyntaxError: Missing initializer in const declaration

```
const b; // Missing initialization
```

Meaning: `const` must be initialized at the time of declaration.

🔴 TypeError: Assignment to constant variable

```
const b = 100;
```

```
b = 200; // Trying to reassign
```

Meaning: Trying to reassign a `const` variable.



Best Practices for Variable Declarations

🎯 Recommended Approach:

1. Use `const` wherever possible

- For values that won't change
- Makes code more predictable

2. Use `let` when you need to reassign

- For loop counters, conditional assignments
- Better than `var` due to block scope

3. Avoid `var`

- Has confusing scoping rules
- Attaches to global object
- No temporal dead zone protection

4. Declare and initialize variables at the top

- Reduces Temporal Dead Zone window to zero
- Prevents access before initialization errors
- Makes code more readable

 **Example of Good Practice:**

```
// ✓ Good: Declare at top, use const when possible
const API_URL = 'https://api.example.com';
const MAX_RETRIES = 3;
let currentRetries = 0;
let userData;

// ... rest of your code
```

 **Quick Summary**

 **What We Learned:****1. Hoisting Differences**

- `let` and `const` are hoisted but in Temporal Dead Zone
- `var` is hoisted and immediately accessible (as `undefined`)
- Different memory storage: `var` → global, `let/const` → script

2. Temporal Dead Zone

- Time from hoisting until initialization
- Prevents access before assignment
- Throws `ReferenceError` when accessed

3. Error Types

- **ReferenceError:** Variable access issues
- **SyntaxError:** Code structure problems (prevents execution)
- **TypeError:** Wrong operations on values

4. Variable Strictness

- `var` → least strict (most flexible, most error-prone)
- `let` → medium strict (block scoped, reassignable)
- `const` → most strict (block scoped, immutable)

Quick Memory Aid:

```
let/const = Hoisted but in TDZ
var = Hoisted and accessible (undefined)
const = Must initialize immediately
TDZ = Time from hoist to assignment
Strictness: var < let < const
```

Where You'll Use This:

Understanding `let/const` helps with:

- **Writing** more predictable code
- **Debugging** hoisting and scope issues
- **Following** modern JavaScript best practices
- **Avoiding** common variable declaration pitfalls

Watch the Video



Episode 9: Block Scope & Shadowing in JavaScript

🎯 What You'll Learn

- Understanding blocks and compound statements in JavaScript
 - How block scope works with var, let, and const
 - The concept of shadowing and its different behaviors
 - Legal vs illegal shadowing patterns
 - Memory allocation differences in block scope vs global scope
-

Understanding Blocks

What is a Block?

Block (also called **compound statement**) is used to group JavaScript statements together into 1 group. We group them within `{ . . . }` .

```
{  
  var a = 10;  
  let b = 20;  
  const c = 30;  
  // Here let and const are hoisted in Block  
  scope,  
  // While, var is hoisted in Global scope.  
}
```

Why Do We Need Blocks?

Blocks are used where JavaScript expects a **single statement**, but we want to write **multiple statements**.

```
// Example: if statement expects one  
statement  
if (true)  
  console.log("Hello"); // ✓ Single  
statement  
  
// But what if we want multiple statements?  
if (true) { // ✓ Block allows
```

```

multiple statements
  console.log("Hello");
  console.log("World");
let message = "Hi";
}

```

Block Scope Behavior

Memory Allocation in Blocks

Variable Type	Storage Location	Accessibility
<code>var</code>	Global scope	 Accessible outside block
<code>let</code>	Block scope	 Only accessible inside block
<code>const</code>	Block scope	 Only accessible inside block

Block Scope and Accessibility Example

```

{
  var a = 10;
  let b = 20;
  const c = 30;
}
console.log(a); //  10
console.log(b); //  Uncaught
ReferenceError: b is not defined

```

```
console.log(c); // ✗ Uncaught  
ReferenceError: c is not defined
```

🧠 What's Happening in Memory?

Inside the Block:

- `let b` and `const c` are initialized as `undefined` in a separate memory space called "Block"
- `var a` is stored in Global scope
- This is why `let` and `const` are BLOCK SCOPED

Outside the Block:

- `var a` can be accessed anywhere as it's in global scope
 - `let b` and `const c` cannot be accessed outside the block
 - They are destroyed when the block execution is complete
-

🌐 Understanding Shadowing

📚 What is Shadowing?

When a variable inside a block has the **same name** as a variable outside the block, the inner variable **shadows** (hides) the outer variable.

🔍 Shadowing with `var` (the value changes in the global space)

```
var a = 100;
{
  var a = 10; // same name as global var
  let b = 20;
  const c = 30;
  console.log(a); // 10
  console.log(b); // 20
  console.log(c); // 30
}
console.log(a); // 10, instead of the 100 we
were expecting
```

Why Does This Happen?

- Block `var a = 10` modifies the value of global `a` as well
- In memory: only `b` and `c` are in block space
- `a` initially is in global space (`a = 100`)
- When `a = 10` line runs, `a` is not created in block space
- Instead, it replaces 100 with 10 in global space itself



Important: This shadowing behavior happens only for var

Proper Shadowing with `let` and `const`

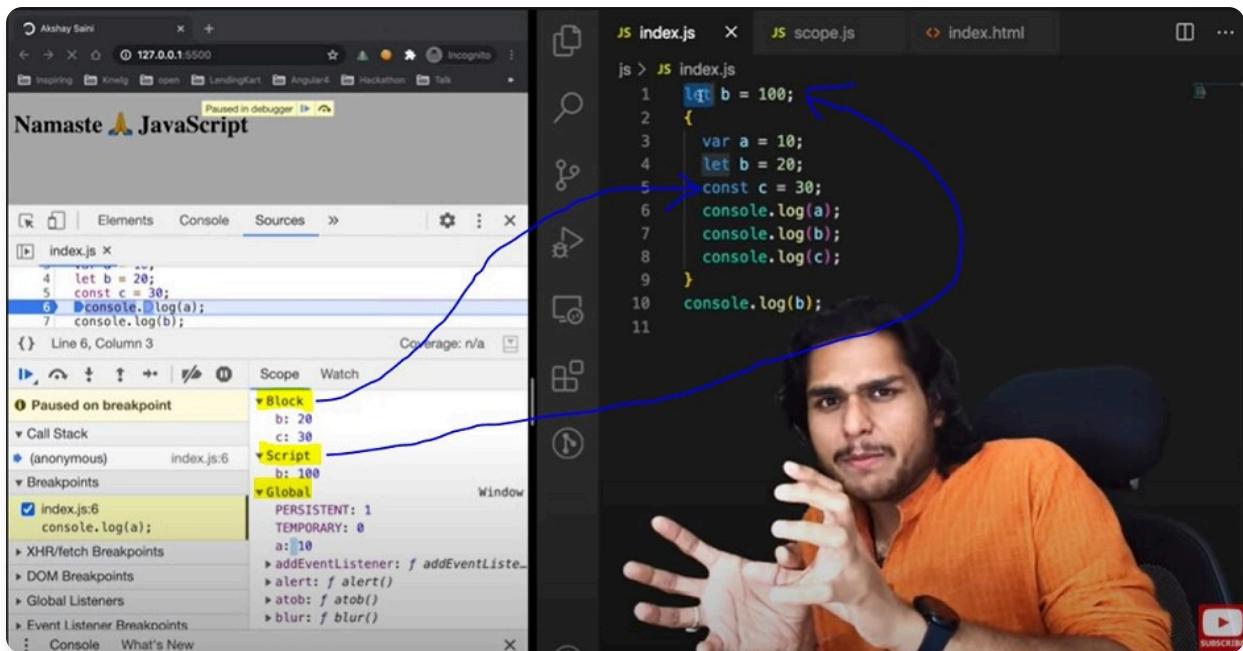
🔍 Let and Const Shadowing Behavior

```
let b = 100;  
{  
    var a = 10;  
    let b = 20;  
    const c = 30;  
    console.log(b); // 20  
}  
console.log(b); // 100
```

💡 *Important: The value of the `let/const` changes for the block/function scope not globally*

🧠 What's Different Here?

- Both `b`'s are in **separate memory spaces**
- Inner `b = 20` is in **Block scope**
- Outer `b = 100` is in **Script scope** (another memory space)
- Same behavior is true for `const` declarations
- **No interference** between inner and outer variables



🎯 Shadowing in Functions

📚 Same Logic Applies to Functions

```
const c = 100;
function x() {
  const c = 10;
  console.log(c); // 10
}
x();
console.log(c); // 100
```

💡 Function Scope Behavior:

- Function creates its own scope
- Inner `const c = 10` shadows outer `const c = 100`

- Both variables exist independently
 - No modification of outer variable
 - *Same as how it behaves in the simple block space* of the let and const
-

Illegal Shadowing

What is Illegal Shadowing?

```
let a = 20;
{
  var a = 20; // ✗ Uncaught SyntaxError:
  Identifier 'a' has already been declared
}
```

Shadowing Rules

Block Scope Shadowing:

Outer Variable	Inner Variable	Result
var	var	✓ Valid (but modifies outer)
var	let	✓ Valid Shadowing
var	const	✓ Valid Shadowing
let	let	✓ Valid Shadowing
let	const	✓ Valid Shadowing
let	var	✗ Illegal Shadowing (scope conflict)

<code>const</code>	<code>const</code>	Valid Shadowing
<code>const</code>	<code>let</code>	Valid Shadowing
<code>const</code>	<code>var</code>	Illegal Shadowing (scope conflict)

Function Scope Shadowing:

Outer Variable	Inner Variable	Result
<code>let</code>	<code>var</code> (in function)	Valid Shadowing
<code>const</code>	<code>var</code> (in function)	Valid Shadowing
All other combinations	Same as block scope rules	

🔍 Why is `let` → `var` Illegal in Block Scope?

```
// ✗ Illegal in block scope
let a = 20;
{
    var a = 20; // ✗ This tries to declare 'a'
    // But 'a' already exists in
    // script scope as let
}
```

Reason: `var` tries to create a variable in global scope, but `let a` already exists in script scope, causing a conflict.

✓ But Valid in Function Scope

```
// ✓ Valid in function scope
let a = 20;
function x() {
    var a = 20; // ✓ Valid - function creates
    separate scope boundary
}
```

Reason: Functions create their own scope, so `var` is confined to function scope and doesn't conflict with outer `let`.

🧠 Why Block Scope vs Function Scope Behaves Differently

🚫 Why NOT in Block Scope?

```
let a = 20;
{
    var a = 20; // ✗ SyntaxError: Identifier
    'a' has already been declared
}
```

What happens in memory:

1. `let a = 20` → Stored in Script scope

2. `var a = 20` → Tries to go to **Global scope**

3. Problem: JavaScript engine sees **TWO declarations of `a`** in the **SAME LEXICAL ENVIRONMENT**

Memory Structure:

```

└── Global Scope
    └── var a (trying to declare here)
└── Script Scope
    └── let a = 20 (already exists here)
└── Block Scope
    └── (empty - var doesn't stay here)
  
```

✗ **Conflict:** Both `let a` (Script) and `var a` (Global) are **visible in the same scope chain** → **SyntaxError**

✓ Why YES in Function Scope?

```

let a = 20;
function someFunction() {
    var a = 20; // ✓ Works perfectly!
}
  
```

What happens in memory:

1. `let a = 20` → Stored in **Script scope**
2. `var a = 20` → Stored in **Function scope (NOT Global!)**
3. **No Problem:** They exist in **DIFFERENT lexical environments**

Memory Structure:

```

    └── Global Scope
        └── (empty)
    └── Script Scope
        └── let a = 20
    └── Function Scope
        └── var a = 20 (completely separate
environment)

```

 **No Conflict:** `let a` (Script) and `var a` (Function) are in **separate scope boundaries**

The Key Difference

Block Scope (`{}`):

- Does NOT create a separate boundary for `var`
- `var` ignores block boundaries and goes to **Global/Function scope**
- Creates **scope pollution** and **identifier conflicts**

Function Scope (`function {}`):

- Creates a **strong boundary** that `var` cannot cross
- `var` respects function boundaries and stays **contained**
- **No scope pollution** - each function is isolated

Real-World Analogy

Think of it like **building boundaries**:

Block Scope = Glass Wall

- `var` can **see through** and **ignore** the glass wall

- Goes to the main house (Global scope)
- Creates **conflicts** with existing residents

Function Scope = Concrete Wall

- `var` cannot cross the concrete wall
- Stays **contained** within the function castle
- **No conflicts** with outside world

Scope Boundary Summary

Scope Type	<code>var</code> Behavior	Conflict Potential
Block <code>{}</code>	Ignores boundary, goes to Global/Function	 High (same lexical environment)
Function <code>function {}</code>	Respects boundary, stays in function	 None (separate lexical environment)

Bottom Line: Function scope creates a true **isolation boundary** that `var` respects, while block scope is just a **suggestion** that `var` ignores! 

Valid Shadowing Examples

```
//  let shadowing let
let a = 20;
{
  let a = 30; // Valid - different scopes
  console.log(a); // 30
```

```

}
console.log(a); // 20

```

```

// ✅ var shadowing with let
var a = 20;
{
  let a = 30; // Valid - let creates block
scope
  console.log(a); // 30
}
console.log(a); // 20

```



Function Scope Exception



Functions Create Their Own Scope

```

let a = 20;
function x() {
  var a = 20; // ✅ Valid - function scope is
separate
}

```



Why is This Valid?

- `var` is function scoped, not block scoped
 - Functions create their own scope boundary
 - No conflict between function scope and outer script scope
 - All scope rules that work in functions are same in arrow functions too
-

Quick Summary

What We Learned:

1. Blocks and Block Scope

- Blocks group statements using `{}`
- `let` and `const` are block scoped
- `var` ignores block scope (function scoped only)

2. Shadowing Behavior

- `var`: Modifies outer variable (same memory space)
- `let/const`: Creates separate memory space (true shadowing)

3. Legal vs Illegal Shadowing

- Cannot shadow `let / const` with `var` in same scope
- Can shadow `var` with `let / const`
- Functions create separate scope (different rules)

4. Memory Spaces

- **Global:** `var` declarations
- **Script:** `let / const` declarations outside blocks
- **Block:** `let / const` declarations inside blocks

- **Function:** All variables inside functions

Quick Memory Aid:

```
Block = {...} groups statements  
let/const = Block scoped (separate memory)  
var = Function scoped (ignores blocks)  
Shadowing: var modifies, let/const separates  
Illegal: let/const → var (scope conflict)
```

Where You'll Use This:

Understanding block scope helps with:

- **Writing** more predictable code
- **Avoiding** variable conflicts
- **Using** proper scoping strategies
- **Debugging** scope-related issues

Watch the Video



Episode 10: Closures in JavaScript

🎯 What You'll Learn

- Understanding closures and how they work in JavaScript
 - How functions bundle with their lexical scope
 - Practical examples of closures in different scenarios
 - Advantages and disadvantages of using closures
 - Real-world applications: Module Pattern, Currying, Memoization
 - Common pitfalls and memory considerations
-

Understanding Closures

What is a Closure?

Closure: A function bundled along with its lexical scope is **closure**.

 **Key Definition:** *A closure is a function that has access to its outer function scope even after the function has returned. Meaning, a closure can remember and access variables and arguments reference of its outer function even after the function has returned.*

How Closures Work

JavaScript has a **lexical scope environment**. If a function needs to access a variable, it first goes to its **local memory**. When it does not find it there, it goes to the **memory of its lexical parent**.



Basic Closure Example

Simple Closure Demonstration

```
function x() {  
    var a = 7;  
    function y() {  
        console.log(a);  
    }  
    return y;  
}  
  
var z = x();
```

```
console.log(z); // value of z is entire code  
of function y.
```

🧠 What's Happening Here?

Step-by-Step Breakdown:

1. Function `x` is called → Creates execution context
2. Variable `a = 7` → Stored in function `x`'s memory
3. Function `y` is defined → Has access to parent scope (`x`)
4. `return y` → Returns not just function `y`, but **entire closure**
5. `var z = x()` → `z` now contains function `y` + its lexical scope
6. When `z()` is called later → Still remembers `var a` inside `x()`

🔑 Key Point:

When `y` is returned, not only is the function returned but the **entire closure** (function `y` + its lexical scope) is returned and put inside `z`. So when `z` is used somewhere else in program, it **still remembers** `var a` inside `x()`.

🧩 Corner Cases in Closures

❓ Question: What will this code print?

```
function x() {  
  var a = 7;  
  function y() {  
    console.log(a);
```

```

}
a = 100;
return y;
}
var z = x();
z(); // What will this print?

```

Think about it...

- Will it print `7` (the initial value)?
- Will it print `100` (the modified value)?
- Will it throw an error?

Answer: It will print `100`

Why does this happen?

When a function is returned along with its **lexical scope**, it points to the **reference** of the variable, not the **value**.

Step-by-Step Explanation:

1. `var a = 7` → Variable `a` is created and assigned value `7`
2. Function `y` is defined → Creates closure with reference to variable `a`
3. `a = 100` → Variable `a`'s value is updated to `100`
4. `return y` → Returns function `y` with its lexical scope (closure)
5. `z()` is called → Accesses the current value of `a`, which is `100`

Key Insight:

Closure stores REFERENCE to variables, not their VALUES

- └─ Variable reference: Points to memory location of 'a'
- └─ Current value: Whatever 'a' holds at execution time
- └─ Result: Always gets the latest value of 'a'



Important Concept:

- Closures maintain live references to outer scope variables
 - Changes to variables are reflected when closure is executed
 - Not a snapshot of values at closure creation time
 - Always accesses current state of referenced variables
-



Complex Closure Example



Multi-Level Closure

```
function z() {
    var b = 900;
    function x() {
        var a = 7;
        function y() {
            console.log(a, b);
        }
        y();
    }
}
```

```

    }
    x();
}
z(); // Output: 7 900

```

Scope Chain in Action:

- Function `y` looks for `a` → Found in parent `X` scope
- Function `y` looks for `b` → Found in grandparent `Z` scope
- **Closure includes** → All accessible variables from scope chain

Important Note: Garbage Collection in Closures

```

// Example: What if 'b' was never used?
function z() {
    var b = 900;           // This variable
exists in scope
    function x() {
        var a = 7;
        function y() {
            console.log(a); // Only 'a' is used,
'b' is NOT used
        }
        return y;          // Function y is
returned as closure
    }
    return x;
}

```

```

}
var closure = z();

```

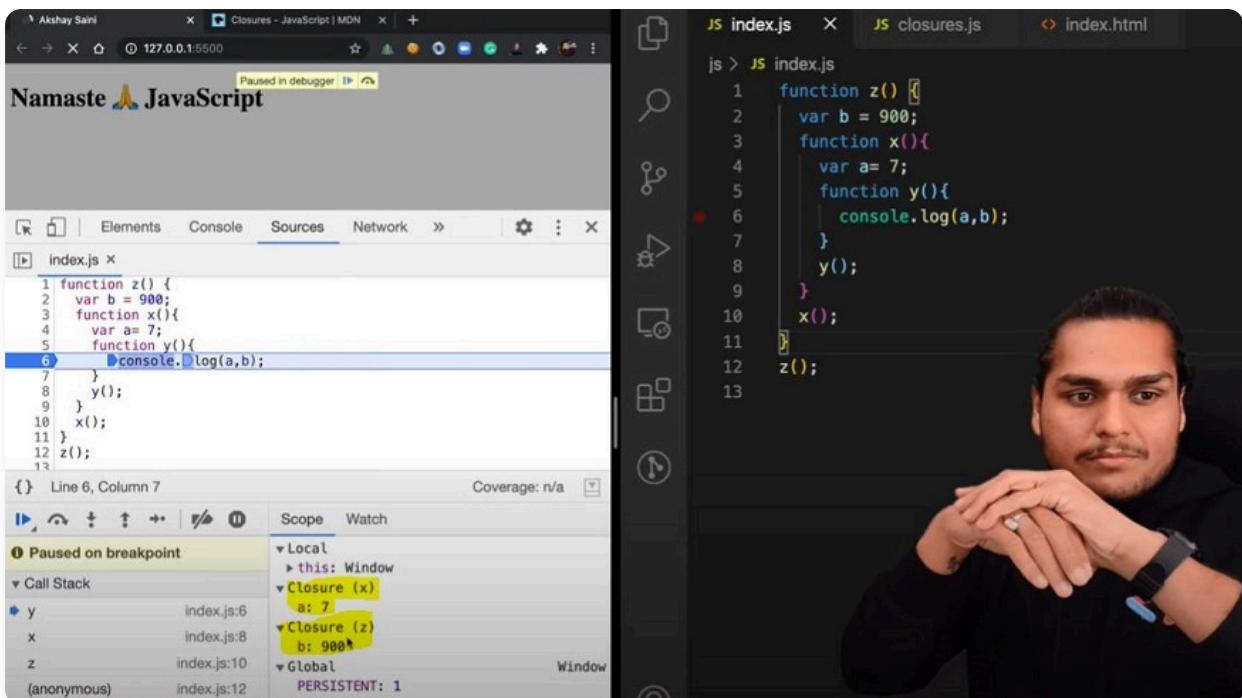
🔑 Key Insight: If variable `b` was never used in the innermost function `y`, JavaScript's smart garbage collection would:

- **Optimize the closure** → Only include variables that are actually referenced
- **Exclude unused variables** → `b` would be garbage collected and **not sent in closure**
- **Save memory** → Closure only carries what it needs

Garbage Collection Rules:

Variable Usage	Included in Closure	Memory Impact
Used in inner function	✓ Yes	Kept in memory
Not used in inner function	✗ No	Garbage collected
Referenced but not accessed	✓ Yes	Kept in memory

💡 Practical Tip: JavaScript engines are smart about closure optimization. Only variables that are **actually referenced** by inner functions are preserved in the closure's lexical environment.



✓ Advantages of Closures

⌚ Real-World Applications

1. 🏗️ Module Design Pattern

The module design pattern allows us to encapsulate related functionality into a single module. It helps organize code, prevent global namespace pollution, and promotes reusability.

```
// auth.js
const authModule = (function () {
  let loggedInUser = null;

  function login(username, password) {
    // Authenticate user logic...
    loggedInUser = username;
  }
})()
```

```

function logout() {
  loggedInUser = null;
}

function getUserInfo() {
  return loggedInUser;
}

return {
  login,
  logout,
  getUserInfo,
};
})();
}

// Usage
authModule.login("john_doe", "secret");
console.log(authModule.getUserInfo()); // 'john_doe'

```

Benefits:

- **Data hiding** → `loggedInUser` is private
- **Controlled access** → Only exposed methods can modify data
- **Global namespace protection** → No pollution

2. 🍔 Currying

Currying is a technique where a function that takes multiple arguments is transformed into a series of functions that take one argument each. It enables partial function application and enhances code flexibility.

```

const calculateTotalPrice = (taxRate) =>
(price) =>
  price + price * (taxRate / 100);

const calculateSalesTax =
calculateTotalPrice(8); // 8% sales tax
const totalPrice = calculateSalesTax(100); //
Price with tax
console.log(totalPrice); // 108

```

Benefits:

- **Reusability** → Create specialized functions
- **Partial application** → Pre-configure parameters
- **Function composition** → Build complex logic from simple parts

3. 🧠 Memoization

Memoization optimizes expensive function calls by caching their results. It's useful for recursive or repetitive computations.

```

function fibonacci(n, memo = {}) {
  if (n in memo) return memo[n];
  if (n <= 1) return n;

  memo[n] = fibonacci(n - 1, memo) +
fibonacci(n - 2, memo);
  return memo[n];
}

```

```
console.log(fibonacci(10)); // 55
```

Benefits:

- **Performance optimization** → Avoid redundant calculations
- **Cache management** → Store and reuse results
- **Recursive optimization** → Dramatically improve recursive functions

4. 🗂️ Data Hiding and Encapsulation

Encapsulation hides the internal details of an object and exposes only necessary methods and properties. It improves code maintainability and security.

```
class Person {
    #name; // Private field

    constructor(name) {
        this.#name = name;
    }

    getName() {
        return this.#name;
    }
}

const person = new Person("Alice");
console.log(person.getName()); // 'Alice'
// console.log(person.#name); // Error:
```

Private field '#name' must be declared in an enclosing class

Benefits:

- **Data protection** → Internal state cannot be directly accessed
- **Controlled interface** → Only specific methods can modify data
- **Security** → Prevents accidental or malicious modifications

5. ⏰ setTimeout Applications

`setTimeout` allows scheduling a function to run after a specified delay. It's commonly used for asynchronous tasks, animations, and event handling.

```
function showMessage(message, delay) {
  setTimeout(() => {
    console.log(message);
  }, delay);
}

showMessage("Hello, world!", 2000); // Display after 2 seconds
```

Benefits:

- **Asynchronous execution** → Non-blocking operations
- **Delayed execution** → Schedule future tasks
- **Event handling** → Manage timed interactions



Disadvantages of Closures



Memory Considerations

1. ⚡ Over Consumption of Memory

- Closures keep references to outer scope variables
- Variables cannot be garbage collected while closure exists
- Multiple closures can hold onto large objects

2. 💀 Memory Leak

- Forgotten closures continue to hold references
- Circular references between closures and DOM elements
- Accumulation of unused but referenced data

3. 🏃 Freeze Browser

- Too many active closures can slow down performance
- Heavy memory usage can cause browser unresponsiveness
- Poor closure management in loops or events



Best Practices to Avoid Issues:

```
// ❌ Potential memory leak
function createHandlers() {
    const largeData = new
    Array(1000000).fill('data');

    return function() {
        console.log('Handler called');
        // largeData is still referenced even if
```

```

not used
};

}

// ✅ Better approach
function createHandlers() {
  const importantData = 'small data';

  return function() {
    console.log('Handler called',
importantData);
    // Only keep what's needed
  };
}

```

Quick Summary

What We Learned:

1. Closure Fundamentals

- Function + lexical scope = Closure
- Remembers outer scope even after parent function returns
- Enables powerful programming patterns

2. Practical Applications

- **Module Pattern** → Encapsulation and data hiding
- **Currying** → Function transformation and reusability
- **Memoization** → Performance optimization

- **Private Variables** → Data protection

3. Memory Management

- Closures can prevent garbage collection
- Be mindful of large data in outer scopes
- Clean up unnecessary references

Quick Memory Aid:

Closure = Function + Lexical Environment

Remembers = Outer scope variables accessible

Applications = Modules, Currying, Memoization

Watch out = Memory leaks and performance

Where You'll Use This:

Understanding closures helps with:

- **Building** reusable and modular code
- **Creating** private variables and methods
- **Implementing** advanced patterns like callbacks
- **Optimizing** performance with memoization

Watch the Video



Episode 11: setTimeout + Closures Interview Question

🎯 What You'll Learn

- How setTimeout works with closures in JavaScript
 - Understanding asynchronous behavior and timing
 - Common interview questions with setTimeout and loops
 - Problem-solving techniques using var vs let
 - Creative solutions using function wrappers for closures
 - Real-world timing and closure scenarios
-



Understanding setTimeout with Closures

Time, tide and JavaScript wait for none.



Basic setTimeout Behavior

```
function x() {
    var i = 1;
    setTimeout(function () {
        console.log(i);
    }, 3000);
    console.log("Namaste Javascript");
}
x();
// Output:
// Namaste Javascript
// 1 // after waiting 3 seconds
```



What's Happening Here?

Expected vs Actual Behavior:

- We expect: JS to wait 3 sec, print 1, then print the string
- What actually happens: JS prints string immediately, waits 3 sec, then prints 1

Step-by-Step Explanation:

1. Function `X` is called → Creates execution context
2. `var i = 1` → Variable declared and assigned

3. **setTimeout** is encountered → Callback function forms a **closure** (remembers reference to **i**)
4. **setTimeout registers callback** → Attaches timer of 3000ms and stores the function
5. JS moves to next line → Does NOT wait, continues execution
6. **console.log("Namaste Javascript")** → Prints immediately
7. After 3000ms → JS takes callback function, puts it into call stack and runs it
8. Closure accesses **i** → Prints the current value of **i** which is **1**

 **Key Insight:**

The function inside setTimeout **forms a closure** (remembers reference to **i**). Wherever the function goes, it carries this reference along with it.

Classic Interview Question

? Question: Print 1 after 1 sec, 2 after 2 sec till 5

This is a **tricky interview question** that catches many developers off-guard.

First Attempt (Naive Approach):

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

```

}
x();
// Output:
// Namaste Javascript
// 6
// 6
// 6
// 6
// 6
// 6

```

🤔 Why does this happen?

The Problem Explained:

1. Loop runs completely → `i` becomes `6` after loop ends
2. **setTimeout callbacks are stored** → All 5 functions are registered with timers
3. **All callbacks share same reference** → They all point to the **same variable**
`i`
4. **When timers expire** → All callbacks access the current value of `i`, which is `6`

🔑 Root Cause:

Closures store REFERENCE to variables, not VALUES

- ─ All 5 `setTimeout` callbacks reference the SAME '`i`'
- ─ Loop completes before any timer expires
- ─ Final value of '`i`' is 6 (loop exit condition)
- ─ Result: All callbacks print 6

Memory Visualization:

Loop Iteration 1: setTimeout(callback1, 1000) →
 callback1 references 'i'
 Loop Iteration 2: setTimeout(callback2, 2000) →
 callback2 references 'i'
 Loop Iteration 3: setTimeout(callback3, 3000) →
 callback3 references 'i'
 Loop Iteration 4: setTimeout(callback4, 4000) →
 callback4 references 'i'
 Loop Iteration 5: setTimeout(callback5, 5000) →
 callback5 references 'i'
 Loop ends: i = 6

After timers expire:

All callbacks access the same 'i' variable →
 Current value is 6

Solution 1: Using `let` (Block Scope)

Let Creates New Variable for Each Iteration

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

```

    }
    console.log("Namaste Javascript");
}
x();
// Output:
// Namaste Javascript
// 1 // after 1 second
// 2 // after 2 seconds
// 3 // after 3 seconds
// 4 // after 4 seconds
// 5 // after 5 seconds

```

Why does `let` work?

Block Scope Magic:

- `let` has block scope → Each iteration creates a new variable `i`
- New closure for each iteration → Each setTimeout callback forms closure with its own copy of `i`
- Separate memory locations → No shared reference problem

Memory Visualization with `let` :

```

Iteration 1: let i = 1 (Block Scope 1) →
callback1 references i1 = 1
Iteration 2: let i = 2 (Block Scope 2) →
callback2 references i2 = 2
Iteration 3: let i = 3 (Block Scope 3) →
callback3 references i3 = 3
Iteration 4: let i = 4 (Block Scope 4) →
callback4 references i4 = 4

```

Iteration 5: let i = 5 (Block Scope 5) →
 callback5 references *i₅* = 5

Each callback has its own 'i' variable with
 preserved value!

Solution 2: Using `var` with Function Wrapper

What if interviewer asks to implement using `var` ?

This is where creativity and deep understanding of closures shines!

```
function x() {
  for (var i = 1; i <= 5; i++) {
    function close(i) {
      setTimeout(function () {
        console.log(i);
      }, i * 1000);
    }
    close(i); // Create new scope for each
  iteration
  }
  console.log("Namaste Javascript");
}
x();
// Output:
// Namaste Javascript
```

```
// 1 // after 1 second
// 2 // after 2 seconds
// 3 // after 3 seconds
// 4 // after 4 seconds
// 5 // after 5 seconds
```

How does this solution work?

Function Scope Creation:

1. **function close(i)** → Creates a new function scope for each iteration
2. Parameter **i** → Creates a new variable **i** in function scope
3. **close(i)** call → Passes current loop value to create new copy of **i**
4. **setTimeout** inside **close** → Forms closure with the local **i** parameter
5. Each callback → Has its own **i** variable with preserved value

Key Insight:

Function parameters create NEW variables in function scope

- └─ **close(1)** creates local **i = 1**
- └─ **close(2)** creates local **i = 2**
- └─ **close(3)** creates local **i = 3**
- └─ **close(4)** creates local **i = 4**
- └─ **close(5)** creates local **i = 5**

Each **setTimeout** callback closes over its own '**i**' parameter!

💡 Alternative Function Wrapper Approaches:

Using Anonymous IIFE:

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

Using Arrow Function Wrapper:

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



Comparison of Solutions

Solution Comparison Table:

Approach	Code Complexity	Memory Usage	Browser Support	Readability
<code>let solution</code>	Simple	Efficient	Modern browsers	Very clear
Function wrapper	Medium	Extra function calls	All browsers	Requires understanding
IIFE approach	Complex	Extra scopes	All browsers	Less readable

When to use which solution?

Use `let` when:

- Working with modern JavaScript (ES6+)
- Code simplicity is priority
- Target audience uses modern browsers

Use function wrapper when:

- Need to support older browsers
- Want to demonstrate deep closure understanding
- Legacy codebase with `var` requirements



Quick Summary



What We Learned:

1. setTimeout + Closures Behavior

- `setTimeout` doesn't block JavaScript execution
- Callbacks form closures with outer scope variables
- Closures store references, not values

2. Common Pitfall

- Loop with `var` creates shared reference problem
- All callbacks access the same variable
- Final loop value is used by all callbacks

3. Solutions

- `let` : Block scope creates new variables per iteration
- **Function wrapper**: Function scope isolates each callback
- **IIFE**: Immediately invoked function creates isolated scope

4. Interview Strategy

- Understand the problem (reference vs value)
- Explain closure behavior clearly
- Provide multiple solution approaches
- Discuss trade-offs of each solution



Quick Memory Aid:

```
setTimeout = Async + Closures  
var in loop = Shared reference problem  
let in loop = Block scope solution  
Function wrapper = Create new scope manually  
Key: Closures reference variables, not values
```

🎯 Where You'll Use This:

Understanding setTimeout + closures helps with:

- Debugging timing-related bugs
 - Building async functionality correctly
 - Solving complex interview questions
 - Creating proper event handling and animations
-

🎥 Watch the Video



Episode 12: Famous Interview Questions ft. Closures

🎯 What You'll Learn

- Essential closure interview questions and their answers
- How closures work with different variable declarations
- Understanding closure scope chain and access patterns
- Data hiding and encapsulation using closures
- Constructor patterns with closures
- Memory management and garbage collection with closures
- Advantages and disadvantages of closures in real scenarios

🔥 Essential Closure Interview Questions

? Q1: What is Closure in JavaScript?

💡 Answer: A function along with reference to its outer environment together forms a closure. Or in other words, a closure is a combination of a function and its lexical scope bundled together.

```
function outer() {  
    var a = 10;  
    function inner() {  
        console.log(a);  
    } // inner forms a closure with outer  
    return inner;  
}  
outer(); // 10  
// First () returns inner function, second ()  
calls inner function
```

🔑 Key Point: The inner function has access to variables in the outer function's scope even after the outer function has finished executing.

? Q2: Will the below code still form a closure?

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

```

    }
    var a = 10;
    return inner;
}
outer(); // 10

```

 Answer: Yes, because inner function forms a closure with its outer environment, so sequence doesn't matter.

 Explanation:

- Due to hoisting, `var a` is available throughout the outer function scope
 - The inner function creates a closure with the entire outer environment
 - Declaration order is irrelevant for closure formation
-

? Q3: Changing var to let, will it make any difference?

```

function outer() {
  let a = 10;
  function inner() {
    console.log(a);
  }
  return inner;
}
outer(); // 10

```

 Answer: It will still behave the same way.

 Explanation:

- Both `var` and `let` are accessible to inner functions via closures
 - The closure mechanism works with both variable declaration types
 - Only difference would be in scoping rules, not closure behavior
-

❓ Q4: Will inner function have access to outer function arguments?

```
function outer(str) {
  let a = 10;
  function inner() {
    console.log(a, str);
  }
  return inner;
}
outer("Hello There")(); // 10 "Hello There"
```

 Answer: Inner function will form closure and will have access to both `a` and `str`.

Explanation:

- Function parameters are part of the function's lexical environment
 - Closures include access to all variables in the outer scope, including parameters
 - Both local variables and parameters are preserved in the closure
-

❓ Q5: In below code, will inner form closure with outest?

```

function outest() {
  var c = 20;
  function outer(str) {
    let a = 10;
    function inner() {
      console.log(a, c, str);
    }
    return inner;
  }
  return outer;
}
outest()["Hello There"](); // 10 20 "Hello
There"

```

 Answer: Yes, inner will have access to all its outer environment.

 Explanation:

- Closures work through the entire scope chain
 - `inner` has access to `outer` scope (`a`, `str`) and `outest` scope (`c`)
 - Multiple levels of nesting create multiple closure relationships
-

? Q6: Output of below code and explanation?

```

function outest() {
  var c = 20;
  function outer(str) {
    let a = 10;

```

```

function inner() {
    console.log(a, c, str);
}
return inner;
}
return outer;
}
let a = 100;
outest()("Hello There")(); // 10 20 "Hello
There"

```

 Answer: Still the same output 10 20 "Hello There".

 Explanation:

- The inner function has reference to inner **a** (value 10), so conflicting names won't matter
- Scope resolution order:** Local scope → Outer function scope → Global scope
- If **a** wasn't found in outer function, it would look in global scope and print **100**
- JavaScript resolves variables through the **scope chain**
- If variable isn't found anywhere, it throws a **ReferenceError**

Scope Chain Visualization:

inner() function looks for 'a':

- Local scope (inner) → Not found
- Outer function scope → Found: let a = 10 ✓
- Global scope → Not needed (already found)

Result: Uses `a = 10` from outer function scope



Q7: Advantages of Closures

🎯 Key Benefits:

- 🏗️ **Module Design Pattern** → Encapsulation and code organization
- ➕ **Currying** → Function transformation and partial application
- 🧠 **Memoization** → Performance optimization through caching
- 🔒 **Data Hiding and Encapsulation** → Private variables and methods
- ⏰ **setTimeout and Callbacks** → Preserving context in asynchronous operations

🔒 Q8: Discuss Data Hiding and Encapsulation

📊 Evolution of Data Protection:

✗ Problem: Without Closures

```
// Global variables are accessible to everyone
var count = 0;
function increment() {
    count++;
}
```

```
// Anyone can access and modify 'count'
directly
// count = 1000; // Oops! Data corruption
```

Issues:

- No data protection
- Global namespace pollution
- Accidental modifications possible

⚠ Attempt 1: Basic Function Wrapping

```
function counter() {
  var count = 0;
  function increment() {
    count++;
  }
  console.log(count); // ReferenceError: count
  is not defined
```

Issues:

- Data is hidden but not accessible
- No way to interact with the counter

✓ Solution 1: Closure with Return Function

```

function counter() {
    var count = 0;
    return function increment() {
        count++;
        console.log(count);
    }
}
var counter1 = counter(); // counter1 has
closure with count
counter1(); // 1
counter1(); // 2

var counter2 = counter(); // Independent
counter
counter2(); // 1 (separate instance)

```

Benefits:

- Data is private and protected
- Controlled access through returned function
- Each instance is independent

Solution 2: Constructor Pattern (Scalable)

```

function Counter() {
    // Constructor function (capitalized first
letter)
    var count = 0;

```

```

this.incrementCounter = function() {
    count++;
    console.log(count);
}

this.decrementCounter = function() {
    count--;
    console.log(count);
}

var counter1 = new Counter(); // Create new
instance
counter1.incrementCounter(); // 1
counter1.incrementCounter(); // 2
counter1.decrementCounter(); // 1

```

Benefits:

- Multiple methods on same private data
- Scalable and extensible
- Object-oriented approach
- Each instance has independent state

🎯 Key Insights:

- **Private variables** → Hidden from external access
- **Controlled interface** → Only exposed methods can modify data
- **Independent instances** → Each closure maintains separate state
- **Scalability** → Easy to add new methods



Q9: Disadvantages of Closures



Memory Consumption Issues

Answer: Overconsumption of memory when using closures, as closed-over variables are not garbage collected until the program expires.

The Problem:

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

var y = a(); // y is a copy of b()
y();
```

What happens in memory:

1. Expected: After `a()` is called, `x` should be garbage collected
2. Reality: Function `b` has closure over `x`, so memory cannot be freed
3. Result: `x` remains in memory as long as `y` exists

Memory Leak Scenarios:

- Creating many closures accumulates memory
- Long-lived closures prevent garbage collection
- Large objects trapped in closures cause memory bloat

🧹 Garbage Collector Explained

🔑 **Definition:** Program in JS engine or browser that frees up unused memory.

Language Comparison:

Language	Garbage Collection
C++, C	Manual (programmer responsibility)
Java	Automatic but configurable
JavaScript	Automatic and implicit

Smart Garbage Collection:

```
function a() {
    var x = 0;      // Used in closure → Kept in
                    // memory
    var z = 10;     // Not used in closure →
                    // Garbage collected
    return function b() {
        console.log(x); // Only x is referenced
    };
}
```

🧠 JavaScript engines (V8, Chrome) are smart:

- **Analyze closure usage** → Only keep referenced variables
- **Remove unused variables** → Z gets garbage collected automatically
- **Optimize memory** → Dead code elimination in closures

 Best Practices to Avoid Memory Issues:

```
// ✗ Potential memory leak
function createHandler() {
    const largeData = new
Array(1000000).fill('data');
    return function() {
        console.log('Handler executed');
        // largeData is kept in memory even if
not used
    };
}

// ✓ Better approach
function createHandler() {
    const neededData = 'small data';
    return function() {
        console.log('Handler executed',
neededData);
        // Only keep what's actually needed
    };
}

// ✓ Clean up when done
let handler = createHandler();
// ... use handler ...
handler = null; // Allow garbage collection
```



Quick Summary



What We Learned:

1. Closure Fundamentals in Interviews

- Definition and basic examples
- Sequence independence in closure formation
- Parameter access through closures
- Multi-level closure relationships

2. Data Hiding Patterns

- Evolution from global variables to private data
- Constructor pattern for scalable solutions
- Independent instance creation

3. Memory Management

- Closure memory retention behavior
- Smart garbage collection in modern engines
- Best practices for memory optimization

4. Interview Strategy

- Understand closure definition clearly
- Explain scope chain resolution
- Demonstrate practical applications
- Discuss trade-offs and limitations



Quick Memory Aid:

Closure = Function + Lexical Environment
Sequence = Doesn't matter (hoisting)
Access = All outer scope variables + parameters
Data Hiding = Private variables + Controlled access
Memory = Smart GC keeps only referenced variables

Interview Preparation Tips:

- Practice explaining closures in simple terms
 - Memorize common patterns and examples
 - Understand memory implications
 - Prepare multiple solution approaches
 - Know real-world applications
-

Watch the Video



Episode 13: First Class Functions ft. Anonymous Functions

🎯 What You'll Learn

- Understanding different ways to create functions in JavaScript
- Difference between function statements and function expressions
- How hoisting affects different function types
- Anonymous functions and their use cases
- Named function expressions and their scope behavior
- Parameters vs arguments distinction

- First-class functions and higher-order functions concept
-

❤️ Functions in JavaScript

Functions are heart ❤️ of JavaScript.

Functions are one of the most powerful and flexible features in JavaScript. They can be created, passed around, and used in multiple ways.

🔍 Function Creation Methods

❓ Q: What is Function Statement?

Function Statement (also called **Function Declaration**) is the traditional way of creating functions.

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

🔑 Key Characteristics:

- **Hoisted completely** → Can be called before declaration
- **Creates named function** → Function has identifier `a`
- **Function declaration** → Standard way to define functions

❓ Q: What is Function Expression?

Function Expression is assigning a function to a variable. Here, **function acts like a value**.

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

b(); // Hello
```

🔑 Key Characteristics:

- **Variable hoisting behavior** → Variable is hoisted, but function is not
- **Anonymous function** → Function itself has no name
- **Assignment operation** → Function is assigned to variable `b`
- **Hoisting behavior** → If you call it before declaration, you'll get an error because the variable will only have the placeholder value `undefined` until the assignment happens

❓ Q: Difference between Function Statement and Function Expression

The major difference between these two lies in **Hoisting**.

```
a(); // "Hello A" ✓ Works fine
b(); // ✗ TypeError: b is not a function

function a() {
    console.log("Hello A");
```

```
}
```

```
var b = function () {
    console.log("Hello B");
};
```

🧠 Why does this happen?

Memory Creation Phase (Hoisting):

Declaration Type	Variable	Function	Accessibility
Function Statement	a: f() {}	Complete function stored	✓ Immediately callable
Function Expression	b: undefined	Not stored yet	✗ Cannot call until assignment

Step-by-Step Explanation:

1. During memory creation phase:

- a → Gets complete function assigned
- b → Gets undefined assigned (like any variable)

2. During code execution:

- a() → Function is available, executes successfully
- b() → undefined is not a function → TypeError

3. After var b = function() {} :

- **b** → Now contains the function and can be called

🔑 Key Insight:

Function Statement = Function + Variable created together

Function Expression = Variable first, Function assigned later

? Q: What is Function Declaration?

Function Declaration is just another name for Function Statement.

```
// Function Declaration (same as Function Statement)
function myFunction() {
    console.log("This is a function declaration");
}
```

 Note: Both terms refer to the same concept - the traditional way of defining functions.

佾 Anonymous Functions

? Q: What is Anonymous Function?

Anonymous Function is a function without a name.

```
function () {
    // This will throw Syntax Error - Function Statement requires function name
}
```

 **Important Points:**

- **Cannot exist alone** → Must be used as a value
- **No identity** → Cannot be called directly
- **Syntax Error** → If used as statement without name

 **Where are Anonymous Functions Used?**

Anonymous functions are used when **functions are used as values**:

```
// ✓ Used in Function Expression
var myFunc = function () {
    console.log("Anonymous function in expression");
};

// ✓ Used as Callback
setTimeout(function () {
    console.log("Anonymous function as callback");
}, 1000);

// ✓ Used in Array Methods
```

```
[1, 2, 3].map(function(x) {
    return x * 2;
});
```



Named Function Expression

? Q: What is Named Function Expression?

Named Function Expression is the same as Function Expression, but the function has a name instead of being anonymous.

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

b(); // ✓ "b called" - Works fine
xyz(); // ✗ ReferenceError: xyz is not
defined
```



Why can't we call xyz() ?

Scope Analysis:

- Function name **xyz** is only available **inside the function scope / local scope**
- Not created in **global scope** → Cannot be accessed from outside
- Only accessible within **function body** → For recursion or self-reference

Example of Internal Access:

```
var factorial = function fact(n) {
    if (n <= 1) return 1;
    return n * fact(n - 1); // ✓ Can call
'fact' inside
};

factorial(5); // ✓ Works: 120
fact(5);     // ✗ ReferenceError: fact is
not defined
```

📊 Function Expression Comparison:

Type	External Call	Internal Call	Use Case
Anonymous	Variable name only	No self-reference	Simple functions
Named	Variable name only	Function name available	Recursion, debugging

🔄 Parameters vs Arguments

❓ Q: What's the difference between Parameters and Arguments?

```
var b = function (param1, param2) {
    // param1, param2 are PARAMETERS
    // (labels/identifiers)
    console.log("b called");
};

b(arg1, arg2); // arg1, arg2 are ARGUMENTS
// (actual values passed)
```

Clear Distinction:

Term	Location	Purpose	Example
Parameters	Function definition	Placeholders/variables	<code>function add(a, b)</code>
Arguments	Function call	Actual values passed	<code>add(5, 10)</code>

Real Example:

```
function greet(name, age) {      // name, age
= Parameters
    console.log(`Hello ${name}, you are ${age}
years old`);

greet("John", 25);              // "John", 25
= Arguments
```

```
greet("Alice", 30); // "Alice",  
30 = Arguments
```



First Class Functions

? Q: What are First Class Functions (aka First Class Citizens)?

First Class Functions means functions can be:

- Passed as arguments to other functions
- Returned from functions
- Assigned to variables
- Stored in data structures

This ability is altogether known as **First Class Function**. It's a programming concept available in some other languages too.



Example 1: Passing Function as Argument

```
var b = function (param1) {  
    console.log(param1); // Prints the entire  
    function  
};  
  
b(function () {  
    console.log("I'm passed as argument");  
});
```

```
// Output: function () { console.log("I'm  
passed as argument"); }
```

🔍 Example 2: Alternative Way

```
var b = function (param1) {  
    console.log(param1);  
};  
  
function xyz() {  
    console.log("Named function passed");  
}  
  
b(xyz); // Same as passing function directly
```

🔍 Example 3: Returning Function from Function

```
var b = function (param1) {  
    return function () {  
        console.log("Function returned from  
another function");  
    };  
};  
  
var returnedFunc = b();  
returnedFunc(); // Calls the returned  
function
```

```
console.log(b()); // Logs the entire returned  
function
```

🎯 Higher-Order Functions

Functions that **operate on other functions** (by taking them as arguments or returning them) are called **Higher-Order Functions**.

```
// Higher-Order Function Example  
function calculator(operation) {  
    return function(a, b) {  
        return operation(a, b);  
    };  
}  
  
function add(x, y) {  
    return x + y;  
}  
  
function multiply(x, y) {  
    return x * y;  
}  
  
var addCalculator = calculator(add);  
var multiplyCalculator =  
calculator(multiply);  
  
console.log(addCalculator(5, 3)); // 8  
console.log(multiplyCalculator(5, 3)); // 15
```



Real-World Applications:

Callback Functions:

```
setTimeout(function() {  
    console.log("Callback executed");  
, 1000);
```

Array Methods:

```
[1, 2, 3, 4, 5]  
.filter(function(x) { return x > 2; })  
.map(function(x) { return x * 2; })  
.forEach(function(x) { console.log(x); });
```

Event Handlers:

```
button.addEventListener('click', function() {  
    console.log('Button clicked');  
});
```



Quick Summary



What We Learned:

1. Function Creation Methods

- **Function Statement/Declaration** → Fully hoisted, callable before declaration
- **Function Expression** → Variable hoisted, function assigned later
- **Named Function Expression** → Function name only available internally

2. Anonymous Functions

- Functions without names
- Must be used as values (expressions, callbacks)
- Cannot exist as standalone statements

3. Parameters vs Arguments

- **Parameters** → Placeholders in function definition
- **Arguments** → Actual values passed during function call

4. First Class Functions

- Functions can be treated like any other value
- Pass as arguments, return from functions, assign to variables
- Enables powerful patterns like Higher-Order Functions

Quick Memory Aid:

Statement = Hoisted completely (callable before)
 Expression = Variable hoisted, function later
 Anonymous = No name, used as values
 Parameters = Definition placeholders
 Arguments = Call-time values

First Class = Functions as values (pass, return, assign)

🎯 Where You'll Use This:

Understanding function types helps with:

- Writing flexible and reusable code
- Understanding hoisting and scope behavior
- Using callbacks and event handlers effectively
- Building higher-order functions and functional patterns

🎥 Watch the Video



Episode 14: Callback Functions in JS ft. Event Listeners



What You'll Learn

- Understanding callback functions and their role in JavaScript
 - How callbacks enable asynchronous programming in synchronous JavaScript
 - Event listeners and their implementation with callbacks
 - Data encapsulation using closures in event handlers
 - Memory management with event listeners
 - Garbage collection and removeEventListeners best practices
-



Understanding Callback Functions



What are Callback Functions?

Callback Functions leverage the fact that functions are **first-class citizens** in JavaScript. You can take a function A and pass it to another function B. Here, A is a **callback function**. Basically, you are giving access to function B to call function A.



Key Insight: Callback functions give us access to the whole Asynchronous world in a Synchronous world.



Basic Callback Example

```
setTimeout(function () {
  console.log("Timer");
}, 1000);
// First argument is callback function,
second is timer
```

What happens here:

- `setTimeout` takes a **callback function** as first parameter
 - After 1000ms, JavaScript **calls back** the function
 - This enables **asynchronous behavior** in synchronous JavaScript
-

JavaScript: Synchronous to Asynchronous

Fundamental Truth:

JavaScript is a synchronous and single-threaded language. But due to callbacks, we can do async things in JS.

Execution Order Example

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

function x(y) {
```

```

console.log("x");
y();
}

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

// Output:
// x
// y
// timer (after 5 seconds)

```

Call Stack Analysis:

Step-by-Step Execution:

1. **setTimeout** is called → Callback registered, timer starts
2. **x(function y() {...})** is called → **X** enters call stack
3. Inside **X** : **console.log("x")** → Prints "x"
4. Inside **X** : **y()** is called → **y** enters call stack
5. Inside **y** : **console.log("y")** → Prints "y"
6. **y** completes → Exits call stack
7. **X** completes → Exits call stack
8. Call stack is empty → Main thread free
9. After 5 seconds → setTimeout callback enters call stack
10. **Callback executes** → Prints "timer"

Critical Understanding:

All functions execute through the call stack
 If any operation blocks the call stack = Blocking the main thread
 Main thread blocked = Entire application freezes

⚠️ Never Block the Main Thread

```
// ✗ Bad Example (Blocking)
function blockingOperation() {
  // Suppose this takes 30 seconds
  for (let i = 0; i < 10000000000; i++) {
    // Heavy computation
  }
}

blockingOperation(); // This will freeze the
browser for 30 seconds
console.log("This won't run until
blockingOperation finishes");
```

```
// ✓ Good Example (Non-blocking)
function nonBlockingOperation() {
  setTimeout(() => {
    // Heavy computation moved to async
    context
    for (let i = 0; i < 10000000000; i++) {
      // Heavy computation
    }
  });
}
```

```

        }
        console.log("Heavy computation done");
    }, 0);
}

nonBlockingOperation();
console.log("This runs immediately"); // Runs
immediately

```

 **Best Practice:** Always use `async` for functions that take time (setTimeout, API calls, file operations, etc.)

Advanced Callback Example

Sequential Execution with Callbacks

```

// Example: Print strings in order with
// random delays
function printStr(str, cb) {
    setTimeout(() => {
        console.log(str);
        cb();
    }, Math.floor(Math.random() * 100) + 1);
}

function printAll() {
    printStr("A", () => {
        printStr("B", () => {

```

```

    printStr("C", () => {
      console.log("All done!");
    });
  });
}

printAll();
// Output: A B C All done! (always in this
order)

```

Why does this work?

- Each `printStr` waits for a **random delay** (1-100ms)
- **Callbacks ensure order** → Next function only runs after previous completes
- **Nested structure** maintains sequence despite random timing

Callback Hell Preview:

```

// This pattern can lead to "Callback Hell"
or "Pyramid of Doom"
getData(function(a) {
  getMoreData(a, function(b) {
    getEvenMoreData(b, function(c) {
      getYetMoreData(c, function(d) {
        // ... and so on
      });
    });
  });
}

```

```
});  
});
```

Event Listeners

What are Event Listeners?

Event listeners are functions that **wait for specific events** to occur (click, hover, scroll, etc.) and then execute **callback functions** in response.

Basic Event Listener Setup

```
<!-- index.html -->  
<button id="clickMe">Click Me!</button>
```

```
// index.js  
document.getElementById("clickMe").addEventListener("click", xyz);  
function xyz() {  
    // When click event occurs, this callback  
    // function (xyz) is called into callstack  
    console.log("Button clicked");  
};
```

How Event Listeners Work:

1. Event listener registered → Browser watches for click events
 2. User clicks button → Browser detects the event
 3. Callback function queued → xyz function added to event queue
 4. Call stack empty → Event loop moves callback to call stack
 5. Callback executes → "Button clicked" is printed
-

1
2
3
4

Implementing Counter with Event Listeners

✗ Approach 1: Global Variable (Not Recommended)

```
let count = 0; // ✗ Global variable - anyone
can modify it

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

// Problem: count can be modified from
anywhere
count = 1000; // Oops! Counter corrupted
```

Issues:

- **Global pollution** → Variable accessible everywhere
- **No data protection** → Any code can modify count

- Accidental modifications → Easy to corrupt state

✓ Approach 2: Closures for Data Abstraction

```

function attachEventListener() {
    let count = 0; // ✓ Private variable
protected by closure

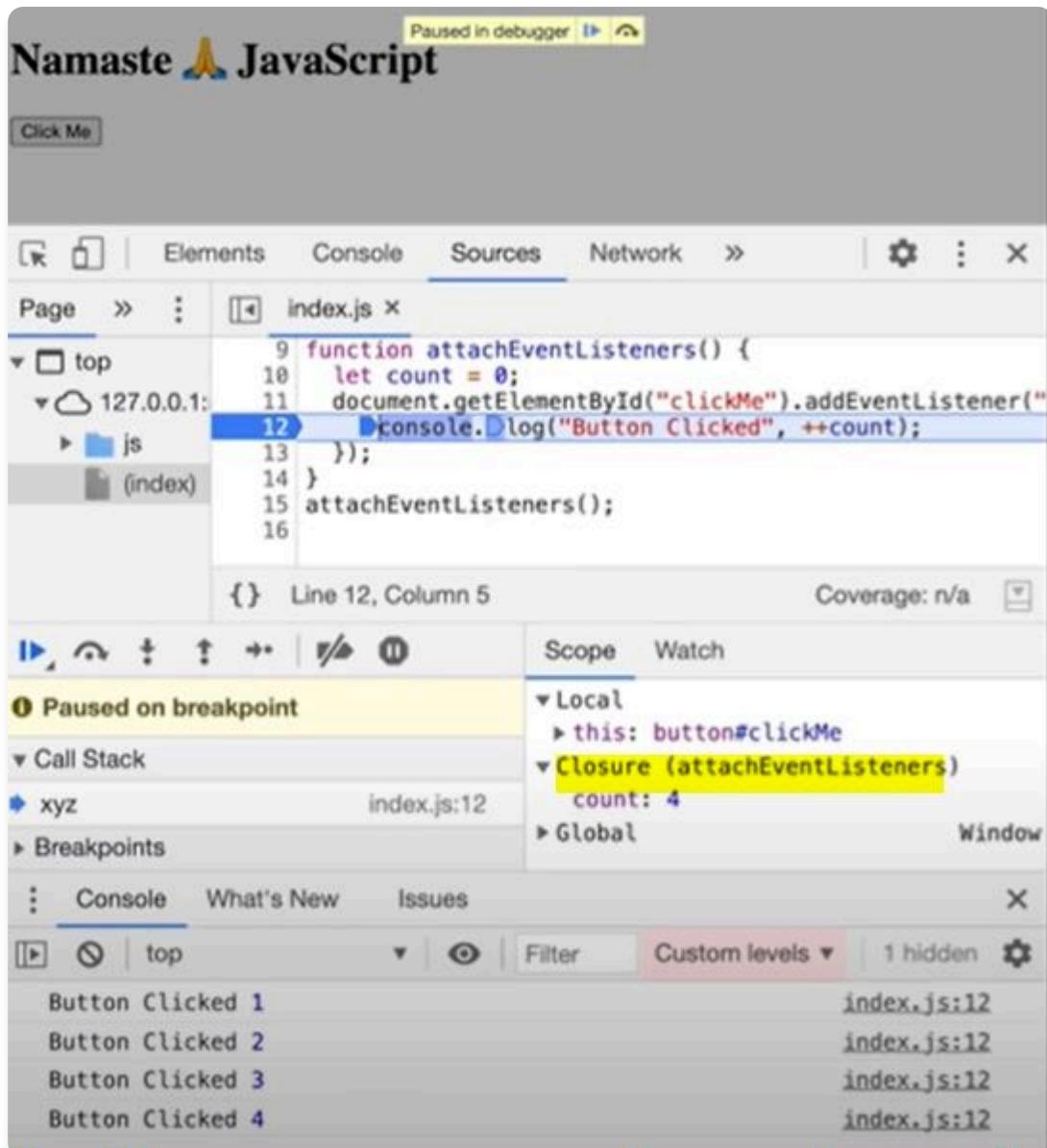
document.getElementById("clickMe").addEventListener("click", xyz);
function xyz() {
    console.log("Button clicked", ++count);
    // Callback function forms closure with
outer scope (count)
})
}

attachEventListener();

```

Benefits:

- Data encapsulation → `count` is private and protected
- Controlled access → Only the event handler can modify count
- No global pollution → Clean global namespace
- Each instance independent → Multiple counters possible



🔍 Multiple Independent Counters:

```

function createCounter(buttonId) {
  let count = 0;

  document.getElementById(buttonId).addEventListener("click", () => {
    console.log(`#${buttonId} clicked ${++count} times`);
  });
}

```

```

        }
      }

      createCounter("button1"); // Independent
      counter 1
      createCounter("button2"); // Independent
      counter 2
      createCounter("button3"); // Independent
      counter 3
    
```

Garbage Collection and removeEventListeners

Memory Management Issues

Event listeners are heavy because they form **closures**. This creates important memory considerations:

```

function attachEventListener() {
  let count = 0;
  const largeData = new
  Array(1000000).fill("data"); // Large object

  document.getElementById("clickMe").addEventListener("click", xyz);
}

function xyz() {
  console.log("Button clicked", ++count);
  // Closure keeps reference to both count
  AND largeData
}

```

```
});  
}
```

The Memory Problem:

Why Memory Isn't Freed:

1. Event listener exists → Callback function is referenced
2. Callback has closure → References outer scope variables
3. Garbage collector can't clean → Variables still "in use"
4. Memory accumulates → Even when call stack is empty

Real-World Impact:

```
//  This can cause memory leaks
for (let i = 0; i < 1000; i++) {
  document.querySelectorAll('.button')
[i].addEventListener('click', function() {
  const heavyData = processLargeDataset();
  // Heavy computation
  console.log('Button', i, 'clicked');
});
}
// 1000 event listeners, each holding heavy
data in memory
```

Best Practices for Memory Management

1. Remove Event Listeners When Not Needed:

```
function setupTemporaryListener() {  
    const button =  
        document.getElementById("clickMe");  
  
    function handleClick() {  
        console.log("Button clicked");  
  
        // Remove listener after first click  
        button.removeEventListener("click",  
            handleClick);  
    }  
  
    button.addEventListener("click",  
        handleClick);  
}
```

2. Clean Up in Component Lifecycle:

```
class ButtonComponent {  
    constructor(buttonId) {  
        this.button =  
            document.getElementById(buttonId);  
        this.handleClick =  
            this.handleClick.bind(this);  
        this.setupEventListener();  
    }  
  
    handleClick() {  
        console.log("Button clicked");  
    }
```

```

}

setupEventListener() {
    this.button.addEventListener("click",
this.handleClick);
}

cleanup() {
    // Always remove event listeners when
component is destroyed
    this.button.removeEventListener("click",
this.handleClick);
}
}

const component = new
ButtonComponent("myButton");
// Later, when component is no longer needed:
component.cleanup();

```

3. Use AbortController (Modern Approach):

```

function setupEventListenerWithAbort() {
    const controller = new AbortController();

    document.getElementById("clickMe").addEventListener(
        function() {
            console.log("Button clicked");
        }, { signal: controller.signal });
}

```

```
// Later, remove all listeners associated
with this controller
setTimeout(() => {
  controller.abort(); // Removes all event
listeners
}, 10000);
}
```

Performance Impact

Multiple event listeners can severely impact performance:

```
// ❌ Performance killer
document.addEventListener('scroll',
heavyScrollHandler);
document.addEventListener('mousemove',
heavyMouseHandler);
document.addEventListener('resize',
heavyResizeHandler);
// Multiple heavy event listeners running
simultaneously
```

Issues:

- **onClick, onHover, onScroll** all in a page can slow it down heavily
- Each listener consumes memory through closures
- Frequent event firing can block main thread



Quick Summary



What We Learned:

1. Callback Functions

- Enable asynchronous programming in synchronous JavaScript
- Functions passed as arguments to other functions
- Bridge between sync and async worlds

2. Call Stack and Threading

- JavaScript is single-threaded with one call stack
- Never block the main thread with heavy operations
- Use async patterns for time-consuming tasks

3. Event Listeners

- Respond to user interactions through callback functions
- Form closures with outer scope for data access
- Enable interactive web applications

4. Memory Management

- Event listeners can cause memory leaks through closures
- Remove listeners when no longer needed
- Use modern patterns like AbortController for cleanup



Quick Memory Aid:

Callback = Function passed to another function
Async = Non-blocking, don't freeze main thread
Event Listener = Wait for events, call callbacks

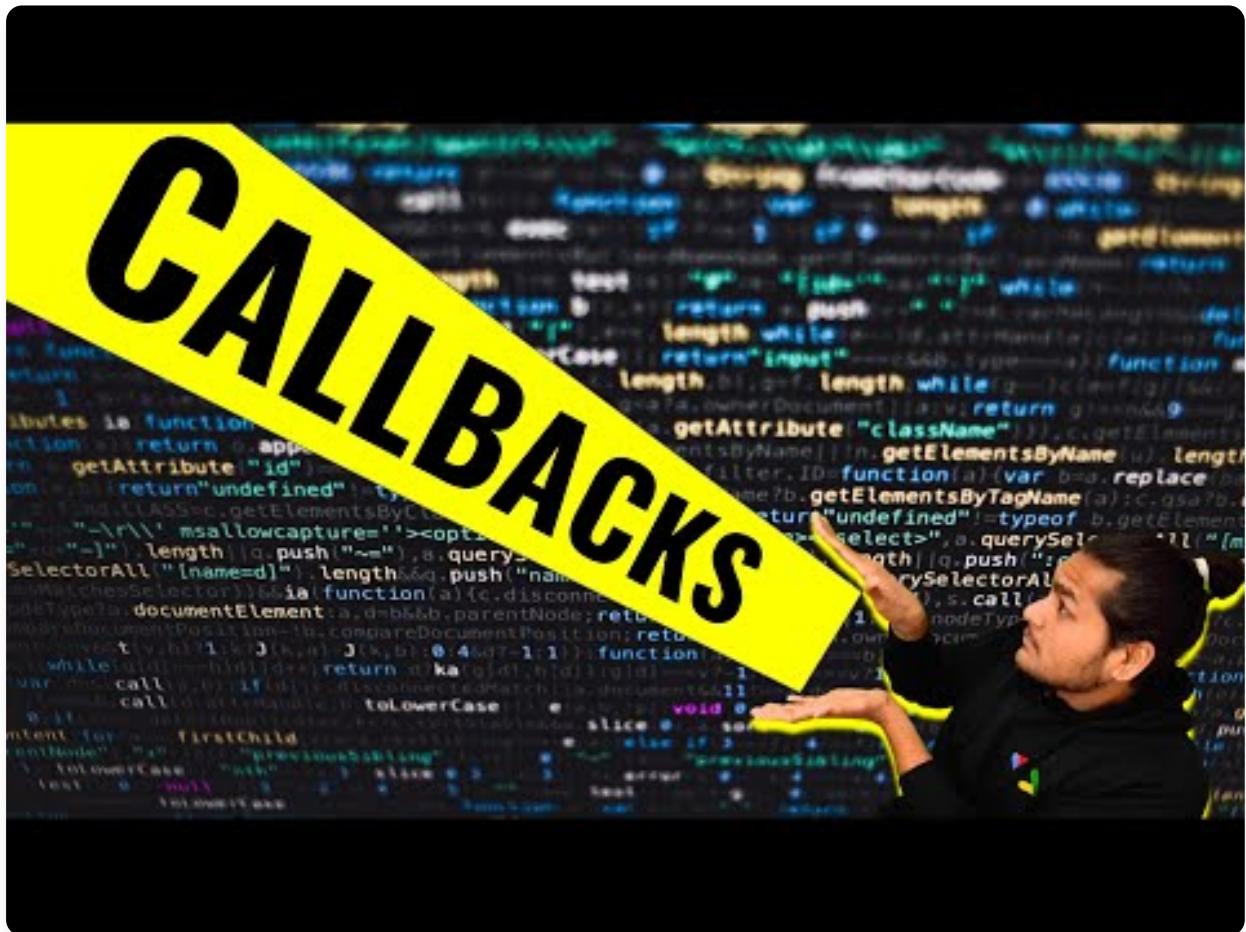
Closure = Event handlers remember outer variables
Cleanup = Remove listeners to prevent memory leaks

⌚ Where You'll Use This:

Understanding callbacks and event listeners helps with:

- **Building** interactive user interfaces
 - **Managing** asynchronous operations
 - **Creating** responsive web applications
 - **Optimizing** memory usage and performance
-

🎥 Watch the Video



Episode 15: Asynchronous JavaScript & EVENT LOOP from scratch

What You'll Learn

- How JavaScript handles asynchronous operations despite being single-threaded
 - Understanding the Browser's Web APIs and their role
 - The Event Loop mechanism and how it coordinates execution
 - Callback Queue vs Microtask Queue priority system
 - How fetch, setTimeout, and DOM events work behind the scenes
 - Common pitfalls and timing issues in asynchronous JavaScript
-

JavaScript's Asynchronous Foundation

 **Key Truth:** Call stack will execute any execution context which enters it.
Time, tide and JS waits for none. **TLDR:** Call stack has no timer.

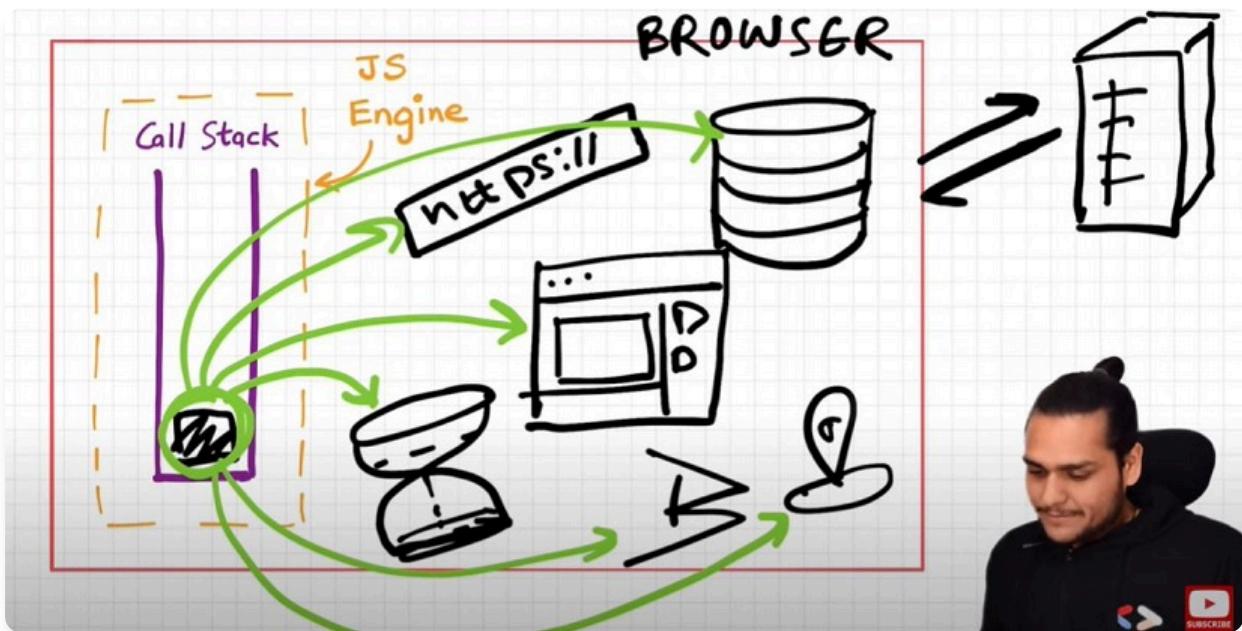
The Big Picture

JavaScript Engine alone is quite limited. The **Browser** provides the real superpowers:

Browser Components:

- └── JavaScript Engine
 - └── Call Stack (Global & Local Execution Contexts)
 - └── Web APIs (Browser's Superpowers)
 - └── Timer (setTimeout, setInterval)
 - └── DOM APIs (document, getElementById, etc.)
 - └── Network (fetch, XMLHttpRequest)
 - └── Storage (localStorage, sessionStorage)
 - └── Location & History
 - └── Many more...
- └── Event Loop System
 - └── Callback Queue
 - └── Microtask Queue

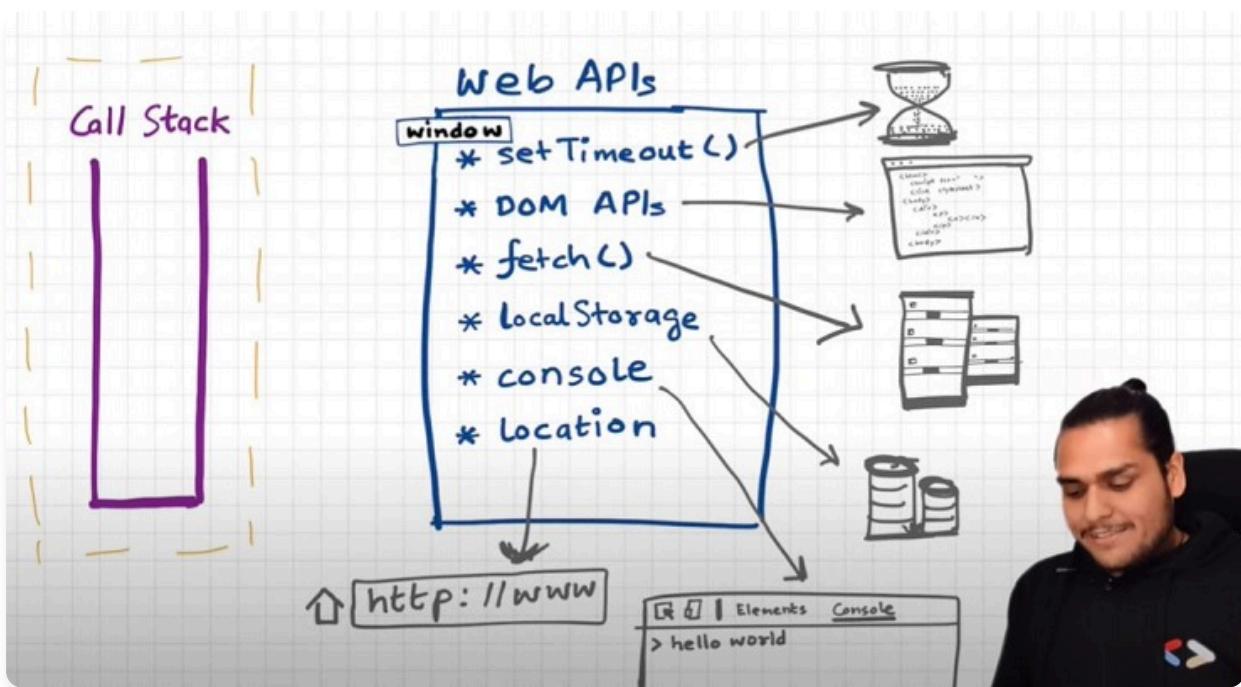
 **Connection:** JavaScript needs some way to connect the call stack with all these browser superpowers. This is done using **Web APIs**.



🌐 Web APIs: Browser's Superpowers

📚 What are Web APIs?

None of the below are part of JavaScript! These are extra superpowers that browser provides. Browser gives access to JS call stack to use these powers.



🔧 Common Web APIs:

Web API	Purpose	Example Usage
<code>setTimeout()</code>	Timer function	<code>setTimeout(callback, delay)</code>
<code>DOM APIs</code>	HTML DOM tree access	<code>document.getElementById()</code>
<code>fetch()</code>	Network requests	<code>fetch('https://api.example.com')</code>
<code>localStorage</code>	Browser storage	<code>localStorage.setItem()</code>

console	Developer tools	console.log() (Yes, even console is not JS!)
location	URL/navigation	window.location.href

Window Object: The Global Gateway

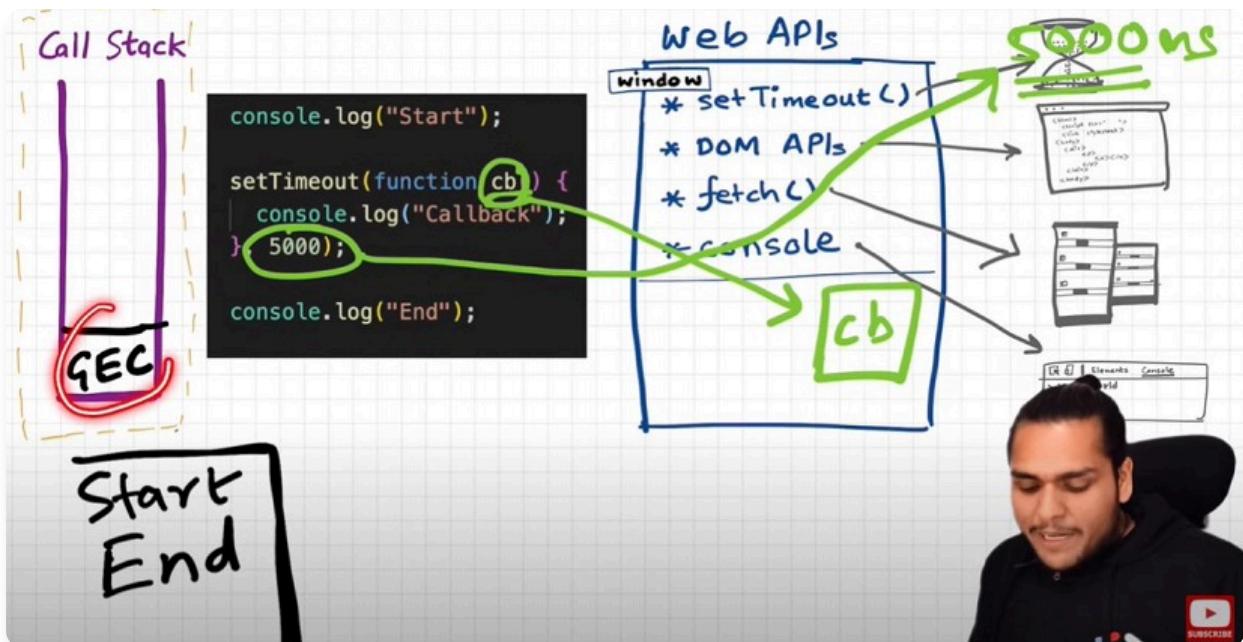
We get all these APIs through the global `window` object:

```
// These are all equivalent:  
setTimeout(callback, 1000);  
window.setTimeout(callback, 1000);  
  
console.log("Hello");  
window.console.log("Hello");  
  
localStorage.setItem("key", "value");  
window.localStorage.setItem("key", "value");
```

 Why we don't write `window` : As `window` is the global object, and all Web APIs are present in it, we don't explicitly write `window` - but it's always implied.

Event Loop in Action

Basic Example: Understanding the Flow



```

console.log("start");
setTimeout(function cb() {
    console.log("timer");
}, 5000);
console.log("end");

// Output:
// start
// end
// timer (after 5 seconds)

```

🧠 Step-by-Step Execution:

Phase 1: Synchronous Execution

1. GEC created → Global Execution Context enters call stack
2. **console.log("start")** → Calls console Web API → Prints "start"
3. **setTimeout(..., 5000)** → Calls setTimeout Web API:
 - Stores callback **cb()** in Web API environment

- Starts 5-second timer
 - Returns immediately (non-blocking)
4. `console.log("end")` → Calls console Web API → Prints "end"
5. GEC completes → Pops from call stack

Phase 2: Timer Completion

6. Timer expires → After 5 seconds, `cb()` needs to execute
7. But wait! → `cb()` cannot directly enter call stack
8. Callback Queue → `cb()` first goes to callback queue
9. Event Loop → Checks if call stack is empty, then moves `cb()` to call stack
10. Execution → `cb()` executes, prints "timer", then pops from call stack



Event Loop: The Gatekeeper

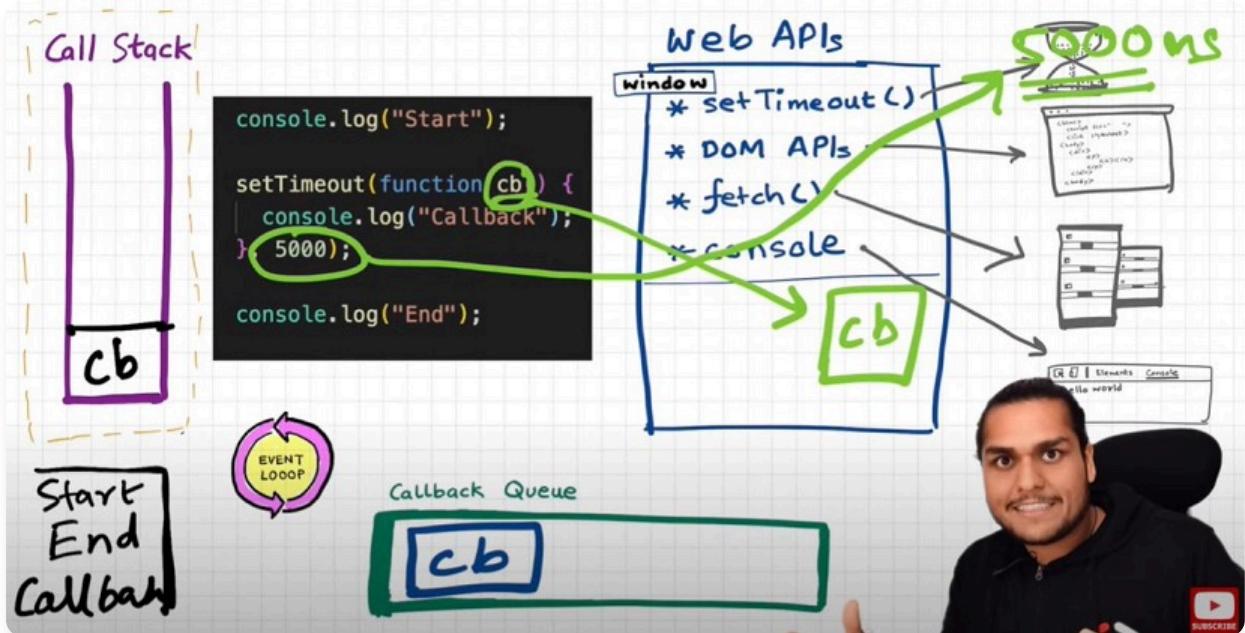
Event Loop's Job:

- Continuously monitor call stack and callback queue
 - If call stack is empty AND callback queue has tasks → Move task to call stack
 - Never interrupt ongoing execution in call stack
-



Callback Queue Deep Dive

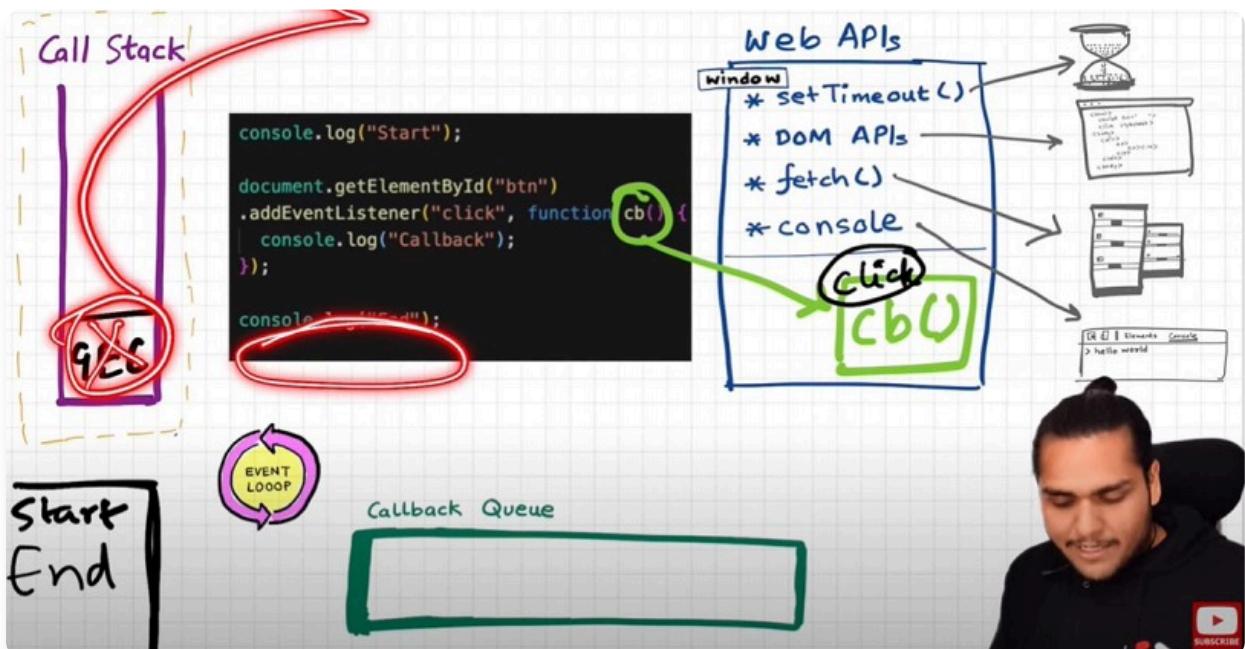
? Q: How does the timer callback reach the call stack after 5 seconds?



Answer: Through the **Callback Queue** and **Event Loop** mechanism:

1. Timer expires → `cb()` ready for execution
2. Enters **callback queue** → Waits for its turn
3. Event loop monitors → Checks call stack status
4. Call stack empty? → Event loop moves `cb()` to call stack
5. Execution → `cb()` runs and completes

🔍 Event Listener Example



```

console.log("Start");

document.getElementById("btn").addEventListener(
function cb() {
    // cb() registered in Web API environment
    // with click event attached
    console.log("Callback");
};

console.log("End");

// Output immediately:
// Start
// End
//
// Output when button clicked:
// Callback

```

What Happens:

1. Event listener registered → `cb()` stored in Web API environment
2. Event attached → Browser watches for click events on button
3. Execution continues → "Start" and "End" printed, GEC pops
4. Event listener persists → Stays in Web API environment indefinitely
5. User clicks → `cb()` moves to callback queue
6. Event loop → Moves `cb()` to call stack for execution

 **Important:** Event listeners stay in Web API environment until explicitly removed or browser closed!

❓ Q: Why do we need a callback queue?

Scenario: User clicks button 6 times rapidly

Answer:

1. **6 callbacks** → All 6 `cb()` instances enter callback queue
2. **Sequential processing** → Event loop processes them one by one
3. **No conflicts** → Each callback executes completely before next one
4. **FIFO order** → First click processed first, last click processed last

 **Benefit:** Maintains order and prevents callback conflicts.

⚡ Microtask Queue: The Priority Lane

🔍 Fetch API Behavior

```

console.log("Start");

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

fetch("https://api.netflix.com").then(function
cbF() {
  console.log("CB Netflix");
}); // Takes 2 seconds to get response

// Millions of lines of code here...

```

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

// Actual Output:
// Start
// End
// CB Netflix (after 2 seconds)
// CB Timeout (after 5 seconds)
```

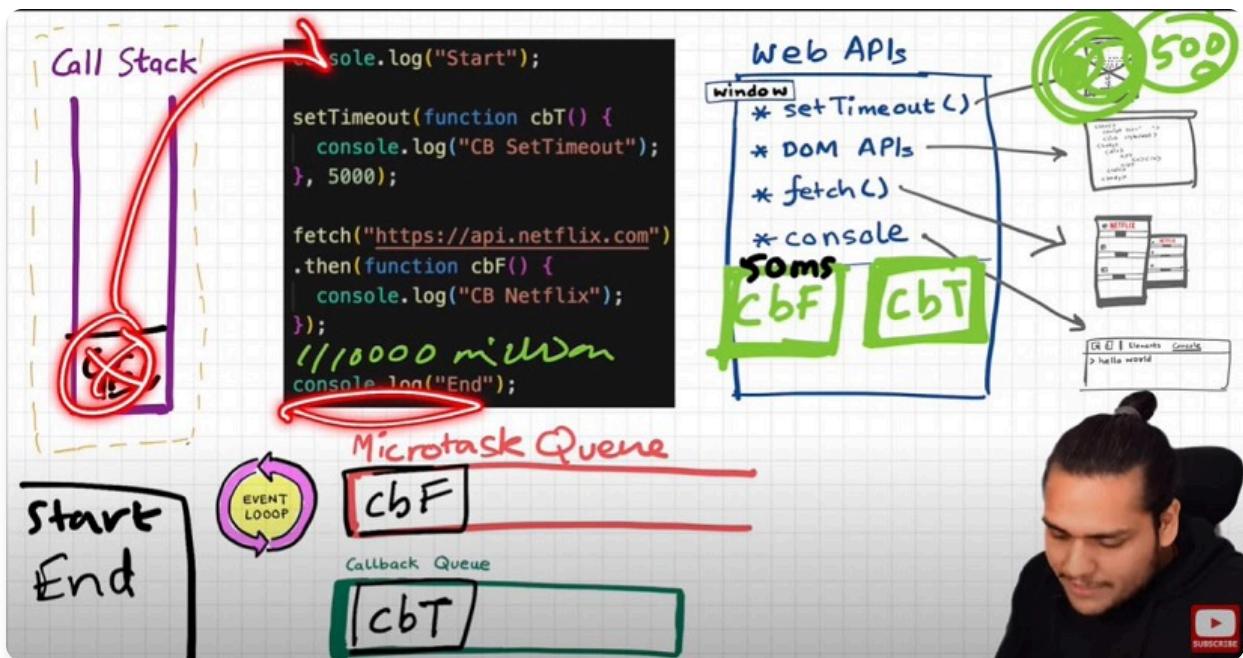
Execution Analysis:

Phase 1: Registration

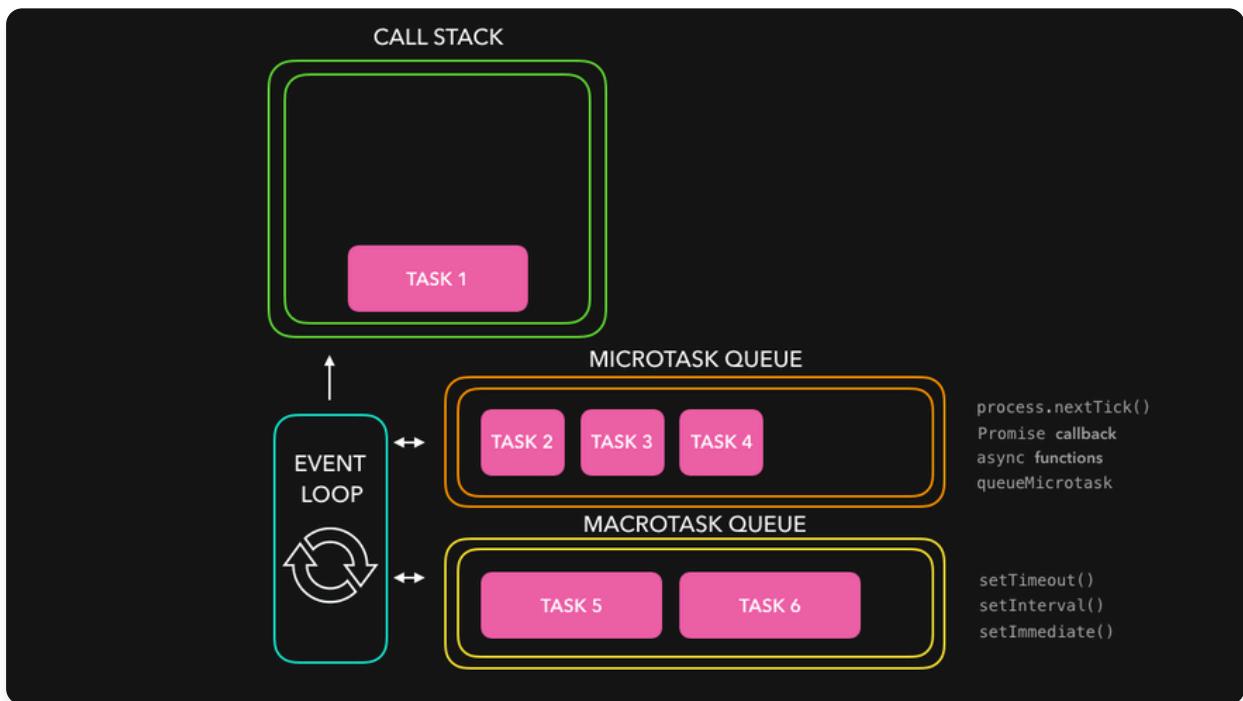
1. `console.log("Start")` → Prints "Start"
2. `setTimeout` → `cbT` registered in Web API, 5-second timer starts
3. `fetch` → `cbF` registered in Web API, network request initiated
4. Millions of lines → Execute synchronously
5. `console.log("End")` → Prints "End"

Phase 2: Async Completion

6. After 2 seconds → Netflix response ready, `cbF` enters **Microtask Queue**
7. After 5 seconds → Timer expires, `cbT` enters **Callback Queue**
8. Event loop priority → Processes Microtask Queue first
9. `cbF executes` → Prints "CB Netflix"
10. `cbT executes` → Prints "CB Timeout"



🏆 Priority System



Event Loop Priority:

1. Call Stack (highest priority)
2. Microtask Queue
3. Callback Queue (lowest priority)

⚡ Microtask Queue gets higher priority than Callback Queue!

📊 What Goes Where?

Queue Type	Contents	Examples
Microtask Queue	Promise callbacks, Mutation Observer	.then() , .catch() , .finally()
Callback Queue	All other async callbacks	setTimeout , setInterval , DOM events

🔍 Detailed Breakdown:

Microtask Queue:

- **Promise callbacks** → .then() , .catch() , .finally()
- **Mutation Observer** → Watches DOM changes and executes callbacks
- **queueMicrotask()** → Manually queued microtasks

Callback Queue (Task Queue):

- **Timer callbacks** → setTimeout , setInterval
- **DOM events** → Click, scroll, resize handlers
- **I/O operations** → File operations, network (excluding fetch promises)

⚠ Starvation Problem

```
// ❌ This can cause starvation
function recursiveMicrotask() {
  Promise.resolve().then(() => {
    console.log("Microtask");
    recursiveMicrotask(); // Creates new
```

```

microtask
  });
}

setTimeout(() => {
  console.log("This may never run!"); // Starved callback
}, 0);

recursiveMicrotask();

```

Problem: If microtask queue keeps creating new tasks, callback queue never gets a chance to run!

? Important Questions & Answers

1. When does the event loop actually start?

Answer: Event loop, as the name suggests, is a single-thread, loop that is **almost infinite**. It's **always running** and doing its job from the moment the JavaScript runtime starts.

```

// Event loop is running even before this
code executes
console.log("Event loop was already
running!");

```

2. Are only asynchronous Web API callbacks registered in Web API environment?

Answer: YES, only asynchronous callbacks go through Web APIs. Synchronous callbacks like those in `map`, `filter`, and `reduce` are **not** registered in Web API environment.

```
// ✗ NOT in Web API (synchronous callback)
[1, 2, 3].map(function(x) {
    return x * 2;
});

// ✓ IN Web API (asynchronous callback)
setTimeout(function() {
    console.log("Async callback");
}, 1000);
```

3. Does Web API environment store the callback function and push the same to queue?

Answer: Yes, callback functions are stored in Web API environment, and a reference is scheduled in the queues.

⚠ Special Case: Event listeners (click handlers) stay in Web API environment **forever** until explicitly removed. This is why it's advised to remove listeners when not needed for garbage collection.

```
// Event listener stays in Web API forever
button.addEventListener('click',
    handleClick);
```

```
// Good practice: Remove when done
button.removeEventListener('click',
handleClick);
```

4. What if setTimeout delay is 0ms?

Answer: There are trust issues with `setTimeout()` 😅. Even with 0ms delay:

```
setTimeout(() => {
  console.log("This might wait!");
}, 0);

// Heavy synchronous operation
for (let i = 0; i < 1000000000; i++) {
  // Blocking operation
}

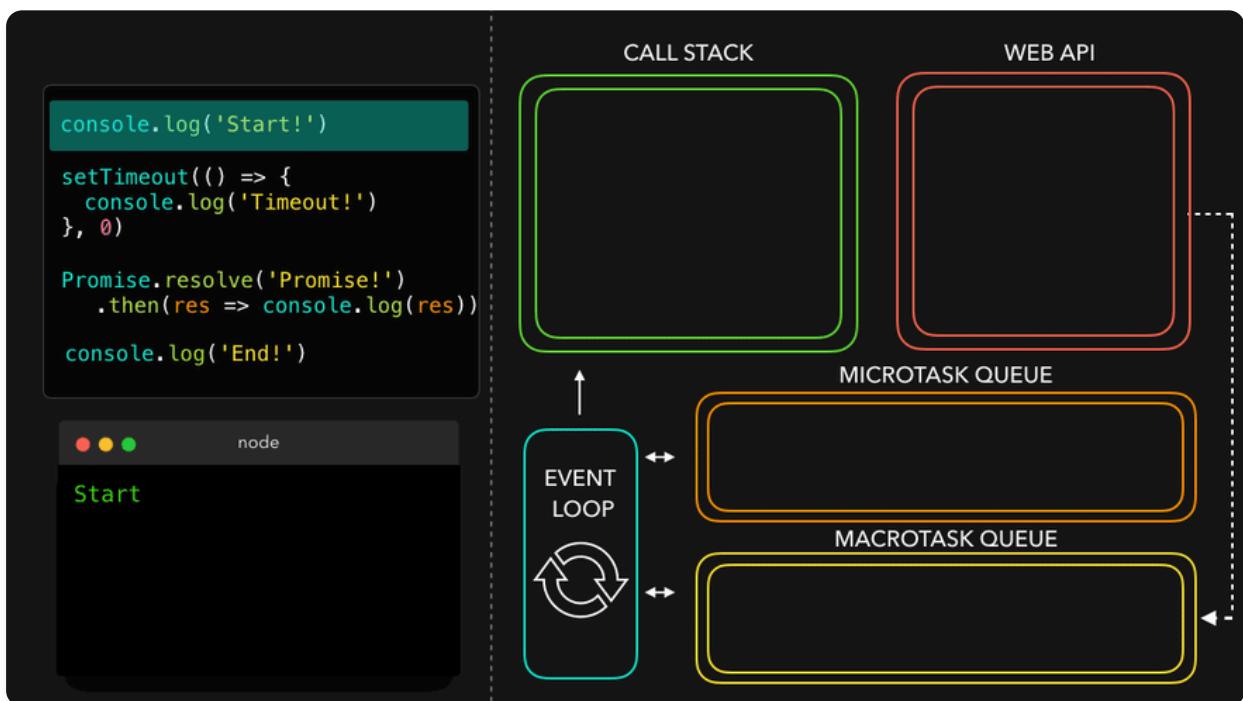
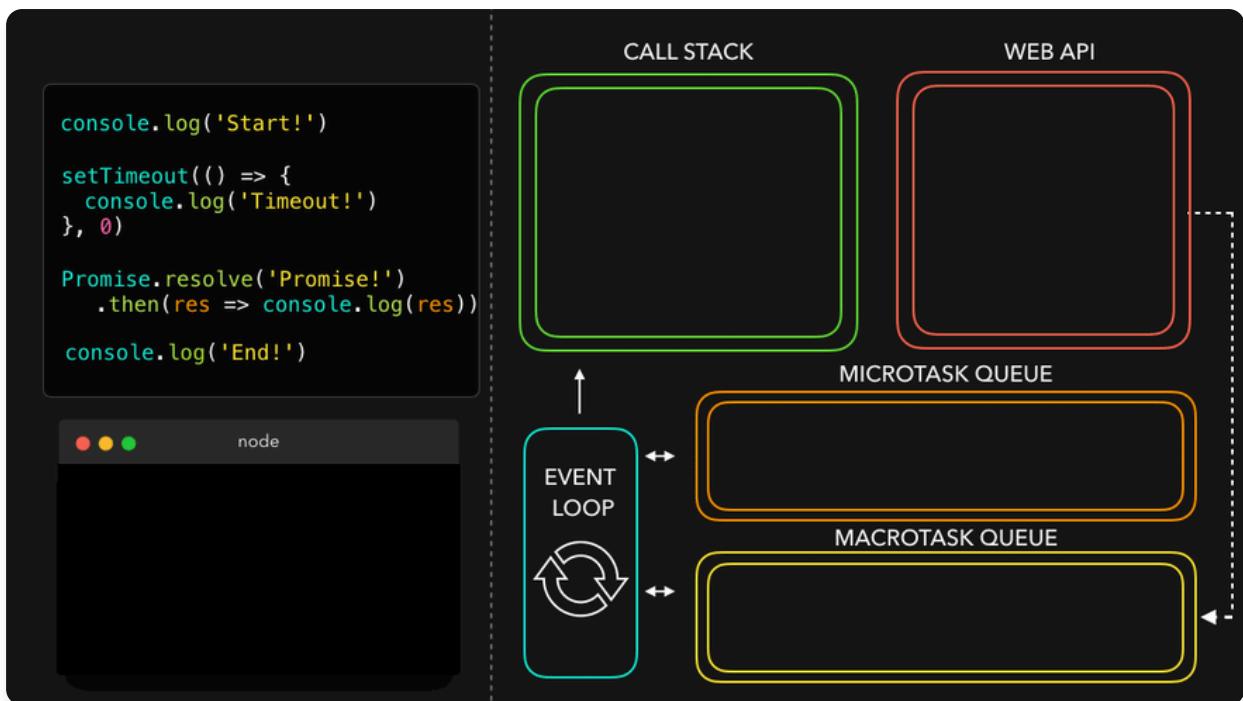
console.log("This runs first!");
```

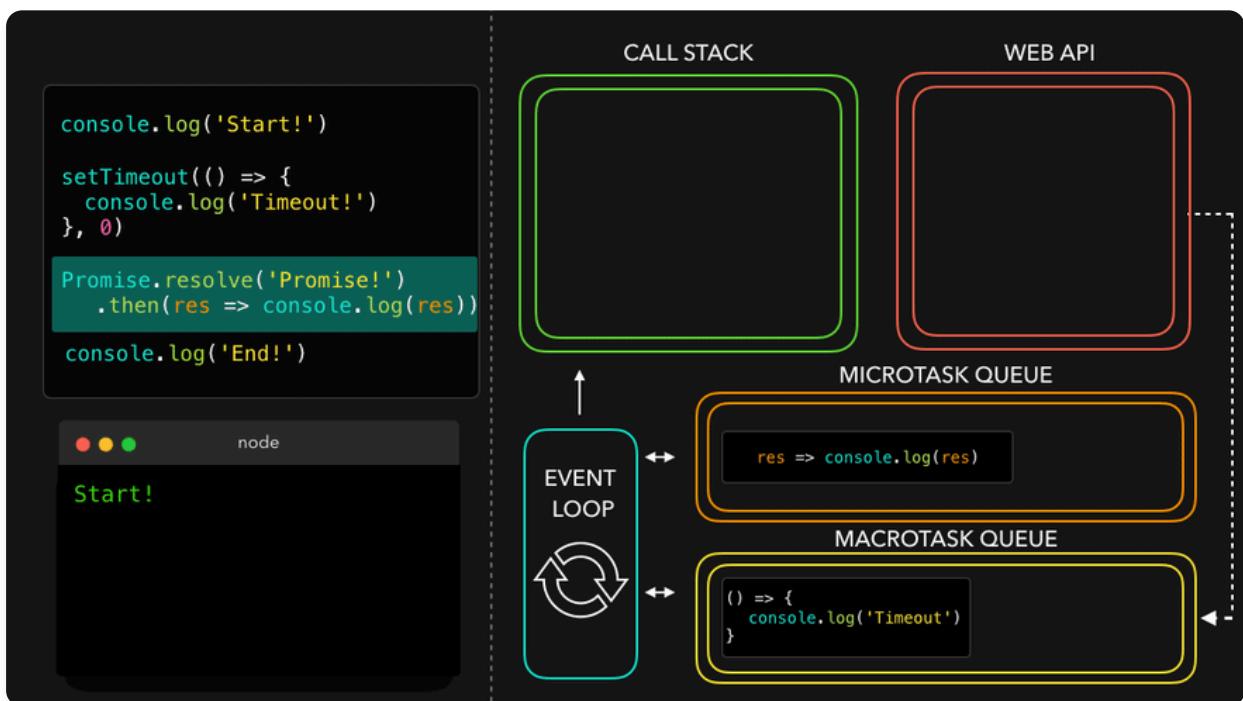
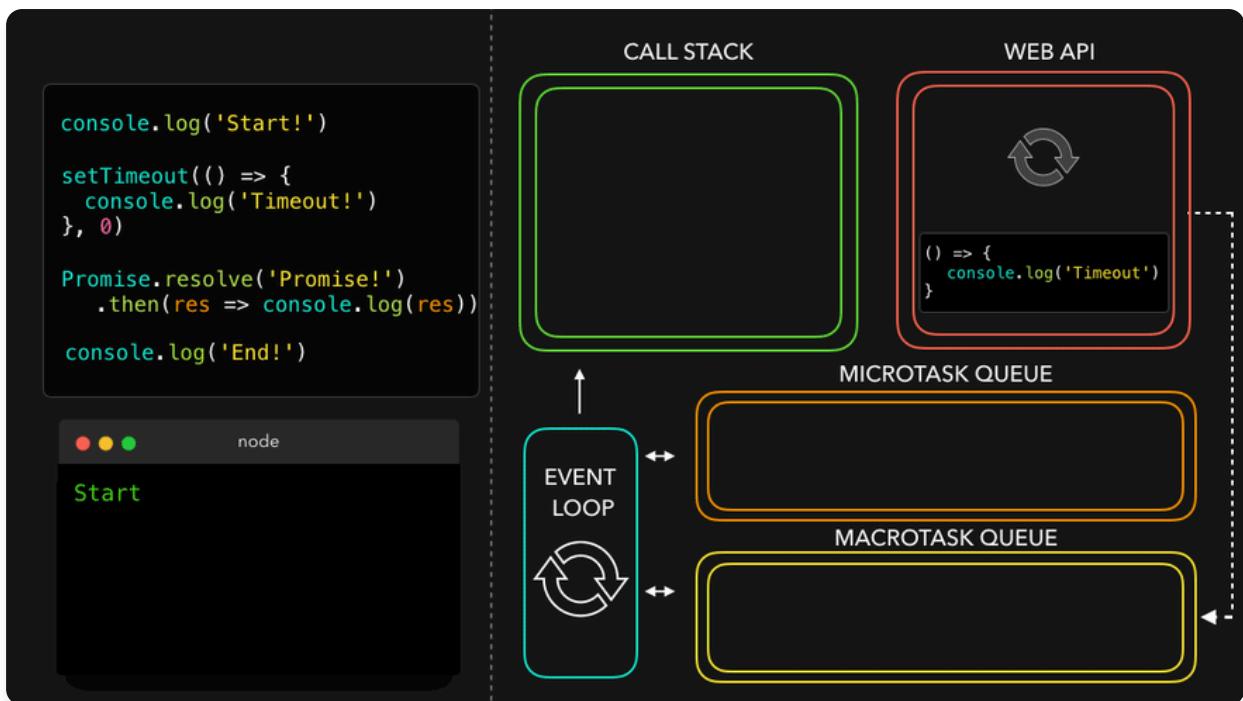
Why? The callback needs to wait until the **call stack is empty**. So the 0ms callback might wait for 100ms+ if the stack is busy.

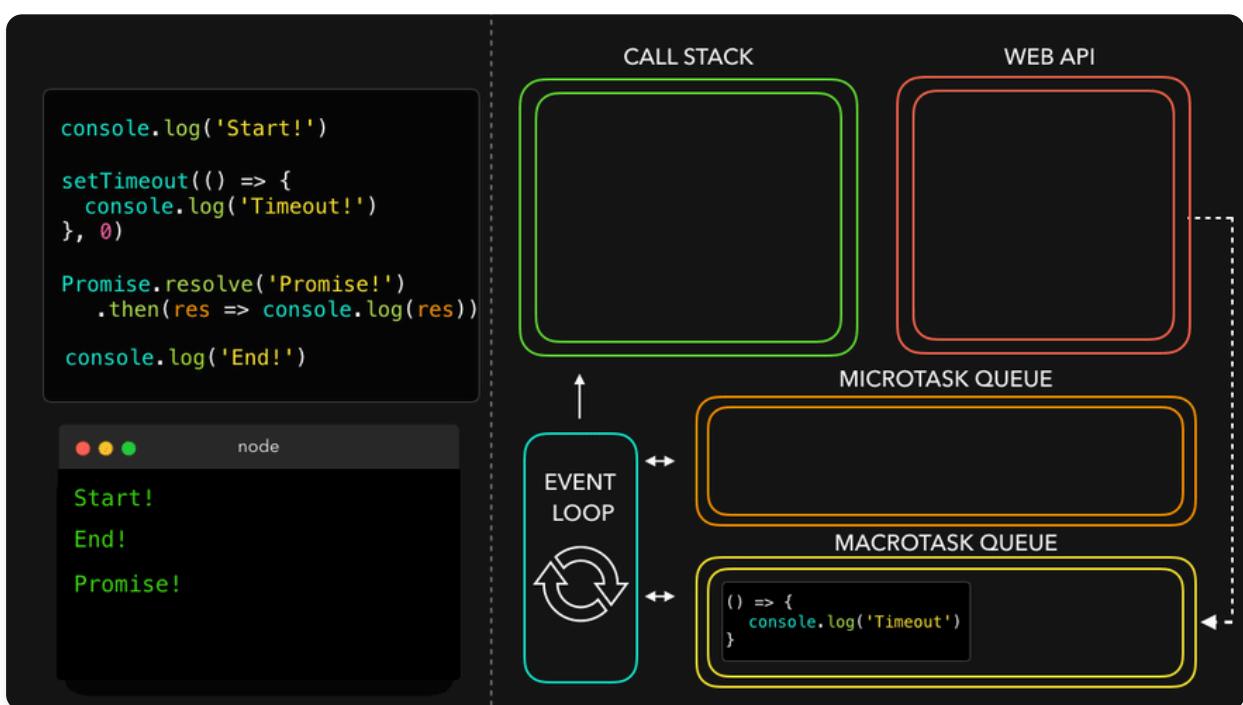
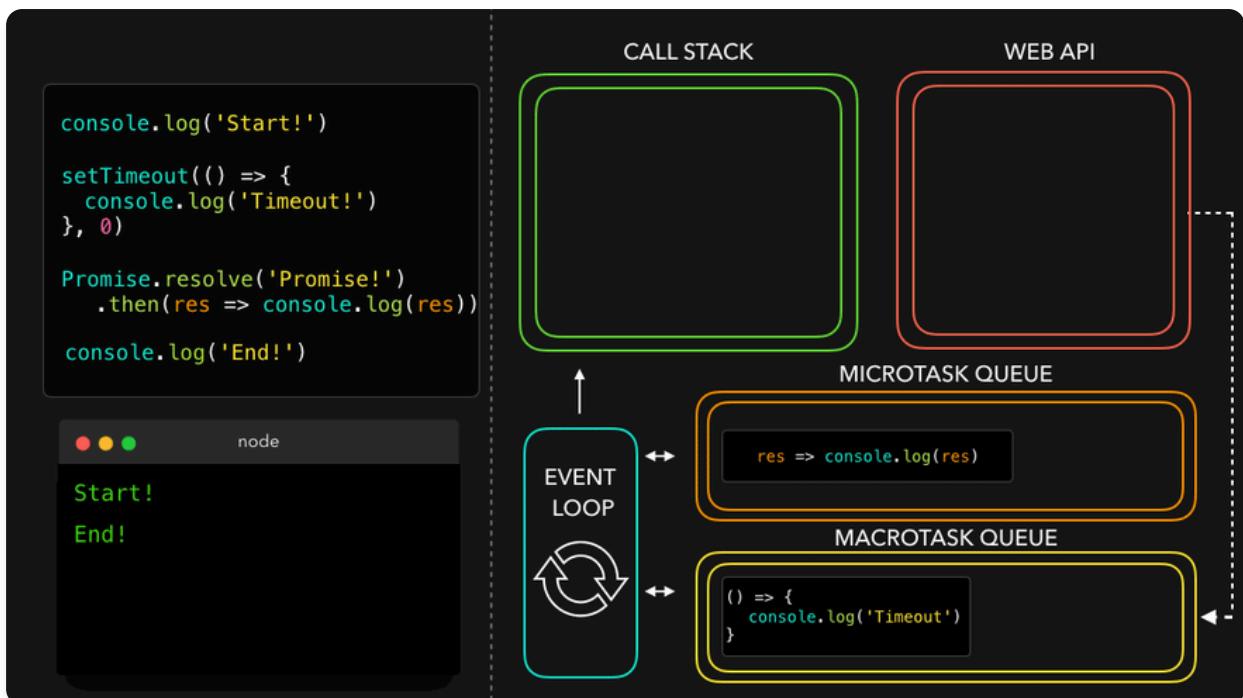


Visual Learning: Event Loop in Motion

Interactive Visualizations







Pro Tip: Study these GIFs to visualize how different async operations flow through the event loop system!

Quick Summary

What We Learned:

1. Browser Architecture

- JavaScript Engine + Web APIs + Event Loop System
- Web APIs provide async capabilities to single-threaded JavaScript
- Window object is the gateway to all Web APIs

2. Event Loop Mechanism

- Call Stack executes synchronous code
- Web APIs handle asynchronous operations
- Event Loop coordinates between queues and call stack

3. Queue Priority System

- **Microtask Queue** → Higher priority (Promises, Mutation Observer)
- **Callback Queue** → Lower priority (setTimeout, DOM events)
- Event Loop always processes microtasks before callbacks

4. Common Patterns

- Timer callbacks go through callback queue
- Promise callbacks go through microtask queue
- Event listeners persist in Web API environment

Quick Memory Aid:

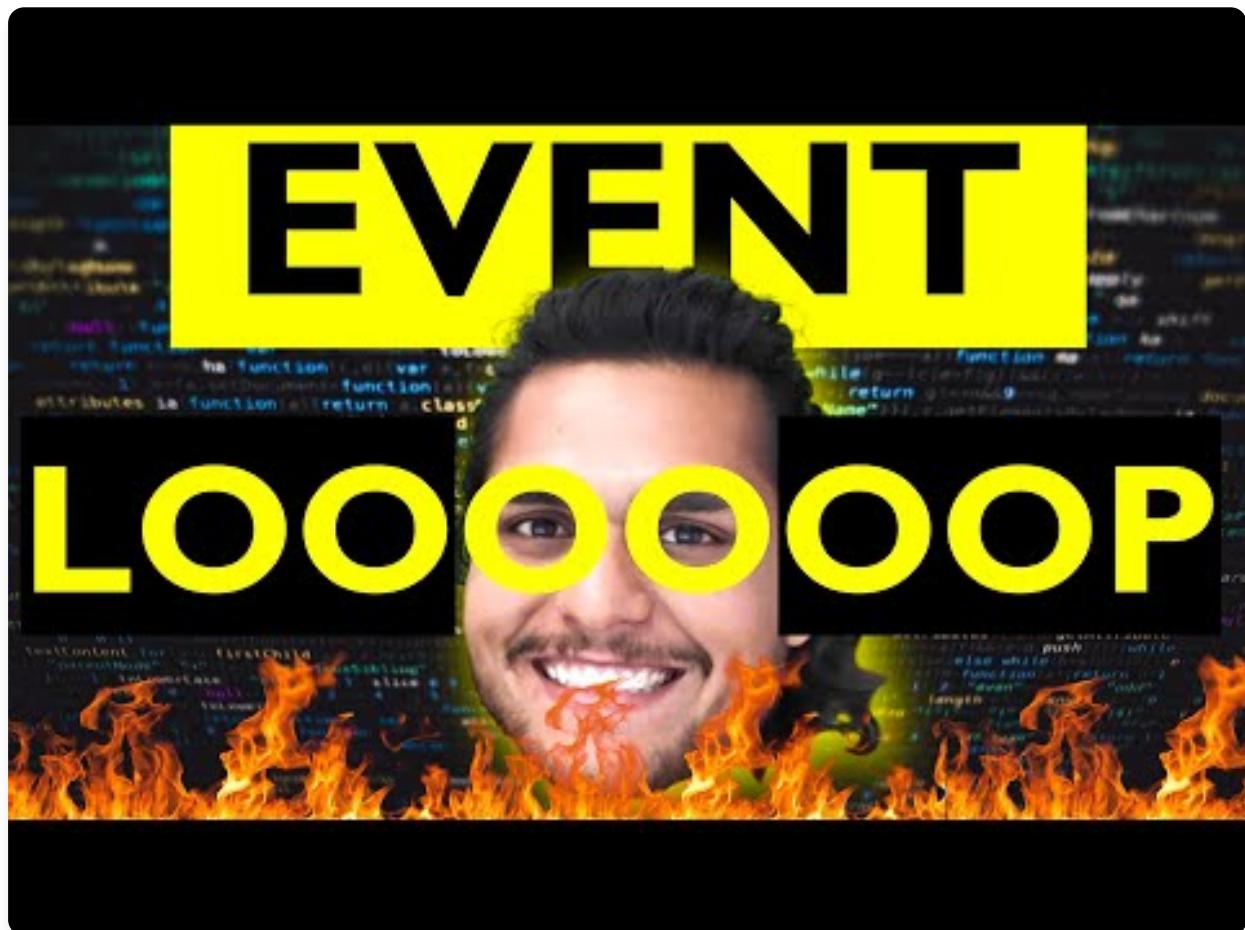
Single Thread + Web APIs = Async JavaScript
 Call Stack → Microtask Queue → Callback Queue
 Promises = Microtask (high priority)
 setTimeout = Callback (low priority)
 Event Loop = The coordinator between all

Where You'll Use This:

Understanding the event loop helps with:

- Debugging timing issues in async code
 - Optimizing performance of web applications
 - Understanding why certain code executes in specific order
 - Writing better asynchronous JavaScript
-

🎥 Watch the Video



Episode 16: JS Engine Exposed, Google's V8 Architecture

What You'll Learn

- Understanding JavaScript Runtime Environment (JRE) and its components
 - How JavaScript engines work and their internal architecture
 - The three phases of JavaScript code execution: Parsing, Compilation, and Execution
 - Just-in-Time (JIT) compilation in modern JavaScript engines
 - Google's V8 engine architecture and optimization techniques
 - ECMAScript standards and different JavaScript engines
-



JavaScript Runtime Environment (JRE)

Why JavaScript Runs Everywhere

JavaScript runs literally **everywhere** - from smart watches to robots to browsers - because of the **JavaScript Runtime Environment (JRE)**.

What is JavaScript Runtime Environment?

JRE is like a **big container** which has everything required to run JavaScript code.

JRE Components:

JavaScript Runtime Environment (JRE)

- └── JavaScript Engine ❤️ (Heart of JRE)
- └── Web APIs (Browser) / Node APIs (Node.js)
- └── Event Loop
- └── Callback Queue
- └── Microtask Queue
- └── Other runtime-specific features

Examples of JRE:

Environment	JavaScript Engine	Additional APIs
Chrome Browser	V8	DOM APIs, Fetch, localStorage
Firefox Browser	SpiderMonkey	DOM APIs, Fetch, localStorage
Node.js	V8	File System, HTTP, Process
Edge Browser	V8 (Chromium)	DOM APIs, Fetch, localStorage

 **Key Insight:** Browser can execute JavaScript code because it has the JavaScript Runtime Environment.



ECMAScript and JavaScript Engines

What is ECMAScript?

ECMAScript is the governing body of JavaScript. It sets the rules and standards that all JavaScript engines must follow.

Popular JavaScript Engines

Engine	Created By	Used In	Special Features
V8	Google	Chrome, Node.js, Edge	High performance, JIT compilation
SpiderMonkey	Mozilla (JS Creator)	Firefox	First ever JavaScript engine
Chakra	Microsoft	Internet Explorer	Legacy engine

JavaScriptCore	Apple	Safari	Optimized for iOS/macOS
----------------	-------	--------	-------------------------

 **Historical Note:** SpiderMonkey was the **first JavaScript engine** created by JavaScript's creator himself (Brendan Eich).

JavaScript Engine Architecture

What is a JavaScript Engine?

JavaScript Engine is NOT a machine. It's **software written in low-level languages** (like C++) that:

- Takes **high-level JavaScript code** as input
- Spits out **low-level machine code** as output

Three Phases of Code Execution

All JavaScript code passes through **3 essential steps**:

JavaScript Code → Parsing → Compilation → Execution → Machine Code

Phase 1: Parsing

What happens during Parsing?

Code is broken down into **tokens** for analysis.

Example: Tokenization

```
let a = 7;
```

Tokens created:

- `let` → Keyword token
- `a` → Identifier token
- `=` → Assignment operator token
- `7` → Number literal token
- `;` → Semicolon token



Abstract Syntax Tree (AST)

Syntax Parser takes tokens and converts them into an **Abstract Syntax Tree (AST)**.

AST Structure:

```
{
  "type": "VariableDeclaration",
  "start": 0,
  "end": 9,
  "declarations": [
    {
      "type": "VariableDeclarator",
      "id": {
        "type": "Identifier",
        "name": "a"
      }
    }
  ]
}
```

```

    },
    "init": {
        "type": "Literal",
        "value": 7
    }
],
"kind": "let"
}

```

 Try it yourself: Visit astexplorer.net to see how your code converts to AST!

 AST is like package.json but for a line of JavaScript code - it contains all the structural information needed for compilation.

Phase 2: Compilation (JIT)

Just-in-Time (JIT) Compilation

Modern JavaScript uses **Just-in-Time (JIT) Compilation** - a hybrid approach that uses **both interpreter AND compiler**.

 Key Characteristic: Compilation and execution go hand in hand.

JIT Compilation Process:

1. AST → Interpreter → Converts to **bytecode**
2. Interpreter starts **execution** → Immediate code execution
3. Compiler works in **parallel** → Identifies optimization opportunities
4. **Hot code optimization** → Frequently used code gets compiled to optimized machine code

5. Runtime optimization → Continuous improvement during execution

❓ Does JavaScript Really Compile?

The answer is a loud YES! 🎉

Evidence:

- **Modern V8 Engine:** Uses both interpreter (Ignition) and compiler (TurboFan)
- **Performance gains:** JIT compilation provides significant speed improvements
- **Runtime optimization:** Code gets faster as it runs more frequently

📚 Further Reading:

- [You Don't Know JS - Interpretation vs Compilation](#)
- [Stanford CS Course - JavaScript Overview](#)
- [JavaScript: Interpreted or Compiled?](#)

📊 Evolution of JavaScript Execution:

Era	Execution Method	Performance	Characteristics
Early JS	Pure Interpreter	Slow	Line-by-line execution
Modern JS	JIT Compilation	Fast	Best of both worlds

💡 **JIT Compilation = Best of both worlds:** Fast startup (interpreter) + High performance (compiler)

🚀 Phase 3: Execution

🔧 Execution Components

JavaScript execution requires **2 main components**:

1. ⚡ Memory Heap

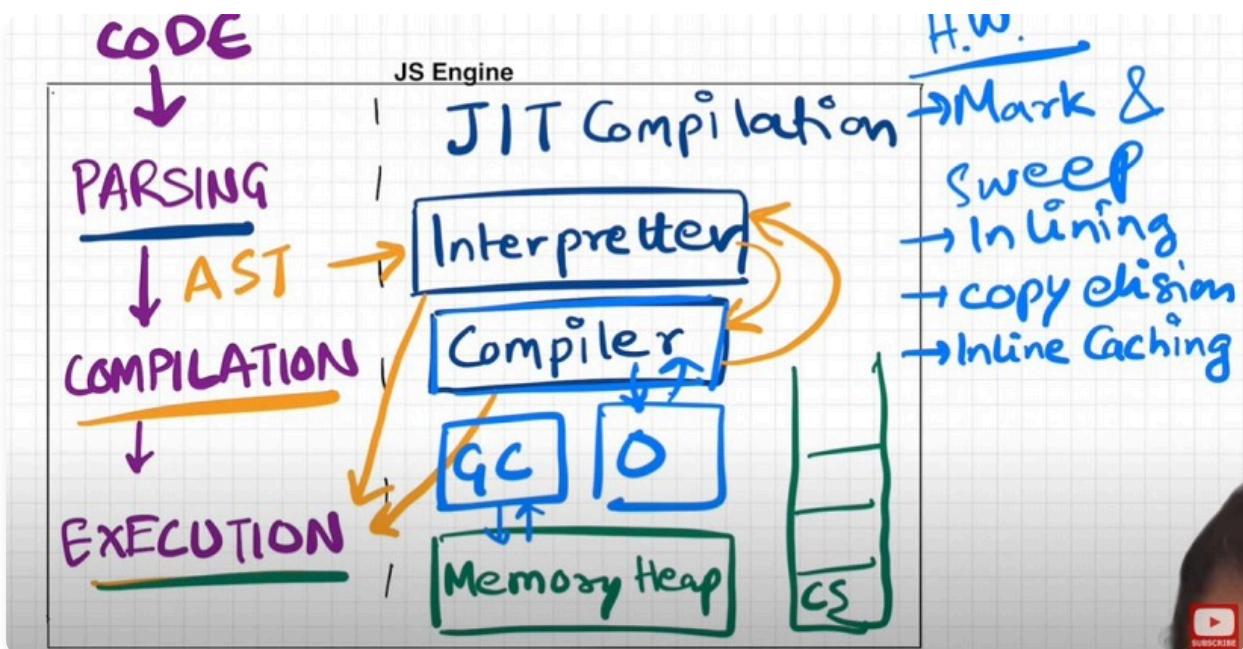
- **Purpose:** Where all memory allocation happens
- **Stores:** Objects, arrays, functions, variables
- **Management:** Automatic memory allocation and deallocation

2. 📜 Call Stack

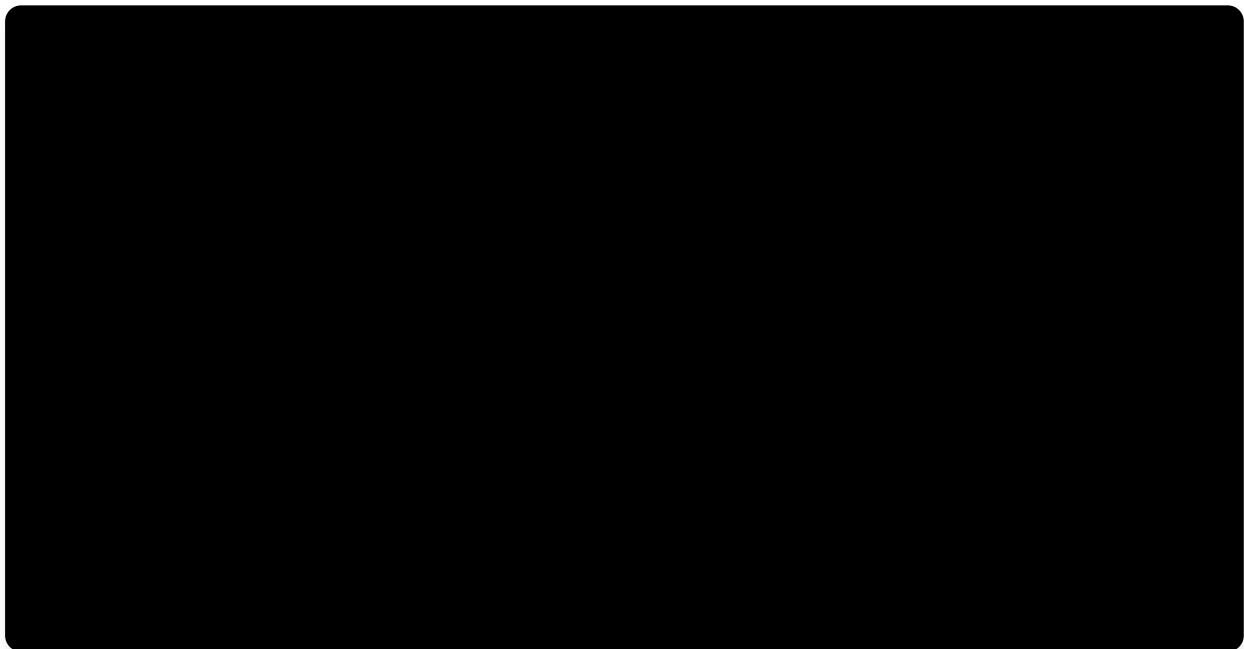
- **Purpose:** Execution context management (same call stack from previous episodes!)
- **Function:** Tracks function calls and execution order
- **Behavior:** LIFO (Last In, First Out)

3. 🗑️ Garbage Collector

- **Algorithm:** Mark and Sweep
- **Purpose:** Automatic memory cleanup
- **Process:**
 1. **Mark:** Identify unreachable objects
 2. **Sweep:** Remove unmarked objects from memory



🎬 Execution in Motion



🏛️ Google's V8 Engine Architecture

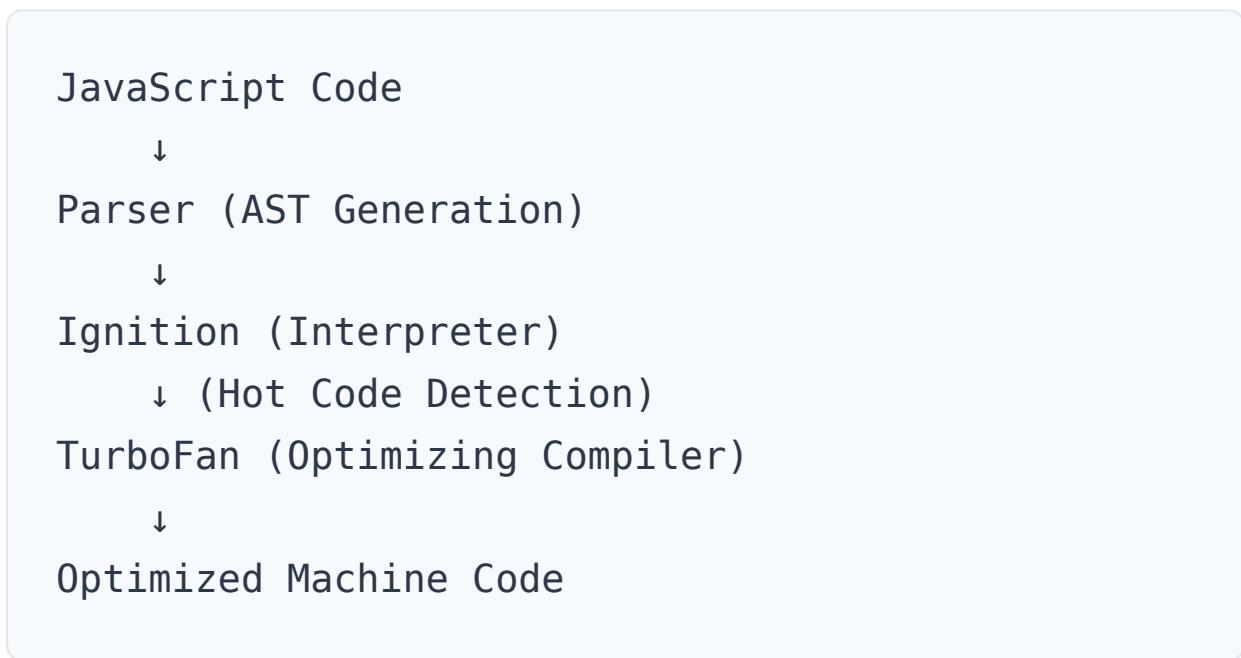
🔥 V8: The Performance Beast

Google's V8 engine powers Chrome, Node.js, and Edge browsers. It's renowned for its exceptional performance and advanced optimization techniques.

🌟 V8 Components

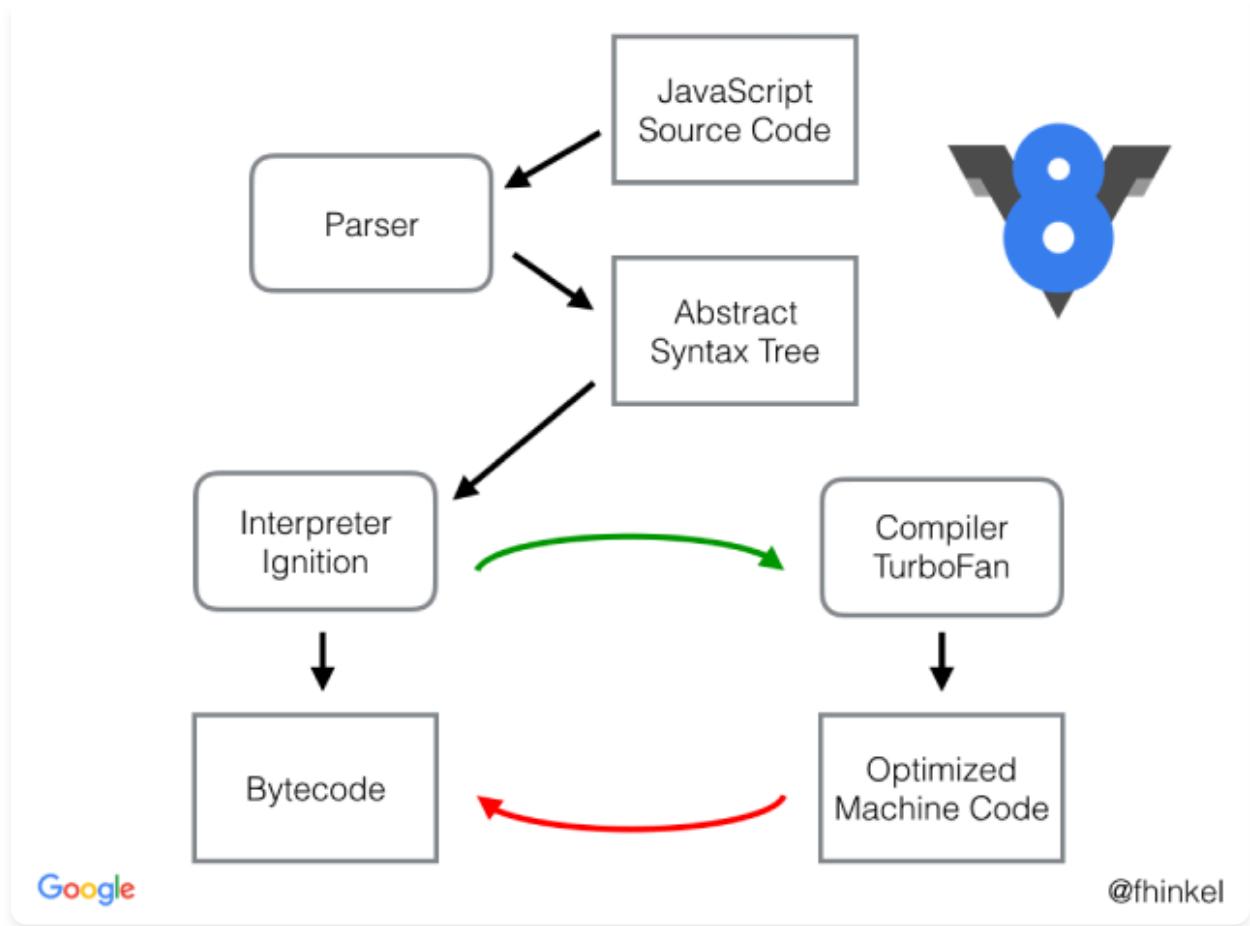
Component	Type	Purpose
Ignition	Interpreter	Fast startup, bytecode generation
TurboFan	Optimizing Compiler	High-performance machine code
Orinoco	Garbage Collector	Concurrent memory management

➡️ V8 Execution Pipeline:



🔥 V8 Optimization Magic:

1. **Ignition starts fast** → Immediate bytecode execution
2. **Profiling during execution** → Identifies frequently used code
3. **TurboFan optimization** → Hot code gets compiled to highly optimized machine code
4. **Speculative optimization** → Makes assumptions about data types
5. **Deoptimization** → Falls back if assumptions are wrong
6. **Continuous optimization** → Gets faster over time



💡 V8 Performance Techniques:

Hidden Classes:

- Optimizes object property access
- Creates internal structure representations
- Enables fast property lookup

Inline Caching:

- Caches property access patterns
- Speeds up repeated operations
- Adapts to changing object shapes

Concurrent Garbage Collection (Orinoco):

- Runs garbage collection in parallel
- Minimizes main thread blocking

- Maintains application responsiveness
-



Engine Competition and Innovation

The Race for Performance

Companies continuously compete to make their JavaScript engines the **fastest** and **most efficient**:

Google V8:

- **Focus:** Maximum performance, Node.js compatibility
- **Strengths:** JIT optimization, concurrent GC
- **Used by:** Chrome, Edge, Node.js

Mozilla SpiderMonkey:

- **Focus:** Standards compliance, innovation
- **Strengths:** First engine, experimental features
- **Used by:** Firefox

Apple JavaScriptCore:

- **Focus:** Mobile optimization, energy efficiency
- **Strengths:** iOS/macOS integration
- **Used by:** Safari



Performance Benchmarks

Modern JavaScript engines compete on:

- **Startup time** → How fast code begins executing
- **Peak performance** → Maximum execution speed

- **Memory usage** → Efficient memory utilization
 - **Battery efficiency** → Important for mobile devices
-

Quick Summary

What We Learned:

1. JavaScript Runtime Environment

- Container that provides everything needed to run JavaScript
- Includes JS Engine, APIs, Event Loop, and Queues
- Enables JavaScript to run everywhere

2. Three-Phase Execution

- **Parsing:** Code → Tokens → AST
- **Compilation:** JIT approach with interpreter + compiler
- **Execution:** Memory Heap + Call Stack + Garbage Collector

3. Modern JavaScript is Compiled

- JIT compilation provides best of both worlds
- Continuous optimization during runtime
- Performance improves with code hotness

4. V8 Engine Excellence

- Ignition (interpreter) + TurboFan (compiler) + Orinoco (GC)
- Advanced optimization techniques
- Powers Chrome, Node.js, and Edge

Quick Memory Aid:

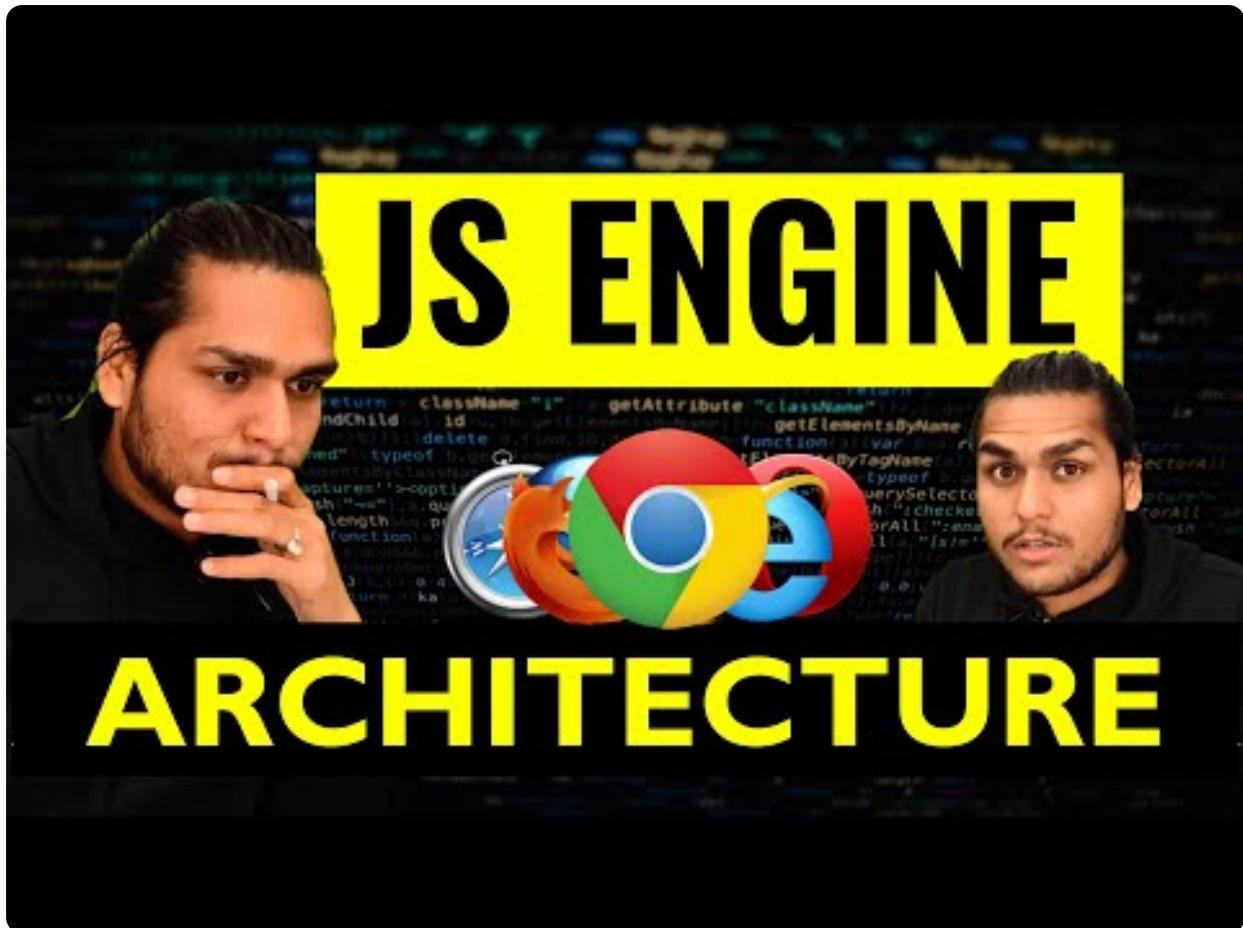
JRE = JS Engine + APIs + Event Loop
Engine Phases = Parse → Compile → Execute
JIT = Interpreter + Compiler (best of both)
V8 = Ignition + TurboFan + Orinoco
AST = JSON representation of code structure

Where You'll Use This:

Understanding JS engines helps with:

- Optimizing code for better performance
 - Understanding why certain patterns are faster
 - Debugging performance bottlenecks
 - Making informed architectural decisions
-

Watch the Video



Episode 17: Trust Issues with setTimeout()

🎯 What You'll Learn

- Why setTimeout doesn't guarantee exact timing execution
 - How call stack blocking affects timer accuracy
 - The concurrency model and event loop interaction with setTimeout
 - Practical implications of main thread blocking
 - setTimeout(0) technique and its use cases
 - Best practices for non-blocking JavaScript code
-



The setTimeout Trust Problem

The Core Issue

`setTimeout` with a 5-second timer does NOT guarantee that the callback will execute exactly after 5 seconds.

The actual execution time depends entirely on the **call stack state** and can be delayed significantly.



Demonstrating the Problem

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

// Millions of lines of code that take 10
seconds to execute
for(let i = 0; i < 1000000000; i++) {
  // Blocking operations
}

// 💥 Output: "Callback" might execute after
10+ seconds, not 5!
```

Why does this happen? Even though we set a 5-second timer, the callback might execute after 6, 7, or even 10+ seconds!

🔍 Step-by-Step Execution Analysis

📋 Detailed Breakdown

Step	Action	Call Stack	Web APIs	Callback Queue	Time
1	console.log("Start")	GEC + console.log	-	-	-
2	setTimeout() registered	GEC	cb() + 5s timer	-	⌚ Start
3	console.log("End")	GEC + console.log	cb() + 5s timer	-	⌚ Run
4	Million lines execute	GEC + heavy code	cb() + 5s timer	-	⌚ Run
5	Timer expires (5s)	GEC + heavy code	-	cb() waiting	⌚ Exp
6	Heavy code continues	GEC + heavy code	-	cb() waiting	⌚ Exp
7	Heavy code finishes (10s)	Empty	-	cb() waiting	⌚ Exp
8	Event loop moves cb()	cb()	-	-	⌚ Exec

Key Insights:

1. Timer vs Execution Are Independent

-  **Timer runs in background** → Completes in exactly 5 seconds
-  **Callback execution waits** → For call stack to be empty
-  **Total delay** → 5s (timer) + 5s (waiting) = 10+ seconds

2. Event Loop Dependency

-  **Event loop constantly checks** → Is call stack empty?
-  **Cannot interrupt** → Currently executing synchronous code
-  **Only moves callback** → When call stack is completely empty

3. Non-Preemptive Execution

-  **JavaScript is single-threaded** → One operation at a time
 -  **No interruption possible** → Until current execution completes
 -  **Callbacks must wait** → No matter how long timer was
-

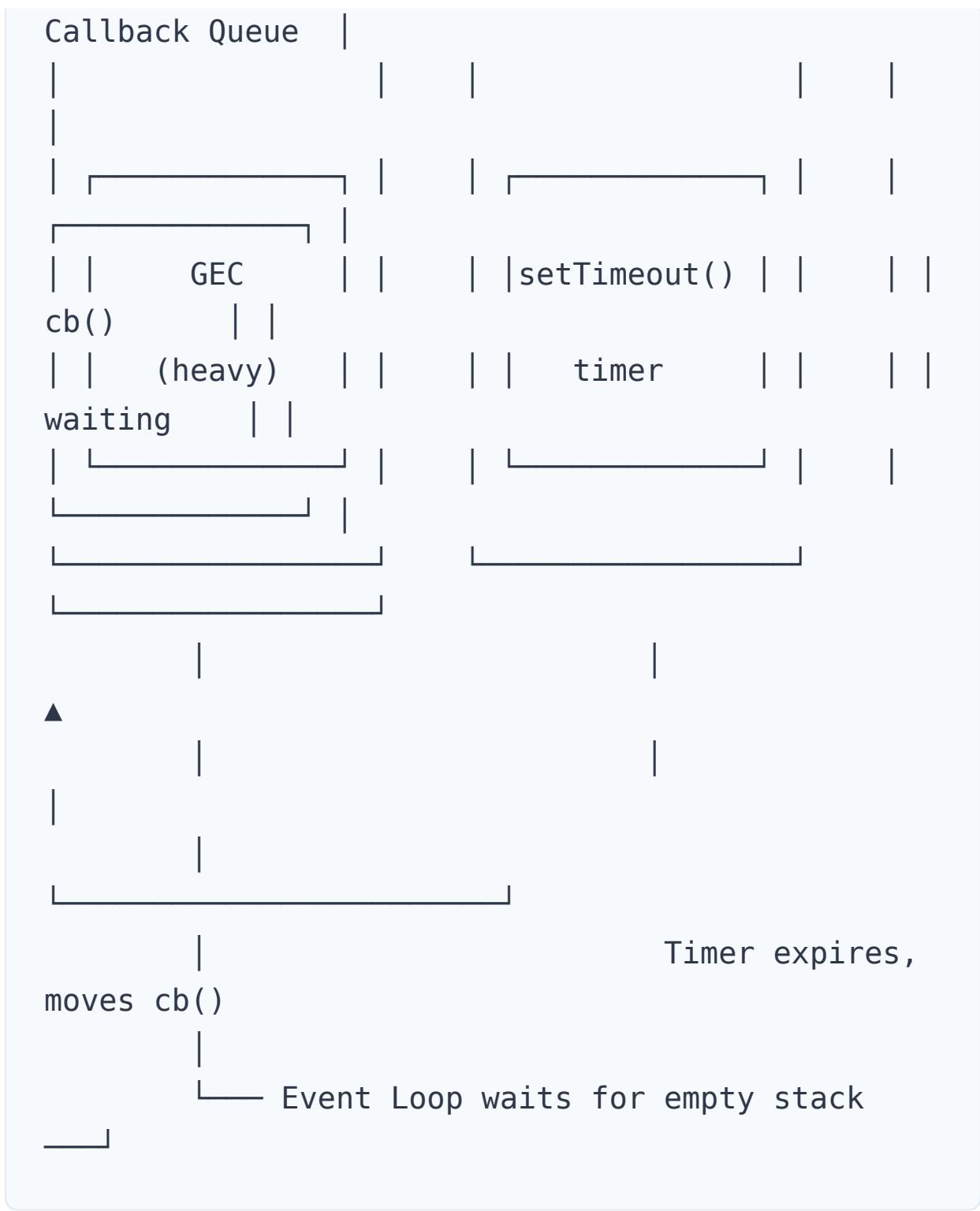
The Concurrency Model

Understanding JavaScript's Concurrency

This behavior is part of JavaScript's [Concurrency Model](#) - how JavaScript handles multiple operations with a single thread.

Concurrency Components:





⚖️ Trust vs Reality Table

What We Expect	What Actually Happens	Reality Check
Execute after 5 seconds	Execute after 5+ seconds	✗ No guarantee

Immediate timer response	Delayed by call stack	Queue dependency
Precise timing	Approximate timing	Best effort only
Interrupts current code	Waits for completion	Non-preemptive

The Cardinal Rule: Don't Block the Main Thread

JavaScript's First Rule

Do NOT block the main thread!

JavaScript is a **single-threaded language** with only **1 call stack**. Blocking it affects everything.

Blocking vs Non-Blocking Comparison

Blocking Code Example:

```
console.log("Start");
setTimeout(() => console.log("Timer"), 1000);

//  BLOCKING: Heavy synchronous operation
for(let i = 0; i < 1000000000; i++) {
    // CPU-intensive loop - blocks everything!
}
console.log("End");
```

```
// Output: "Start" → "End" (after 5+ seconds)
→ "Timer"
// Timer waits until loop finishes!
```

✓ Non-Blocking Code Example:

```
console.log("Start");
setTimeout(() => console.log("Timer"), 1000);

// ✓ NON-BLOCKING: Asynchronous operation
setTimeout(() => {
    for(let i = 0; i < 1000000000; i++) {
        // Heavy work in separate timer
    }
    console.log("Heavy work done");
}, 0);

console.log("End");

// Output: "Start" → "End" → "Timer" (after
~1s) → "Heavy work done"
// Timer executes on schedule!
```

⌚ Practical Impact Demonstration

```

JS index.js  X  < index.html
js > JS index.js
1  console.log("Start");
2
3  setTimeout(function cb() {
4    console.log("Callback");
5  }, 5000);
6
7  console.log("End");
8
9  // million
10
11 let startDate = new Date().getTime();
12 let endDate = startDate;
13 while (endDate < startDate + 10000) {
14   endDate = new Date().getTime();
15 }
16
17 console.log("While expires");
18

```

⚠️ Observe: Even UI interactions become unresponsive when main thread is blocked!

⌚ setTimeout Timing Guarantees

📚 What setTimeout Actually Guarantees

setTimeout guarantees MINIMUM delay, not EXACT delay

🎯 Correct Understanding:

```

setTimeout(callback, 5000);
// ✅ Will execute AT LEAST 5 seconds later

```

```
// ❌ Will NOT execute EXACTLY after 5 seconds
```

Timing Scenarios

Scenario	Timer Set	Call Stack State	Actual Execution	Delay Reason
Ideal	5000ms	Empty	~5000ms	<input checked="" type="checkbox"/> No blocking
Blocked	5000ms	Busy for 10s	~10000ms	<input type="checkbox"/> Call stack busy
Heavy Load	5000ms	Multiple operations	15000ms+	<input type="checkbox"/> Queue backlog



Memory Aid:

```
setTimeout = "Execute AFTER AT LEAST X milliseconds"
            ≠ "Execute EXACTLY AFTER X milliseconds"
```



The setTimeout(0) Technique

What Happens with Zero Timeout?

```

console.log("Start");
setTimeout(function cb() {
  console.log("Callback");
}, 0); // ⚡ Zero timeout!
console.log("End");

// Output: "Start" → "End" → "Callback"
// Even with 0ms, callback still goes through
the queue!

```

⌚ setTimeout(0) Execution Flow:

1. **Registration** → Callback registered in Web APIs
2. **Immediate expiry** → Timer expires instantly (0ms)
3. **Queue placement** → Callback moves to queue immediately
4. **Wait for stack** → Must wait for current synchronous code
5. **Execution** → Runs only after `console.log("End")`

⌚ Practical Use Cases for setTimeout(0)

1. 📋 Task Deferral Pattern

```

function processData(data) {
  console.log("Processing critical data...");

  // Defer less important work
  setTimeout(() => {
    console.log("Updating UI...");
    updateProgressBar();
  });
}

```

```
    }, 0);

    console.log("Critical processing
complete");
}

// Output: Critical work happens first, UI
updates after
```

2. ⚡ Breaking Up Heavy Operations

```
function processLargeArray(array) {
    const CHUNK_SIZE = 1000;
    let index = 0;

    function processChunk() {
        let count = 0;
        while (count < CHUNK_SIZE && index <
array.length) {
            // Process array[index]
            index++;
            count++;
        }

        if (index < array.length) {
            setTimeout(processChunk, 0); // Allow
other code to run
        }
    }
}
```

```
    processChunk();
}
```

3. 🎨 DOM Update Optimization

```
function updateMultipleElements() {
    // Batch DOM updates
    element1.textContent = "Updated";
    element2.style.color = "red";

    // Defer non-critical updates
    setTimeout(() => {
        updateAnalytics();
        logUserActivity();
    }, 0);
}
```



JavaScript's Async Capabilities

🧵 Single-Threaded Yet Asynchronous

JavaScript achieves asynchronous behavior despite being single-threaded through:



Enabling Technologies:

- 🌐 **Web APIs** → Browser provides threading for timers, HTTP, DOM events
- ⚡ **Event Loop** → Coordinates between call stack and queues

-  **Task Queues** → Organize asynchronous callbacks
-  **JIT Compilation** → Fast execution without compilation delays

Benefits:

-  **Fast startup** → No compilation wait time
-  **Responsive** → Non-blocking operations possible
-  **Efficient** → Single thread with cooperative multitasking
-  **Universal** → Runs in browsers, servers, mobile apps

Synchronous vs Asynchronous Execution

Aspect	Synchronous	Asynchronous
Execution	Blocking	Non-blocking
Order	Predictable	Event-driven
Performance	Can block UI	Responsive
Use Cases	Calculations	I/O operations

Common setTimeout Pitfalls

Mistake 1: Expecting Exact Timing

```
// ❌ Wrong expectation
console.time("timer");
setTimeout(() => {
  console.timeEnd("timer"); // Might show
```

```
1005ms instead of 1000ms
```

```
}, 1000);
```

🚫 Mistake 2: Blocking Main Thread

```
// ✗ This blocks everything
setTimeout(() => console.log("Should be
fast"), 100);
for(let i = 0; i < 1000000000; i++) {} // 5
second loop
// Timer waits 5+ seconds instead of 100ms
```

🚫 Mistake 3: Assuming setTimeout(0) is Instant

```
// ✗ Not truly instant
console.log("1");
setTimeout(() => console.log("2"), 0);
console.log("3");
// Output: 1, 3, 2 (not 1, 2, 3)
```

✓ Best Practices

1. ⚡ Use for I/O Operations

```
// ✅ Good: Non-blocking I/O
setTimeout(() => {
  fetch('/api/data').then(handleResponse);
}, 100);
```

2. ⏪ Break Up Heavy Work

```
// ✅ Good: Cooperative processing
function processInChunks(data) {
  const chunk = data.splice(0, 100);
  processChunk(chunk);

  if (data.length > 0) {
    setTimeout(() => processInChunks(data),
    0);
  }
}
```

3. 🕒 Use Appropriate Timing

```
// ✅ Good: Reasonable delays
setTimeout(updateUI, 16); // ~60fps for
animations
setTimeout(saveData, 1000); // 1s for auto-
save
```

```
setTimeout(cleanup, 300000); // 5min for cleanup
```



Quick Summary

Key Takeaways:

1. setTimeout Timing Reality

- **Guarantees:** Minimum delay, not exact timing
- **Depends on:** Call stack availability and event loop
- **Affected by:** Blocking operations and queue backlog

2. Concurrency Model Understanding

- **Single thread** with cooperative multitasking
- **Event loop** manages asynchronous operations
- **Non-preemptive** - cannot interrupt running code

3. Main Thread Protection

- **Never block** the main thread with heavy operations
- **Break up** large tasks into smaller chunks
- **Use async patterns** for I/O and time-consuming work

4. setTimeout(0) Technique

- **Defers execution** until current stack is clear
- **Useful for** task prioritization and UI responsiveness
- **Not instant** - still goes through event loop

 **Quick Memory Aid:**

```
setTimeout = "AT LEAST" not "EXACTLY"  
Main Thread = Never Block It  
setTimeout(0) = Queue It For Later  
Event Loop = Waits For Empty Stack  
Concurrency = Single Thread + Web APIs
```

 **Real-World Applications:**

- 🎮 **Game loops** - Non-blocking animation frames
- 📊 **Data processing** - Chunked large dataset processing
- 🎨 **UI updates** - Deferred non-critical operations
- 📱 **Progressive loading** - Staggered content loading
- ⌚ **Auto-save** - Periodic background saves

 **Remember:**

JavaScript's setTimeout "trust issues" aren't bugs - they're features of the concurrency model that enable non-blocking, responsive applications when used correctly!

 **Watch the Video**



Episode 18: Higher-Order Functions ft. Functional Programming

🎯 What You'll Learn

- Understanding Higher-Order Functions (HOF) and their characteristics
- Functional programming principles and benefits
- DRY principle and code reusability patterns
- Building custom polyfills (map function implementation)
- Evolution from imperative to functional programming style

- Real-world applications of HOFs in modern JavaScript
-

🎯 What are Higher-Order Functions?

📚 Definition

Higher-Order Functions (HOF) are regular functions that either:

- 1. Take one or more functions as arguments, OR*
- 2. Return a function as their result*

🔧 Basic Example

```
function x() {  
    console.log("Hi");  
}  
  
function y(x) {  
    x(); // y accepts function x as parameter  
}  
  
y(x); // Output: "Hi"  
  
// ✅ y is a Higher-Order Function  
// ✅ x is a Callback Function
```



HOF Characteristics Table

Aspect	Higher-Order Function	Regular Function
Parameters	Can accept functions	Only accepts data
Return Value	Can return functions	Only returns data
Reusability	Highly reusable	Limited reusability
Abstraction	High-level operations	Specific operations
Examples	<code>map</code> , <code>filter</code> , <code>reduce</code>	<code>Math.sqrt</code> , <code>console.log</code>



Problem-Solving Evolution: Interview Approach



Problem Statement

"Given an array of radius values, calculate the area for each radius and store in a new array."



Approach 1: Repetitive Code (Anti-Pattern)

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

// Calculate areas
const calculateArea = function (radius) {
    const output = [];
    for (let i = 0; i < radius.length; i++) {
        output.push(Math.PI * radius[i] *
radius[i]);
```

```

    }
    return output;
};

console.log(calculateArea(radius));
// Output: [3.14159, 12.56636, 28.27431,
50.26544]

```

🚫 Problems with this approach:

- Works fine for area calculation
- But what about circumference? diameter? volume?

✖ Approach 2: Code Duplication

```

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

// Calculate circumference - Notice the
// duplication!
const calculateCircumference = function
(radius) {
    const output = [];
    for (let i = 0; i < radius.length; i++) {
        output.push(2 * Math.PI * radius[i]); // Only this line differs!
    }
    return output;
};

console.log(calculateCircumference(radius));

```

```
// Output: [6.28318, 12.56636, 18.84954,  
25.13272]
```

🚫 Violates DRY Principle:

- Don't Repeat Yourself
 - 90% of code is identical
 - Hard to maintain and error-prone
-



Functional Programming Solution

⌚ Approach 3: Higher-Order Function Pattern

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

// 🔧 Pure functions for mathematical
operations
const area = function (radius) {
    return Math.PI * radius * radius;
};

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

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

```

// ⚪ Higher-Order Function: Generic
calculator

const calculate = function(radiusArr,
operation) {
    const output = [];
    for (let i = 0; i < radiusArr.length;
i++) {
        output.push(operation(radiusArr[i]));
    // Dynamic operation!
    }
    return output;
};

// 🚀 Usage - Same function, different
operations

console.log(calculate(radiusArr, area));
// Output: [3.14159, 12.56636, 28.27431,
50.26544]

console.log(calculate(radiusArr,
circumference));
// Output: [6.28318, 12.56636, 18.84954,
25.13272]

console.log(calculate(radiusArr, diameter));
// Output: [2, 4, 6, 8]

```

Benefits of This Approach:

1.  Reusability

- One `calculate` function handles all operations
- Add new operations without changing core logic

2. Separation of Concerns

- Data transformation logic → `calculate` function
- Mathematical operations → Individual pure functions
- Business logic → Stays separate

3. Modularity

```
// Easy to add new operations
const volume = (radius) => (4/3) * Math.PI *
radius * radius * radius;
console.log(calculate(radiusArr, volume));
```

4. Testability

```
// Each function can be tested independently
console.assert(area(1) === Math.PI, "Area
calculation test");
console.assert(diameter(5) === 10, "Diameter
calculation test");
```



Understanding the HOF Pattern



Function Roles Analysis

Function	Type	Role	Input	Output
calculate	Higher-Order	Data transformer	Array + Function	Array
area	Callback	Math operation	Number	Number
circumference	Callback	Math operation	Number	Number
diameter	Callback	Math operation	Number	Number



Execution Flow Visualization

```
radiusArr = [1, 2, 3, 4]
```

↓

```
calculate(radiusArr, area)
```

↓

```

| for each radius in array:
|   1. Call area(radius)
|   2. Push result to output
|   3. Return transformed array
| 
```

↓

```
[π×12, π×22, π×32, π×42]
```

↓

```
[3.14159, 12.56636, 28.27431, 50.26544]
```



Building Custom Polyfills

🧠 Connection to Native Methods

💡 Insight: Our `calculate` function is essentially a **polyfill** for JavaScript's native `map` method!

```
// Our custom function
console.log(calculate(radiusArr, area));

// Native map method (equivalent)
console.log(radiusArr.map(area));

// Both produce identical results! ✓
```



🛠️ Creating a Map Polyfill

```
// ⚡ Adding our custom method to Array
prototype
Array.prototype.calculate =
function(operation) {
    const output = [];
    for (let i = 0; i < this.length; i++) {
        output.push(operation(this[i]));
    }
    return output;
};
```

```
// 🚀 Usage - Now all arrays have our method!
console.log(radiusArr.calculate(area));
console.log(radiusArr.calculate(circumference));
console.log(radiusArr.calculate(diameter));
```

⚠ Polyfill Best Practices

✓ Production-Ready Polyfill:

```
// 🛡 Safe polyfill implementation
if (!Array.prototype.myMap) {
    Array.prototype.myMap =
        function(callback, thisArg) {
            // Input validation
            if (typeof callback !== 'function') {
                throw new TypeError(callback + ' is not a function');
            }

            const output = [];
            for (let i = 0; i < this.length; i++) {
                if (i in this) { // Skip holes in sparse arrays
                    output[i] =
                        callback.call(thisArg, this[i], i, this);
                }
            }
            return output;
        }
}
```

```
    };
}
```

 **Avoid in Production:**

```
//  Don't modify native prototypes directly
Array.prototype.calculate = function() { /* ... */ };

//  Better: Create utility functions
const mapArray = (arr, operation) => { /* ... */ };
```

Functional Programming Principles

Core Concepts Demonstrated

1. Higher-Order Functions

- Functions that operate on other functions
- Enable code reuse and abstraction
- Foundation of functional programming

2. Pure Functions

```
//  Pure function - same input, same
output, no side effects
```

```

const area = (radius) => Math.PI * radius *
radius;

// ✗ Impure function - depends on external
state

let multiplier = 2;
const impureArea = (radius) => Math.PI *
radius * radius * multiplier;

```

3. Function Composition

```

// Combining simple functions to create
complex behavior

const processRadius = (radiusArr) =>
  radiusArr
    .filter(r => r > 0) // Remove invalid radii
    .map(area) // Calculate areas
      .map(a => parseFloat(a.toFixed(2)));
    // Round to 2 decimals

console.log(processRadius([1, -2, 3, 0, 4]));
// Output: [3.14, 28.27, 50.27]

```

4. Immutability

```
// ✓ Original array unchanged
const original = [1, 2, 3, 4];
const areas = calculate(original, area);
console.log(original); // Still [1, 2, 3, 4]
console.log(areas);    // [3.14159, 12.56636,
28.27431, 50.26544]
```



Real-World HOF Applications



E-commerce Example

```
const products = [
  { name: 'Laptop', price: 1000, category:
  'Electronics' },
  { name: 'Book', price: 20, category:
  'Education' },
  { name: 'Phone', price: 800, category:
  'Electronics' }
];

// HOF for filtering
const filterBy = (array, predicate) =>
array.filter(predicate);

// HOF for transforming
const mapBy = (array, transformer) =>
```

```
array.map(transformer);

// Usage
const expensiveItems = filterBy(products, p => p.price > 500);
const productNames = mapBy(products, p => p.name);
const discountedPrices = mapBy(products, p => p.price * 0.9);
```

UI Event Handling

```
// HOF for event management
const createEventHandler = (callback) => {
    return function(event) {
        event.preventDefault();
        console.log(`Event ${event.type} handled`);
        callback(event);
    };
};

// Usage
const handleClick = createEventHandler((e) =>
{
    console.log('Button clicked!');
});

const handleSubmit = createEventHandler((e) => {
```

```
    console.log('Form submitted!');  
});
```

Data Processing Pipeline

```
const processData = (data, ...operations) =>  
{  
    return operations.reduce((result,  
operation) => {  
        return operation(result);  
    }, data);  
};  
  
// Usage  
const numbers = [1, 2, 3, 4, 5];  
const result = processData(  
    numbers,  
    arr => arr.filter(n => n % 2 === 0), //  
    Even numbers  
    arr => arr.map(n => n * n), //  
    Square them  
    arr => arr.reduce((sum, n) => sum + n, 0)  
    // Sum them  
);  
console.log(result); // 20 (22 + 42 = 4 + 16  
= 20)
```



HOF vs Traditional Programming



Comparison Analysis

Aspect	Traditional Approach	HOF Approach
Code Reuse	Copy-paste, modify	Write once, use anywhere
Maintainability	Hard to change	Easy to modify
Testing	Test everything together	Test parts independently
Readability	Procedural, verbose	Declarative, concise
Bugs	More prone to errors	Fewer logical errors



Performance Considerations

✓ HOF Benefits:

- **Smaller bundle size** → Less code duplication
- **Better optimization** → Browser engines optimize HOFs
- **Lazy evaluation** → Operations only when needed



⚠ HOF Considerations:

- **Function call overhead** → Minimal in modern engines
- **Memory usage** → Creating many functions
- **Debugging complexity** → Requires understanding composition



Advanced HOF Patterns



Currying with HOFs

```
// Higher-order function that returns
configured functions
const createCalculator = (operation) => {
    return (array) => array.map(operation);
};

// Create specialized calculators
const calculateAreas =
createCalculator(area);
const calculateCircumferences =
createCalculator(circumference);

// Usage
console.log(calculateAreas([1, 2, 3]));
console.log(calculateCircumferences([1, 2,
3]));
```

Function Composition HOF

```
const compose = (...functions) => {
    return (value) =>
functions.reduceRight((acc, fn) => fn(acc),
value);
};

// Create complex operations
const processRadius = compose(
    arr => arr.map(x =>
```

```

    parseFloat(x.toFixed(2))),
    arr => arr.map(area),
    arr => arr.filter(r => r > 0)
);

console.log(processRadius([-1, 1, 2, 3]));

```

Memoization HOF

```

const memoize = (fn) => {
  const cache = new Map();
  return function(...args) {
    const key = JSON.stringify(args);
    if (cache.has(key)) {
      return cache.get(key);
    }
    const result = fn.apply(this, args);
    cache.set(key, result);
    return result;
  };
};

// Memoized expensive operations
const memoizedArea = memoize(area);

```



Quick Summary

Key Takeaways:

1. Higher-Order Functions

- **Accept functions** as parameters or return functions
- **Enable abstraction** and code reuse
- **Foundation** of functional programming

2. Problem-Solving Evolution

- **Start simple** → Identify patterns → **Abstract common logic**
- **DRY principle** → Don't repeat yourself
- **Separate concerns** → Data transformation vs business logic

3. Practical Benefits

- **Reusable code** → Write once, use everywhere
- **Maintainable** → Easy to modify and extend
- **Testable** → Independent, pure functions
- **Readable** → Declarative over imperative

4. Native Methods Understanding

- **map, filter, reduce** are built-in HOFs
- **Understanding polyfills** helps grasp internal workings
- **Custom implementations** provide deeper insight

Quick Memory Aid:

HOF = Function that takes/returns functions

Pattern = Data + Operation → Transformation

Benefits = Reusable + Maintainable + Testable

Examples = map, filter, reduce, forEach

Principle = Separate "what to do" from "how to do"

🎯 Real-World Applications:

-  **Data processing** → Transform arrays and objects
-  **Event handling** → Create reusable event managers
-  **API calls** → Generic request/response handlers
-  **Form validation** → Composable validation rules
-  **Game logic** → Configurable behavior systems

⚡ Interview Tips:

- **Start with simple solution** → Show working code first
- **Identify problems** → Point out code duplication
- **Refactor to HOF** → Demonstrate functional thinking
- **Explain benefits** → Reusability, maintainability, testability
- **Show native equivalent** → Connect to built-in methods

Watch Live On Youtube below:



Episode 19: map, filter & reduce

🎯 What You'll Learn

- Master the three essential array methods: map, filter, and reduce
 - Understand when and how to use each Higher-Order Function
 - Transform data efficiently using functional programming patterns
 - Chain array methods for complex data transformations
 - Build real-world applications with array manipulation
 - Compare functional vs imperative programming approaches
-

The Power Trio of Array Methods

map, filter & reduce are Higher-Order Functions that form the foundation of functional programming in JavaScript.

Quick Overview Table

Method	Purpose	Input	Output	Use Case
<code>map</code>	Transform each element	Array	New Array (same length)	Convert data format
<code>filter</code>	Select elements by condition	Array	New Array (\leq length)	Remove unwanted items
<code>reduce</code>	Combine all elements	Array	Single Value	Aggregate calculations

Map Function: Data Transformation

What is Map?

Map is used to transform an array. The `map()` method creates a new array with the results of calling a function for every array element.

```
const output = arr.map(function)
// The function tells map what transformation
```

to apply on each element

Map Function Characteristics

Aspect	Description
Immutability	Original array remains unchanged
1:1 Mapping	Each element produces exactly one result
Same Length	Output array has same length as input
Pure Function	No side effects, predictable results

Practical Examples

Example 1: Double the Numbers

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

// Task: Double each element → [10, 2, 6, 4, 12]
function double(x) {
    return x * 2;
}

const doubleArr = arr.map(double);
console.log(doubleArr); // [10, 2, 6, 4, 12]
console.log(arr);      // [5, 1, 3, 2, 6] -
Original unchanged ✓
```

Example 2: Triple the Numbers

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

// Transformation logic
function triple(x) {
    return x * 3;
}

const tripleArr = arr.map(triple);
console.log(tripleArr); // [15, 3, 9, 6, 18]
```

Example 3: Convert to Binary

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

// ⚪ Multiple ways to write the same
transformation:

// Method 1: Named function
function binary(x) {
    return x.toString(2);
}

const binaryArr = arr.map(binary);

// Method 2: Inline function
const binaryArr = arr.map(function(x) {
    return x.toString(2);
});
```

```
// Method 3: Arrow function (most concise)
const binaryArr = arr.map(x =>
x.toString(2));

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

Map Internals: How It Works

```
// 🔎 Conceptual implementation of map
Array.prototype.myMap = function(callback) {
    const result = [];
    for (let i = 0; i < this.length; i++) {
        result.push(callback(this[i], i,
this)); // value, index, array
    }
    return result;
};
```

Filter Function: Selective Filtering

What is Filter?

Filter is used to select elements from an array. The `filter()` method creates a new array with all elements that pass a test implemented by the provided function.

```
const filteredArray =
arr.filter(conditionFunction)
// Returns only elements where
conditionFunction returns true
```

🎯 Filter Characteristics

Aspect	Description
Selective	Only elements passing the test are included
Boolean Logic	Callback must return truthy/falsy values
Smaller/Equal Length	Output length ≤ input length
Immutable	Original array remains unchanged

💡 Practical Examples

Example 1: Filter Odd Numbers

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

// Filter odd values
function isOdd(x) {
    return x % 2; // Returns 1 (truthy) for
odd, 0 (falsy) for even
}

const oddArr = array.filter(isOdd);
console.log(oddArr); // [5, 1, 3]
```

```
// Arrow function version
const oddArr = array.filter(x => x % 2);
```

Example 2: Filter by Condition

```
const numbers = [1, 4, 9, 16, 25, 36];

// Get numbers greater than 10
const largeNumbers = numbers.filter(num =>
num > 10);
console.log(largeNumbers); // [16, 25, 36]

// Get perfect squares under 20
const smallSquares = numbers.filter(num =>
num < 20);
console.log(smallSquares); // [1, 4, 9, 16]
```

🔍 Filter Truth Table

Element	Condition	Result	Included?
5	$5 \% 2 = 1$	Truthy	✓ Yes
1	$1 \% 2 = 1$	Truthy	✓ Yes
3	$3 \% 2 = 1$	Truthy	✓ Yes
2	$2 \% 2 = 0$	Falsy	✗ No
6	$6 \% 2 = 0$	Falsy	✗ No

Reduce Function: Data Aggregation

What is Reduce?

Reduce takes all values of an array and produces a single output. It "reduces" the array to give a consolidated result.

```
const result =
arr.reduce(function(accumulator, current,
index, array) {
    // accumulator: accumulated result from
    // previous iterations
    // current: current element being
    // processed
    // index: current element's index
    // (optional)
    // array: the array being processed
    // (optional)
    return newAccumulatorValue;
}, initialValue);
```

Reduce Characteristics

Aspect	Description
Aggregation	Combines multiple values into one
Accumulator	Carries forward the result from each iteration

Flexible Output	Can return any data type
Initial Value	Starting point for accumulator

Practical Examples

Example 1: Sum of Array Elements

```

const array = [5, 1, 3, 2, 6];

// ✗ Non-functional programming way
function findSum(arr) {
    let sum = 0;
    for (let i = 0; i < arr.length; i++) {
        sum = sum + arr[i];
    }
    return sum;
}
console.log(findSum(array)); // 17

// ✅ Functional programming way with reduce
const sumOfElem = array.reduce(function
(accumulator, current) {
    // accumulator represents the running total
    // (like 'sum' variable above)
    // current represents the current array
    // element (like 'arr[i]' above)
    accumulator = accumulator + current;
    return accumulator;
}, 0); // Initial value: 0 (like sum = 0
above)

```

```
console.log(sumOfElem); // 17
```

Example 2: Find Maximum Value

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

// ✗ Imperative approach
function findMax(arr) {
    let max = 0;
    for(let i = 0; i < arr.length; i++) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }
    return max;
}
console.log(findMax(array)); // 6
```

```
// ✓ Functional approach with reduce
const maxValue = array.reduce((acc, current)
=> {
    if (current > acc) {
        acc = current;
    }
    return acc;
}, 0);
console.log(maxValue); // 6
```

// ⚡ Even cleaner version with meaningful

```

variable names
const maxValue = array.reduce((max, current)
=> {
    if (current > max) {
        max = current;
    }
    return max;
}, 0);
console.log(maxValue); // 6

```

Reduce Execution Flow

```

// Step-by-step execution for sum calculation
const array = [5, 1, 3, 2, 6];

/*
Iteration 1: accumulator = 0, current = 5 →
return 0 + 5 = 5
Iteration 2: accumulator = 5, current = 1 →
return 5 + 1 = 6
Iteration 3: accumulator = 6, current = 3 →
return 6 + 3 = 9
Iteration 4: accumulator = 9, current = 2 →
return 9 + 2 = 11
Iteration 5: accumulator = 11, current = 6 →
return 11 + 6 = 17
Final result: 17
*/

```

🎯 Real-World Examples: Working with Objects

👤 User Data Processing

```
const users = [
    { firstName: "Alok", lastName: "Raj",
age: 23 },
    { firstName: "Ashish", lastName: "Kumar",
age: 29 },
    { firstName: "Ankit", lastName: "Roy",
age: 29 },
    { firstName: "Pranav", lastName:
"Mukherjee", age: 50 },
];
```

🗺 Map Example: Get Full Names

```
// Task: Get array of full names → ["Alok
Raj", "Ashish Kumar", ...]
const fullNameArr = users.map(user =>
user.firstName + " " + user.lastName);
console.log(fullNameArr);
// Output: ["Alok Raj", "Ashish Kumar",
"Ankit Roy", "Pranav Mukherjee"]

// Alternative approaches:
const fullNameArr = users.map(user =>
```

```
`${user.firstName} ${user.lastName}`);
const fullNameArr = users.map(({firstName,
lastName}) => `.${firstName} ${lastName}`);
```

⚡ Reduce Example: Age Distribution Report

```
// Task: Get count of people by age → {23: 1,
29: 2, 50: 1}
const ageReport = users.reduce((acc, curr) =>
{
    if (acc[curr.age]) {
        acc[curr.age] = acc[curr.age] + 1; // Increment existing count
    } else {
        acc[curr.age] = 1; // Initialize count for new age
    }
    return acc; // Always return updated accumulator
}, {}); // Start with empty object

console.log(ageReport); // {23: 1, 29: 2, 50: 1}

// ⚡ Cleaner version using logical OR
const ageReport = users.reduce((acc, curr) =>
{
    acc[curr.age] = (acc[curr.age] || 0) + 1;
```

```
    return acc;
}, {});
```

Step-by-Step Reduce Execution

```
// How the age report builds up:
/*
Initial: acc = {}
User 1 (Alok, 23): acc = {23: 1}
User 2 (Ashish, 29): acc = {23: 1, 29: 1}
User 3 (Ankit, 29): acc = {23: 1, 29: 2}
User 4 (Pranav, 50): acc = {23: 1, 29: 2, 50: 1}
Final result: {23: 1, 29: 2, 50: 1}
*/
```

Function Chaining: The Power of Composition

Combining Multiple Operations

Function chaining allows you to **combine multiple array methods** to create powerful data processing pipelines.

```
const users = [
  { firstName: "Alok", lastName: "Raj", age:
```

```

23 },
  { firstName: "Ashish", lastName: "Kumar",
age: 29 },
  { firstName: "Ankit", lastName: "Roy", age:
29 },
  { firstName: "Pranav", lastName:
"Mukherjee", age: 50 },
];
// Task: Get first names of people under 30
const youngUserNames = users
  .filter(user => user.age < 30)      // Step
1: Filter young users
  .map(user => user.firstName);        // Step
2: Extract first names

console.log(youngUserNames); // ["Alok",
"Ashish", "Ankit"]

```

Chain Execution Breakdown

```

// Step-by-step execution:

// Original array:
[
  { firstName: "Alok", lastName: "Raj", age:
23 },
  { firstName: "Ashish", lastName: "Kumar",
age: 29 },
  { firstName: "Ankit", lastName: "Roy", age:

```

```

29 },
  { firstName: "Pranav", lastName:
"Mukherjee", age: 50 }
]

// After .filter(user => user.age < 30):
[
  { firstName: "Alok", lastName: "Raj", age:
23 },
  { firstName: "Ashish", lastName: "Kumar",
age: 29 },
  { firstName: "Ankit", lastName: "Roy", age:
29 }
]

// After .map(user => user.firstName):
["Alok", "Ashish", "Ankit"]

```

🏆 Homework Challenge: Using Reduce Only

```

// Challenge: Implement the same logic using
only reduce
const youngUserNames = users.reduce((acc,
curr) => {
  if (curr.age < 30) {
    acc.push(curr.firstName);
  }
  return acc;
}, []);

```

```
console.log(youngUserNames); // ["Alok",  
"Ashish", "Ankit"]
```



Advanced Patterns and Techniques

🎯 Complex Transformations

Multi-step Processing Pipeline

```
const salesData = [  
  { product: "Laptop", price: 1200, quantity:  
 2, category: "Electronics" },  
  { product: "Mouse", price: 25, quantity:  
10, category: "Electronics" },  
  { product: "Book", price: 15, quantity: 5,  
category: "Education" },  
  { product: "Phone", price: 800, quantity:  
3, category: "Electronics" }  
];  
  
// Get total revenue from Electronics with  
quantity > 5  
const electronicsRevenue = salesData  
  .filter(item => item.category ===  
"Electronics")  
  .filter(item => item.quantity > 5)  
  .map(item => ({ ...item, revenue:  
item.price * item.quantity }))  
  .reduce((total, item) => total +
```

```
item.revenue, 0);

console.log(electronicsRevenue); // 250
(Mouse: 25 * 10)
```

Data Aggregation with Grouping

```
// Group products by category with total
revenue
const categoryReport = salesData.reduce((acc,
item) => {
    const category = item.category;
    const revenue = item.price * item.quantity;

    if (!acc[category]) {
        acc[category] = { count: 0, totalRevenue:
0, products: [] };
    }

    acc[category].count++;
    acc[category].totalRevenue += revenue;
    acc[category].products.push(item.product);

    return acc;
}, {});

console.log(categoryReport);
/*
{
    Electronics: { count: 3, totalRevenue:
```

```

4850, products: ["Laptop", "Mouse", "Phone"]
},
Education: { count: 1, totalRevenue: 75,
products: ["Book"] }
}
*/

```

Functional vs Imperative Comparison

Approach	Readability	Performance	Immutability	Debugging
Functional	High	Good	 Preserved	Method-by-method
Imperative	Medium	Better	 Mutates	Step-by-step

Performance Considerations

```

// Large datasets: Consider performance
trade-offs
const largeArray = Array.from({length:
1000000}, (_, i) => i);

//  Multiple iterations (3 passes through
data)
const result1 = largeArray
  .filter(n => n % 2 === 0)
  .map(n => n * 2)
  .reduce((sum, n) => sum + n, 0);

```

```
// ✓ Single iteration with reduce
const result2 = largeArray.reduce((sum, n) =>
{
  if (n % 2 === 0) {
    return sum + (n * 2);
  }
  return sum;
}, 0);
```



Building Custom Implementations



Understanding Internals

Custom Map Implementation

```
Array.prototype.myMap = function(callback,
thisArg) {
  if (typeof callback !== 'function') {
    throw new TypeError(callback + ' is not a
function');
  }

  const result = [];
  for (let i = 0; i < this.length; i++) {
    if (i in this) { // Handle sparse arrays
      result[i] = callback.call(thisArg,
this[i], i, this);
    }
  }
}
```

```
    return result;  
};
```

Custom Filter Implementation

```
Array.prototype.myFilter = function(callback,  
thisArg) {  
    const result = [];  
    for (let i = 0; i < this.length; i++) {  
        if (i in this && callback.call(thisArg,  
this[i], i, this)) {  
            result.push(this[i]);  
        }  
    }  
    return result;  
};
```

Custom Reduce Implementation

```
Array.prototype.myReduce = function(callback,  
initialValue) {  
    let hasInitial = arguments.length > 1;  
    let accumulator = hasInitial ? initialValue  
: this[0];  
    let startIndex = hasInitial ? 0 : 1;  
  
    for (let i = startIndex; i < this.length;  
i++) {
```

```

if (i in this) {
    accumulator = callback(accumulator,
this[i], i, this);
}
return accumulator;
};

```



Common Pitfalls and Best Practices

🚫 Common Mistakes

1. Mutating Original Array

```

// ❌ Wrong: Mutating during map
const numbers = [1, 2, 3];
const doubled = numbers.map((num, index, arr)
=> {
    arr[index] = num * 2; // Don't do this!
    return num * 2;
});

// ✅ Correct: Pure transformation
const doubled = numbers.map(num => num * 2);

```

2. Not Returning from Reduce

```
// ❌ Wrong: Forgetting to return accumulator
const sum = numbers.reduce((acc, num) => {
    acc + num; // Missing return!
});
```

```
// ✅ Correct: Always return accumulator
const sum = numbers.reduce((acc, num) => acc
+ num, 0);
```

3. Forgetting Initial Value in Reduce

```
// ❌ Dangerous: No initial value with empty
array
const emptyArray = [];
const sum = emptyArray.reduce((acc, num) =>
acc + num); // TypeError!
```

```
// ✅ Safe: Always provide initial value
const sum = emptyArray.reduce((acc, num) =>
acc + num, 0); // Returns 0
```

✓ Best Practices

1. Use Method Chaining Thoughtfully

```
// ✅ Good: Logical flow, easy to read
const result = data
```

```

    .filter(isValid)          // Remove invalid
items
    .map(transform)           // Transform valid
items
    .reduce(aggregate, 0);   // Aggregate results

// ✗ Avoid: Too many steps, hard to debug
const result = data

.filter(a).map(b).filter(c).map(d).filter(e).rec
0);

```

2. Prefer Descriptive Function Names

```

// ✅ Good: Clear intent
const adults = users.filter(isAdult);
const fullNames = users.map(getFullName);

// ✗ Unclear: Anonymous functions everywhere
const adults = users.filter(u => u.age >=
18);

```

3. Handle Edge Cases

```

// ✅ Robust: Handle undefined/null values
const safeDivide = numbers.map(num => num !==
0 ? 100 / num : 0);

```

```
const validUsers = users.filter(user => user  
&& user.age && user.name);
```



Quick Summary

Key Takeaways:

1. Map - Transformation

- **Purpose:** Transform each element into something new
- **Length:** Output has same length as input
- **Use Case:** Convert data format, apply calculations
- **Returns:** New array with transformed elements

2. Filter - Selection

- **Purpose:** Select elements that meet a condition
- **Length:** Output length \leq input length
- **Use Case:** Remove unwanted elements, find specific items
- **Returns:** New array with filtered elements

3. Reduce - Aggregation

- **Purpose:** Combine all elements into single value
- **Length:** Always returns one value (any type)
- **Use Case:** Sum, average, grouping, complex aggregations
- **Returns:** Single accumulated value

4. Function Chaining

- **Combine methods** for complex data processing

- **Read left-to-right** like a pipeline
- **Each method** receives output from previous method
- **Powerful pattern** for data transformation

Quick Memory Aid:

Map = Transform (1:1)
 Filter = Select (1:0 or 1:1)
 Reduce = Aggregate (N:1)
 Chain = Pipeline (method1 → method2 → method3)
 All = Higher-Order Functions (take functions as arguments)

Method Selection Guide:

Need to...	Use	Example
Transform all elements	<code>map</code>	Convert strings to uppercase
Find matching elements	<code>filter</code>	Get users over 18
Calculate single result	<code>reduce</code>	Sum of numbers
Complex processing	Chain	Filter then transform then sum

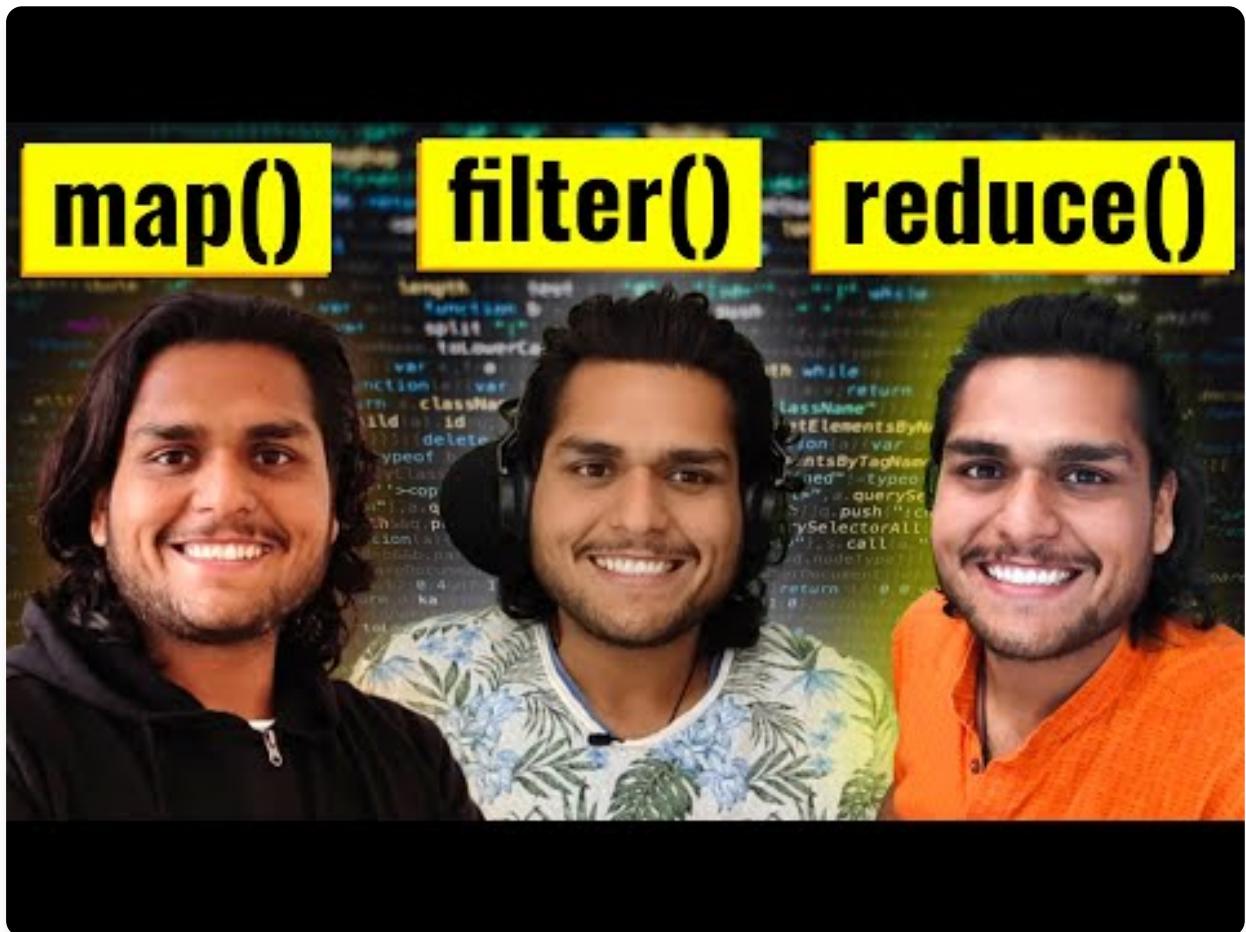
Real-World Applications:

-  **Data visualization** → Transform API data for charts
-  **E-commerce** → Filter products, calculate totals
-  **Form validation** → Check fields, transform inputs
-  **UI updates** → Process user lists, generate components
-  **Analytics** → Aggregate user behavior data

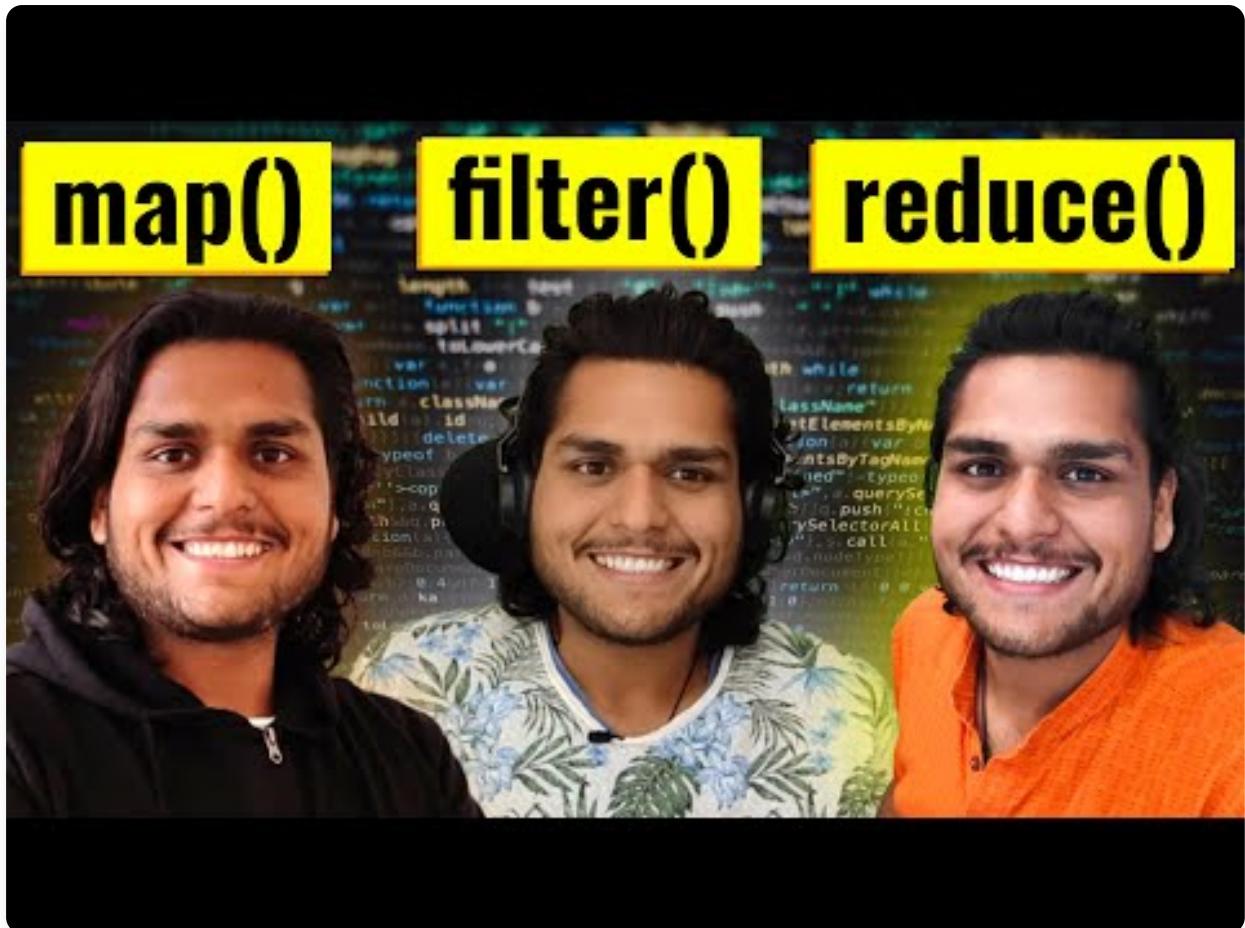
⚡ Performance Tips:

- Single reduce is often faster than multiple chained methods
 - Consider data size when choosing between approaches
 - Use early returns in filter conditions when possible
 - Profile your code with large datasets
-

🎥 Watch the Video



Watch Live On Youtube below:



Episode 20 : Callback

- There are 2 Parts of Callback:
 1. Good Part of callback - Callback are super important while writing asynchronous code in JS
 2. Bad Part of Callback - Using callback we can face issue:
 - Callback Hell
 - Inversion of control
- Understanding of Bad part of callback is super important to learn Promise in next lecture.

 *JavaScript is synchronous, single threaded language. It can Just do one thing at a time, it has just one call-stack and it can execute one thing at a time. Whatever code we give to Javascript will be quickly executed by Javascript engine, it does not wait.*

```
console.log("Namaste");
console.log("JavaScript");
console.log("Season 2");
// Namaste
// JavaScript
// Season 2

//  It is quickly printing because `Time, tide & Javascript waits for none.`
```

But what if we have to delay execution of any line, we could utilize callback, How?

```
console.log("Namaste");
setTimeout(function () {
    console.log("JavaScript");
}, 5000);
console.log("Season 2");
// Namaste
// Season 2
// JavaScript
```

//💡 Here we are delaying the execution using callback approach of setTimeout.

e-Commerce web app situation

Assume a scenario of e-Commerce web, where one user is placing order, he has added items like, shoes, pants and kurta in cart and now he is placing order. So in backend the situation could look something like this.

```
const cart = ["shoes", "pants", "kurta"];
// Two steps to place a order
// 1. Create a Order
// 2. Proceed to Payment

// It could look something like this:
api.createOrder();
api.proceedToPayment();
```

Assumption, once order is created then only we can proceed to payment, so there is a dependency. So How to manage this dependency. Callback can come as rescue, How?

```
api.createOrder(cart, function () {
    api.proceedToPayment();
});
//💡 Over here `createOrder` api is first creating a order then it is responsible to
```

```
call `api.proceedToPayment()` as part of  
callback approach.
```

To make it a bit complicated, what if after payment is done, you have to show Order summary by calling `api.showOrderSummary()` and now it has dependency on `api.proceedToPayment()` Now my code should look something like this:

```
api.createOrder(cart, function () {  
    api.proceedToPayment(function () {  
        api.showOrderSummary();  
    });  
});
```

Now what if we have to update the wallet, now this will have a dependency over `showOrderSummary`

```
api.createOrder(cart, function () {  
    api.proceedToPayment(function () {  
        api.showOrderSummary(function () {  
            api.updateWallet();  
        });  
    });  
});  
//💡 Callback Hell
```

When we have a large codebase and multiple apis and have dependency on each other, then we fall into callback hell. These codes are tough to maintain. These callback hell structure is also known as **Pyramid of Doom**.

Till this point we are comfortable with concept of callback hell but now lets discuss about **Inversion of Control**. It is very important to understand in order to get comfortable around the concept of promise.

 *Inversion of control is like that you lose the control of code when we are using callback.*

Let's understand with the help of example code and comments:

```
api.createOrder(cart, function () {
    api.proceedToPayment();
});

//💡 So over here, we are creating a order
//and then we are blindly trusting
//`createOrder` to call `proceedToPayment`.

//💡 It is risky, as `proceedToPayment` is
//important part of code and we are blindly
//trusting `createOrder` to call it and handle
//it.

//💡 When we pass a function as a callback,
//basically we are dependant on our parent
//function that it is his responsibility to run
//that function. This is called `inversion of
//control` because we are dependant on that
//function. What if parent function stopped
```

working, what if it was developed by another programmer or callback runs two times or never run at all.

//💡 In next session, we will see how we can fix such problems.

💡 *Async programming in JavaScript exists because callback exits.*

more at <http://callbackhell.com/>

Watch Live On Youtube below:



Episode 21 : Promises

Promises are used to handle async operations in JavaScript.

We will discuss with code example that how things used to work before **Promises** and then how it works after **Promises**

Suppose, taking an example of E-Commerce

```
const cart = ["shoes", "pants", "kurta"];  
  
// Below two functions are asynchronous and  
// dependent on each other  
const orderId = createOrder(cart);  
proceedToPayment(orderId);  
  
// with Callback (Before Promise)  
// Below here, it is the responsibility of  
// createOrder function to first create the  
// order then call the callback function  
createOrder(cart, function () {  
    proceedToPayment(orderId);  
});  
// Above there is the issue of `Inversion of  
// Control`
```

Q: How to fix the above issue?

A: *Using Promise.*

Now, we will make `createOrder` function return a promise and we will capture that `promise` into a `variable`

Promise is nothing but we can assume it to be empty object with some data value in it, and this data value will hold whatever this `createOrder` function will return.

Since `createOrder` function is an async function and we don't know how much time will it take to finish execution.

So the moment `createOrder` will get executed, it will return you a `undefined` value. Let's say after 5 secs execution finished so now `orderId` is ready so, it will fill the `undefined` value with the `orderId`.

In short, When `createOrder` get executed, it immediately returns a `promise object` with `undefined` value. then javascript will continue to execute with other lines of code. After sometime when `createOrder` has finished execution and `orderId` is ready then that will `automatically` be assigned to our returned `promise` which was earlier `undefined`.

Q: Question is how we will get to know `response` is ready?

A: So, we will attach a `callback` function to the `promise object` using `then` to get triggered automatically when `result` is ready.

```
const cart = ["shoes", "pants", "kurta"];

const promiseRef = createOrder(cart);
// this promiseRef has access to `then`

// {data: undefined}
// Initially it will be undefined so below
```

```
code won't trigger
// After some time, when execution has
finished and promiseRef has the data then
automatically the below line will get
triggered.
```

```
promiseRef.then(function () {
  proceedToPayment(orderId);
});
```

Q: How it is better than callback approach?

In Earlier solution we used to pass the function and then used to trust the function to execute the callback.

But with promise, we are attaching a callback function to a promiseObject.

There is difference between these words, passing a function and attaching a function.

Promise guarantee, it will callback the attached function once it has the fulfilled data. And it will call it only once. Just once.

Earlier we talked about promise are object with empty data but that's not entirely true, **Promise** are much more than that.

Now let's understand and see a real promise object.

fetch is a web-api which is utilized to make api call and it returns a promise.

We will be calling public github api to fetch data

<https://api.github.com/users/alok722>

```
// We will be calling public github api to
fetch data
```

```

const URL =
"https://api.github.com/users/alok722";
const user = fetch(URL);
// User above will be a promise.
console.log(user); // Promise {<Pending>}

/** OBSERVATIONS:
 * If we will deep dive and see, this
`promise` object has 3 things
 * `prototype`, `promiseState` &
`promiseResult`
 * & this `promiseResult` is the same data
which we talked earlier as data
 * & initially `promiseResult` is `undefined`
 *
 * `promiseResult` will store data returned
from API call
 * `promiseState` will tell in which state
the promise is currently, initially it will
be in `pending` state and later it will
become `fulfilled`
 */

/** 
 * When above line is executed, `fetch` makes
API call and return a `promise` instantly
which is in `Pending` state and Javascript
doesn't wait to get it `fulfilled`
 * And in next line it console out the
`pending promise`.
 * NOTE: chrome browser has some in-
consistency, the moment console happens it

```

shows in pending state but if you will expand that it will show fulfilled because chrome updated the log when promise get fulfilled.

- * Once fulfilled data is there in promiseResult and it is inside body in ReadableStream format and there is a way to extract data.

*/

Now we can attach callback to above response?

Using `.then`

```
const URL =
"https://api.github.com/users/alok722";
const user = fetch(URL);

user.then(function (data) {
  console.log(data);
});

// And this is how Promise is used.
// It guarantees that it could be resolved
// only once, either it could be `success` or
// `failure`
/**
```

A Promise is in one of these states:

`pending`: initial state, neither fulfilled nor rejected.

`fulfilled`: meaning that the operation was completed successfully.

rejected: meaning that the operation failed.

*/

 Promise Object are immutable.

→ Once promise is fulfilled and we have data we can pass here and there and we don't have to worry that someone can mutate that data. So over above we can't directly mutate `user` promise object, we will have to use `.then`

Interview Guide

 What is Promise?

→ Promise object is a placeholder for certain period of time until we receive value from asynchronous operation.

→ A container for a future value.

→ **A Promise is an object representing the eventual completion or failure of an asynchronous operation.**

We are now done solving one issue of callback i.e. Inversion of Control

But there is one more issue, callback hell...

```
// Callback Hell Example
createOrder(cart, function (orderId) {
    proceedToPayment(orderId, function
(paymentInf) {
        showOrderSummary(paymentInf, function
(balance) {
            updateWalletBalance(balance);
        });
    });
});
```

```

});

// And now above code is expanding
// horizontally and this is called pyramid of
// doom.

// Callback hell is ugly and hard to
// maintain.

//💡 Promise fixes this issue too using
`Promise Chaining`
// Example Below is a Promise Chaining
createOrder(cart)
  .then(function (orderId) {
    proceedToPayment(orderId);
  })
  .then(function (paymentInf) {
    showOrderSummary(paymentInf);
  })
  .then(function (balance) {
    updateWalletBalance(balance);
  });

//⚠ Common PitFall
// We forget to return promise in Promise
// Chaining
// The idea is promise/data returned from one
// .then become data for next .then
// So,
createOrder(cart)
  .then(function (orderId) {
    return proceedToPayment(orderId);
  })
  .then(function (paymentInf) {

```

```
        return showOrderSummary(paymentInf);
    })
    .then(function (balance) {
        return updateWalletBalance(balance);
    });

// To improve readability you can use arrow
function instead of regular function
```

Watch Live On Youtube below:



Episode 22 : Creating a Promise, Chaining & Error

Handling

```
const cart = ["shoes", "pants", "kurta"];  
  
// Consumer part of promise  
const promise = createOrder(cart); // orderId  
// Our expectation is above function is going  
to return me a promise.  
  
promise.then(function (orderId) {  
    proceedToPayment(orderId);  
});  
  
// Above snippet we have observed in our  
previous lecture itself.  
// Now we will see, how createOrder is  
implemented so that it is returning a promise  
// In short we will see, "How we can create  
Promise" and then return it.  
  
// Producer part of Promise  
function createOrder(cart) {  
    // JS provides a Promise constructor  
    through which we can create promise  
    // It accepts a callback function with two  
    parameter `resolve` & `reject`  
    const promise = new Promise(function  
(resolve, reject) {  
        // What is this `resolve` and `reject`?
```

```

    // These are function which are passed by
    javascript to us in order to handle success
    and failure of function call.

    // Now we will write logic to
    `createOrder` 

    /** Mock logic steps
     * 1. validateCart
     * 2. Insert in DB and get an orderId
     */

    // We are assuming in real world
    scenario, validateCart would be defined

    if (!validateCart(cart)) {
        // If cart not valid, reject the
        promise

        const err = new Error("Cart is not
Valid");
        reject(err);
    }

    const orderId = "12345"; // We got this
    id by calling to db (Assumption)
    if (orderId) {
        // Success scenario
        resolve(orderId);
    }
});

return promise;
}

```

Over above, if your validateCart is returning true, so the above promise will be resolved (success),

```

const cart = ["shoes", "pants", "kurta"];

const promise = createOrder(cart); // orderId
// ? What will be printed in below line?
// It prints Promise {<pending>}, but why?
// Because above createOrder is going to take
// sometime to get resolved, so pending state.
// But once the promise is resolved, ` `.then` 
// would be executed for callback.
console.log(promise);

promise.then(function (orderId) {
    proceedToPayment(orderId);
});

function createOrder(cart) {
    const promise = new Promise(function
(resolve, reject) {
        if (!validateCart(cart)) {
            const err = new Error("Cart is not
Valid");
            reject(err);
        }
        const orderId = "12345";
        if (orderId) {
            resolve(orderId);
        }
    });
    return promise;
}

```

Now let's see if there was some error and we are rejecting the promise, how we could catch that?

→ Using `.catch`

```
const cart = ["shoes", "pants", "kurta"];

const promise = createOrder(cart); // orderId

// Here we are consuming Promise and will try
// to catch promise error
promise
  .then(function (orderId) {
    // ✅ success aka resolved promise
    // handling
    proceedToPayment(orderId);
  })
  .catch(function (err) {
    // ⚠️ failure aka reject handling
    console.log(err);
});

// Here we are creating Promise
function createOrder(cart) {
  const promise = new Promise(function
(resolve, reject) {
    // Assume below `validateCart` return
    // false then the promise will be rejected
    // And then our browser is going to throw
    // the error.
    if (!validateCart(cart)) {
      const err = new Error("Cart is not
        valid");
      reject(err);
    } else {
      resolve("Order placed successfully!");
    }
  });
  return promise;
}
```

```

    Valid");
        reject(err);
    }
    const orderId = "12345";
    if (orderId) {
        resolve(orderId);
    }
});
return promise;
}

```

Now, Let's understand the concept of Promise Chaining

→ for this we will assume after `createOrder` we have to invoke

`proceedToPayment`

→ In promise chaining, whatever is returned from first `.then` become data for next `.then` and so on...

→ At any point of promise chaining, if promise is rejected, the execution will fallback to `.catch` and others promise won't run.

```

const cart = ["shoes", "pants", "kurta"];

createOrder(cart)
    .then(function (orderId) {
        // ✅ success aka resolved promise
        handling
            //💡 we have return data or promise so
            // that we can keep chaining the promises, here
            // we are returning data
        console.log(orderId);
        return orderId;
    })
    .catch(function (err) {
        // handle error
    })

```

```

    })
    .then(function (orderId) {
        // Promise chaining
        //💡 we will make sure that
        `proceedToPayment` returns a promise too
        return proceedToPayment(orderId);
    })
    .then(function (paymentInfo) {
        // from above, `proceedToPayment` is
        returning a promise so we can consume using
        `.then`
        console.log(paymentInfo);
    })
    .catch(function (err) {
        //⚠️ failure aka reject handling
        console.log(err);
    });
}

// Here we are creating Promise
function createOrder(cart) {
    const promise = new Promise(function
(resolve, reject) {
        // Assume below `validateCart` return
        false then the promise will be rejected
        // And then our browser is going to throw
        the error.
        if (!validateCart(cart)) {
            const err = new Error("Cart is not
Valid");
            reject(err);
        }
        const orderId = "12345";
    })
}

```

```

if (orderId) {
    resolve(orderId);
}
});

return promise;
}

function proceedToPayment(cart) {
    return new Promise(function (resolve,
reject) {
        // For time being, we are simply
`resolving` promise
        resolve("Payment Successful");
    });
}

```

Q: What if we want to continue execution even if any of my promise is failing, how to achieve this?

- By placing the `.catch` block at some level after which we are not concerned with failure.
- There could be multiple `.catch` too. Eg:

```

createOrder(cart)
    .then(function (orderId) {
        // ✓ success aka resolved promise
        handling
        //💡 we have return data or promise so
        that we can keep chaining the promises, here
        we are returning data
        console.log(orderId);
    })

```

```
    return orderId;
})
  .catch(function (err) {
    // ⚠️ Whatever fails below it, catch wont
    care
      // this block is responsible for code
    block above it.
    console.log(err);
});
.then(function (orderId) {
  // Promise chaining
  //💡 we will make sure that
`proceedToPayment` returns a promise too
  return proceedToPayment(orderId);
})
.then(function (paymentInfo) {
  // from above, `proceedToPayment` is
  returning a promise so we can consume using
  `.then`
  console.log(paymentInfo);
})
```

Watch Live On Youtube below:



Episode 23 : async await

Topics Covered

- What is async?
- What is await?
- How async await works behind the scenes?
- Example of using async/await
- Error Handling
- Interviews
- Async await vs Promise.then/.catch

Q: What is async?

A: Async is a keyword that is used before a function to create a async function.

Q: What is async function and how it is different from normal function?

```
//💡 async function always returns a
promise, even if I return a simple string
from below function, async keyword will wrap
it under Promise and then return.

async function getData() {
    return "Namaste JavaScript";
}

const dataPromise = getData();
console.log(dataPromise); // Promise
{<fulfilled>: 'Namaste JavaScript'}
```

//❓ How to extract data from above promise?
One way is using promise .then
dataPromise.then((res) => console.log(res));
// Namaste JavaScript

Another example where `async` function is returning a Promise

```
const p = new Promise((resolve, reject) => {
    resolve("Promise resolved value!!");
});

async function getData() {
    return p;
}

// In above case, since we are already
returning a promise async function would
```

simply return that instead of wrapping with a new Promise.

```
const dataPromise = getData();
console.log(dataPromise); // Promise
{<fulfilled>: 'Promise resolved value!!!'}
dataPromise.then((res) => console.log(res));
// Promise resolved value!!
```

Q: How we can use `await` along with async function?

A: `async` and `await` combo is used to handle promises.

But Question is how we used to handle promises earlier and why we even need `async/await`?

```
const p = new Promise((resolve, reject) => {
  resolve("Promise resolved value!!!");
});
```

```
function getData() {
  p.then((res) => console.log(res));
}
```

```
getData(); // Promise resolved value!!
```

// 💚 Till now we have been using
`Promise.then/.catch` to handle promise.

// Now let's see how `async await` can help us
 and how it is different

// The rule is we have to use keyword `await`
 in front of promise.

```
async function handlePromise() {
  const val = await p;
  console.log(val);
}

handlePromise(); // Promise resolved value!!
```

📌 `await` is a keyword that can only be used inside a `async` function.

```
await function () {}; // Syntax error: await
is only valid under async function.
```

Q: What makes `async - await` special?

A: Let's understand with one example where we will compare async-await way of resolving promise with older `.then/.catch` fashion. For that we will modify our promise `p`.

```
const p = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Promise resolved value!!");
  }, 3000);
});
```

// Let's now compare with some modification:

```
// 📌 Promise.then/.catch way
function getData() {
  // JS engine will not wait for promise to
  be resolved
```

```

    p.then((res) => console.log(res));
    console.log("Hello There!");
}

getData(); // First `Hello There!` would be
printed and then after 3 secs 'Promise
resolved value!!' will be printed.
// Above happened as Javascript wait for
none, so it will register this promise and
take this callback function and register
separately then js will move on and execute
the following console and later once promise
is resolved, following console will be
printed.

```

// ? Problem: Normally one used to get confused that JS will wait for promise to be resolved before executing following lines.

```

// 🔜 async-way:
async function handlePromise() {
    // JS Engine will waiting for promise to
    resolve.

    const val = await p;
    console.log("Hello There!");
    console.log(val);
}

handlePromise(); // This time `Hello There!` won't be printed immediately instead after 3 secs `Hello There!` will be printed followed by 'Promise resolved value!!'
// 💡 So basically code was waiting at

```

`await` line to get the promise resolve before moving on to next line.

```
// Above is the major difference between
Promise.then/.catch vs async-await
```

//🤓 Let's brainstorm more around async-await

```
async function handlePromise() {
```

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

```
    const val = await p;
```

```
    console.log("Hello There!");
```

```
    console.log(val);
```

```
    const val2 = await p;
```

```
    console.log("Hello There! 2");
```

```
    console.log(val2);
```

```
}
```

```
handlePromise();
```

// In above code example, will our program wait for 2 time or will it execute parallelly.

//👉 `Hi` printed instantly -> now code will wait for 3 secs -> After 3 secs both promises will be resolved so ('Hello There!' 'Promise resolved value!!!' 'Hello There! 2' 'Promise resolved value!!!') will get printed immediately.

// Let's create one promise and then resolve two different promise.

```
const p2 = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve("Promise resolved value by
```

```

p2!!");
}, 2000);
});

async function handlePromise() {
  console.log("Hi");
  const val = await p;
  console.log("Hello There!");
  console.log(val);

  const val2 = await p2;
  console.log("Hello There! 2");
  console.log(val2);
}

handlePromise();
// ✏️ `Hi` printed instantly -> now code will
wait for 3 secs -> After 3 secs both promises
will be resolved so ('Hello There!' 'Promise
resolved value!!!' 'Hello There! 2' 'Promise
resolved value by p2!!!') will get printed
immediately. So even though `p2` was resolved
after 2 secs it had to wait for `p` to get
resolved

// Now let's reverse the order execution of
promise and observe response.
async function handlePromise() {
  console.log("Hi");
  const val = await p2;
  console.log("Hello There!");
  console.log(val);
}

```

```

const val2 = await p;
console.log("Hello There! 2");
console.log(val2);

}

handlePromise();
// ✖️ `Hi` printed instantly -> now code will
wait for 2 secs -> After 2 secs ('Hello
There!' 'Promise resolved value by p2!!!')
will get printed and in the subsequent second
i.e. after 3 secs ('Hello There! 2' 'Promise
resolved value!!!') will get printed

```

Q: Question is Is program actually waiting or what is happening behind the scene?

A: As we know, Time, Tide and JS wait for none. And it's true. Over here it appears that JS engine is waiting but JS engine is not waiting over here. It has not occupied the call stack if that would have been the case our page may have got frozen. So JS engine is not waiting. So if it is not waiting then what it is doing behind the scene? Let's understand with below code snippet.

```

const p1 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Promise resolved value by
p1!!!");
  }, 5000);
});

const p2 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Promise resolved value by
p2!!!");
  }, 5000);
});

```

```

    }, 10000);
});

async function handlePromise() {
  console.log("Hi");
  debugger;
  const val = await p1;
  console.log("Hello There!");
  debugger;
  console.log(val);

  const val2 = await p2;
  console.log("Hello There! 2");
  debugger;
  console.log(val2);
}

handlePromise();
// When this function is executed, it will go
line by line as JS is synchronous single
threaded language. Lets observe what is
happening under call-stack. Above you can see
we have set the break-points.

// call stack flow -> handlePromise() is
pushed -> It will log `Hi` to console -> Next
it sees we have await where promise is
suppose to be resolved -> So will it wait for
promise to resolve and block call stack? No -
-> thus handlePromise() execution get
suspended and moved out of call stack -> So
when JS sees await keyword it suspend the
execution of function till promise is

```

resolved -> So `p` will get resolved after 5 secs so handlePromise() will be pushed to call-stack again after 5 secs. -> But this time it will start executing from where it had left. -> Now it will log 'Hello There!' and 'Promise resolved value!!!' -> then it will check whether `p2` is resolved or not -> It will find since `p2` will take 10 secs to resolve so the same above process will repeat -> execution will be suspended until promise is resolved.

//  Thus JS is not waiting, call stack is not getting blocked.

// Moreover in above scenario what if p1 would be taking 10 secs and p2 5 secs -> even though p2 got resolved earlier but JS is synchronous single threaded language so it will first wait for p1 to be resolved and then will immediately execute all.

Real World example of async/await

```
async function handlePromise() {
  // fetch() => Response Object which has body
  // as Readable stream => Response.json() is also
  // a promise which when resolved => value
  const data = await
  fetch("https://api.github.com/users/alok722");
```

```

const res = await data.json();
console.log(res);
}
handlePromise();

```

Error Handling

While we were using normal Promise we were using .catch to handle error, now in `async-await` we would be using `try-catch` block to handle error.

```

async function handlePromise() {
  try {
    const data = await
fetch("https://api.github.com/users/alok722");
    const res = await data.json();
    console.log(res);
  } catch (err) {
    console.log(err);
  }
}

handlePromise();

```

// In above whenever any error will occur the execution will move to catch block. One could try above with bad url which will result in error.

```

// Other way of handling error:
handlePromise().catch((err) =>
  console.log(err)); // this will work as

```

handlePromise will return error promise in case of failure.

Async await vs Promise.then/.catch

What one should use? `async-await` is just a syntactic sugar around promise. Behind the scene `async-await` is just promise. So both are same, it's just `async-await` is new way of writing code. `async-await` solves few of the short-coming of Promise like `Promise Chaining`. `async-await` also increases the readability. So sort of it is always advisable to use `async-await`.

Watch Live On Youtube below:



Episode 24 : Promise APIs (all, allSettled, race, any) + Interview Questions 🔥

4 Promise APIs which are majorly used:

- Promise.all()
- Promise.allSettled()
- Promise.race()
- Promise.any()

 One simply doesn't use async/await without knowing promises!

Promise.all()

A promise is a placeholder for a value that's going to be available sometime later. The promise helps handle asynchronous operations. JavaScript provides a helper function `Promise.all(promisesArrayOrIterable)` to handle multiple promises at once, in parallel, and get the results in a single aggregate array.

Q: In what situation one could use above api?

A: Suppose, you have to make parallel API call and get the result, how one can do? This is where Promise.all can be utilized. It is used to handle multiple promises together.

`Promise.all([p1, p2, p3])` → Lets assume we are making 3 API call to fetch data. Also assume **p1 takes 3 seconds, p2 takes 1 second, p3 takes 2 seconds.**

In first scenario let's assume all 3 promises are successful. So `Promise.all` will take **3secs** and will give promise value of result like `[val1, val2, val3]`. It will wait for all of them to finish then it will collect the results and give array as output.

What if any of the promise gets rejected, for eg: `Promise.all([p1, p2, p3])`. But this time, `p2` get rejected after 1 sec. Thus `Promise.all` will throw same error as `p2` immediately as soon as error happened. It will not wait for other promise to either become success or failure. Moreover, `p1` and `p2` wont get cancelled as they are already triggered so it may result in success or failure depending upon their fate but `Promise.all` wont care. So its a situation of or/null.

 To conclude, the `Promise.all()` waits for all the input promises to resolve and returns a new promise that resolves to an array containing the results of the input promises. If one of the input promises is rejected, the `Promise.all()` method immediately returns a promise that is rejected with an error of the first rejected promise.

Promise.allSettled()

Promise.allSettled() method that accepts a list of Promises and returns a new promise that resolves after all the input promises have settled, either resolved or rejected.

`Promise.allSettled([p1, p2, p3])` → Lets assume we are making 3 API call to fetch data. Also assume **p1** takes **3 seconds**, **p2** takes **1 second**, **p3** takes **2 seconds**.

In first scenario let's assume all 3 promises are successful. So `Promise.allSettled` will take **3secs** and will give promise value of result like `[val1, val2, val3]`. It will wait for all of them to finish then it will collect the results and give array as output.

What if any of the promise gets rejected, for eg: `Promise.all([p1, p2, p3])`. But this time, `p2` get rejected after 1 sec. Thus `Promise.allSettled` will still wait for all promises to get settled. So After 3 secs, it will be `[val1, err, val3]`

 `Promise.all()` → Fail Fast

 `Promise.allSettled()` → Will wait and provide accumulative result

Promise.race()

The Promise.race() static method accepts a list of promises as an iterable object and returns a new promise that fulfills or rejects as soon as there is one promise that fulfills or rejects, with the value or reason from that promise. The name of Promise.race() implies that all the promises race against each other with a single winner, either resolved or rejected.

Promise.race([p1, p2, p3]) → Lets assume we are making 3 API call to fetch data. Also assume **p1 takes 3 seconds, p2 takes 1 second, p3 takes 2 seconds**. So as soon as first promise will resolve or reject, it will give the output.

So in Happy scenario, Promise.race will give (val2) as output after 1sec as p2 got resolved at the earliest. Whereas if it would have been failed Promise.race would have still given output after 1 sec but this time with error.

Promise.any()

The Promise.any() method accepts a list of Promise objects as an iterable object. If one of the promises in the iterable object is fulfilled, the Promise.any() returns a single promise that resolves to a value which is the result of the fulfilled promise.

Promise.any([p1, p2, p3]) → Lets assume we are making 3 API call to fetch data. Also assume **p1 takes 3 seconds, p2 takes 1 second, p3 takes 2 seconds**. So as soon as first promise will be successful, it will give the output.

If in above situation what if p2 got rejected, nothing will happen as Promise.any seek for success, so the moment first success will happen that will become the result.

❓ But what if all promises got failed, so the returned result will be aggregated error i.e. [err1, err2, err3].

Code Examples:

Promise.all()

```
// ⚡ First Scenario

const p1 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("P1 Success");
  }, 3000);
});

const p2 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("P2 Success");
  }, 1000);
});

const p3 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("P3 Success");
  }, 2000);
};

Promise.all([p1, p2, p3]).then((results) => {
  console.log(results); // ['P1 Success', 'P2 Success', 'P3 Success'] -> took 3 secs
});
```

```
// 🔒 Second Scenario

const p1 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("P1 Success");
  }, 3000);
});

const p2 = new Promise((resolve, reject) => {
  setTimeout(() => {
    reject("P2 Fail");
  }, 1000);
});

const p3 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("P3 Success");
  }, 2000);
});

Promise.all([p1, p2, p3])
  .then((results) => console.log(results))
  .catch((err) => console.error(err)); // throws error after 1 sec i.e. 'P2 Fails'
```

Promise.allSettled()

 This is safest among all Promises API.

```
const p1 = new Promise((resolve, reject) => {
  setTimeout(() => {
```

```
        resolve("P1 Success");
    }, 3000);
});

const p2 = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve("P2 Success");
    }, 1000);
});

const p3 = new Promise((resolve, reject) => {
    setTimeout(() => {
        reject("P3 Fail");
    }, 2000);
});

Promise.allSettled([p1, p2, p3])
    .then((results) => console.log(results))
    .catch((err) => console.error(err));

// Over here, it will wait for all promises
// to be either settled or rejected and then
// return,
/*
[
    {status: 'fulfilled', value: 'P1
Success'},
    {status: 'fulfilled', value: 'P2
Success'},
    {status: 'rejected', reason: 'P3 Fail'}
]
*/
```

Promise.race()

```
// 🔜 First Scenario

const p1 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("P1 Success");
  }, 3000);
});

const p2 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("P2 Success");
  }, 1000);
});

const p3 = new Promise((resolve, reject) => {
  setTimeout(() => {
    reject("P3 Fail");
  }, 2000);
});

Promise.race([p1, p2, p3])
  .then((results) => console.log(results))
  .catch((err) => console.error(err));

// It will return as soon as first promise is
// resolved or rejected.
// In above example O/P: "P2 Success"
```

```
// 🔒 Second Scenario

const p1 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("P1 Success");
  }, 3000);
});

const p2 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("P2 Success");
  }, 5000);
});

const p3 = new Promise((resolve, reject) => {
  setTimeout(() => {
    reject("P3 Fail");
  }, 2000);
});

Promise.race([p1, p2, p3])
  .then((results) => console.log(results))
  .catch((err) => console.error(err));

//After 2 secs O/P: "P3 Fail"
```

Notes:

- Once promise is settled, it means → got the result. Moreover, settled is broadly divided into two categories:

1. resolve, success, fulfilled

2. reject, failure, rejected

Promise.any()

```
// ⚡ First Scenario

const p1 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("P1 Success");
  }, 3000);
});

const p2 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("P2 Success");
  }, 5000);
});

const p3 = new Promise((resolve, reject) => {
  setTimeout(() => {
    reject("P3 Fail");
  }, 2000);
});

Promise.any([p1, p2, p3])
  .then((results) => console.log(results))
  .catch((err) => console.error(err));

// It will wait for first settled **success**
// In above, p3 will settle first, but since
```

it is rejected, so it will wait further so at 3rd second it will print "P1 Success"

// 🔜 Second Scenario

```
const p1 = new Promise((resolve, reject) => {
  setTimeout(() => {
    reject("P1 Fail");
  }, 3000);
});

const p2 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("P2 Success");
  }, 5000);
});

const p3 = new Promise((resolve, reject) => {
  setTimeout(() => {
    reject("P3 Fail");
  }, 2000);
});

Promise.any([p1, p2, p3])
  .then((results) => console.log(results))
  .catch((err) => console.error(err));

// After 5 secs: 'P2 Success'
```

// 🔔 Third Scenario

```
const p1 = new Promise((resolve, reject) => {
  setTimeout(() => {
    reject("P1 Fail");
  }, 3000);
});

const p2 = new Promise((resolve, reject) => {
  setTimeout(() => {
    reject("P2 Fail");
  }, 5000);
});

const p3 = new Promise((resolve, reject) => {
  setTimeout(() => {
    reject("P3 Fail");
  }, 2000);
});

Promise.any([p1, p2, p3])
  .then((results) => console.log(results))
  .catch((err) => {
    console.error(err);
    console.error(err.errors); // ['P1 Fail',
    'P2 Fail', 'P3 Fail']
  });

// Since all are rejected, so it will give
// "aggregate error" as output
// AggregateError: All promises were rejected
```

```
// To get AggregateError array you need to  
// write "err.errors"
```

Summary

There are 6 static methods of Promise class:

Promise.all(promises) – waits for all promises to resolve and returns an array of their results. If any of the given promises rejects, it becomes the error of Promise.all, and all other results are ignored.

Promise.allSettled(promises) (recently added method) – waits for all promises to settle and returns their results as an array of objects with: status: "fulfilled" or "rejected" value (if fulfilled) or reason (if rejected).

Promise.race(promises) – waits for the first promise to settle, and its result/error becomes the outcome.

Promise.any(promises) (recently added method) – waits for the first promise to fulfill, and its result becomes the outcome. If all of the given promises are rejected, AggregateError becomes the error of Promise.any.

Promise.resolve(value) – makes a resolved promise with the given value.

Promise.reject(error) – makes a rejected promise with the given error. Of all these, Promise.all is probably the most common in practice.

Watch Live On Youtube below:



Episode 25 : **this** keyword in JavaScript

In JavaScript, the `this` keyword refers to an object, which object depends on how `this` is being invoked (used or called).

this in global space

Anything defined globally is said to be in a global space.

```
console.log(this); // refers to global object
i.e. window in case of browser
//💡 global object differs based on runtime
environment,
```

this inside a function

```
function x() {
    // the below value depends on strict/non-
    strict mode
    console.log(this);
    // in strict mode - undefined
    // in non-strict mode - refers to global
    window object
}
x();
//💡 Notes:
```

// On the first go feels like `this` keyword in global space and inside function behaves same but in reality it's different.

// The moment you make JS run in strict mode by using: "use strict" at the top, `this` keyword inside function returns `undefined`

whereas global space will still refers to global window object

this substitution → According to **this** substitution, if the value of **this** keyword is **null/undefined**, it will be replaced by **globalObject** only in non-strict mode. This is the reason why **this** refers to global window object inside function in non-strict mode.

💡 So to summarize, the value of **this** keyword inside function is **undefined**, but because of **this substitution** in non-strict mode **this** keyword refers to **globalWindowObject** and in strict mode it will still be **undefined**

this keyword value depends on how the **function** is called. For eg:

In strict mode:

```
x(); // undefined
window.x(); // global window object
```

this inside a object's method

```
// `x` key below is a method as per
terminology
const obj = {
  a: 10,
  x: function () {
```

```

        console.log(this); // {a: 10, x: f()}
        console.log(this.a); // 10
    },
};

obj.x(); // value of `this` is referring to
current object i.e. `obj`

```

call , apply & bind methods

For detail around call, apply and bind method. Refer [here](#).

```

const student = {
    name: "Alok",
    printName: function () {
        console.log(this.name);
    },
};
student.printName(); // Alok

const student2 = {
    name: "Kajal",
};
student2.printName(); // throw error

// ? how to re-use printName method from
// `student` object
student.printName.call(student2); // Kajal
// Above `call` method is taking the value of

```

```
`this` keyword
// So, Inside `printName` method value of
`this` is now `student2` object

// So, call, bind and apply is used to set
the value of this keyword.
```

this inside arrow function

Arrow function doesn't have their own `this` value, they take the value from enclosing lexical context.

```
const obj = {
  a: 10,
  x: () => {
    console.log(this); // window object
    // Above the value of `this` won't be obj
    // anymore instead it will be enclosing lexical
    // context i.e. window object in current
    // scenario.
  },
};

obj.x();

const obj2 = {
  a: 10,
  x: function () {
    const y = () => {
      console.log(this);
    }
  }
}
```

```
// Above the value of `this` will be  
obj2 as function y's enclosing lexical  
context is function `x`.  
};  
y();  
},  
};  
obj2.x();
```

this inside DOM

It refers to HTML element.

```
<button onclick="alert(this)">Click  
Me</button>  
<!-- [object HTMLButtonElement] Button  
element -->
```

Watch Live On Youtube below:

