

How JavaScript Works & Execution Context

1. What is Execution Context?

Everything in JavaScript runs inside an **Execution Context**, which acts like a container where code executes. Two main parts:

- **Memory Component** (*Variable Environment*)
Stores variables & function declarations as key-value pairs.
 - **Code Component** (*Thread of Execution*)
Executes code sequentially, one line at a time.
-

2. JavaScript is Single-Threaded and Synchronous

- **Single-threaded** → only one command runs at a time
 - **Synchronous** → executes in order, line-by-line
You cannot move to the next line until the current one finishes.
-

Why this matters

Understanding execution context helps you grasp:

- How variables and functions are stored and managed
 - Why JavaScript is synchronous by default
 - The groundwork for deeper concepts like hoisting, scope, closures, and the event loop
 -
-

How JS Is Executed & Call Stack

1. Program Execution Context

When a JavaScript program starts, a **Global Execution Context (GEC)** is created automatically. It runs in two phases:

- **Memory Creation Phase:** JS allocates memory:
 - Variables (**var**) → **undefined**
 - Functions → full definition
 - **Code Execution Phase:** JS executes code line-by-line.
-

2. Example Walkthrough

```
var n = 2;
```

```
function square(num) {  
  var ans = num * num;  
  return ans;  
}  
  
var square2 = square(n);  
var square4 = square(4);
```

Execution Breakdown:

1. Memory Phase

- `n` → `undefined`
- `square` → function definition
- `square2, square4` → `undefined`

2. Code Phase

- `n = 2`
- Calling `square(2)`:
 - New function execution context (memory component/ variable environment and code Component/thread of execution):
 - `num = 2`
 - `ans = 4`
 - returns `4`, stored in `square2`
- Calling `square(4)`:
 - Similar context:
 - `num = 4`
 - `ans = 16`
 - returns `16`, stored in `square4`

3. Call Stack Mechanics

JS is **single-threaded** and synchronous. The **Call Stack** manages order of execution of execution contexts:

- Start: `Global Execution Context` (bottom)
- `square(n)` is called → new context pushed on top
- Once `square` returns → its context is popped
- Then `square(4)` → another context push/pop
- End: Back to just the GEC
- Some fancy names for call stack
 - Execution context stack
 - program stack
 - control stack
 - runtime stack
 - machine stack

4. Why This Matters

Understanding this gives clarity on:

- How functions create their own execution contexts
 - How local variables (**num**, **ans**) don't pollute global scope
 - How the call stack keeps track of what to run next
 - How recursion and nested calls are handled
-

Key Takeaways

- JS manages execution via **Execution Contexts** and **Call Stack**
 - Two-phase execution ensures hoisting and safe variable access
 - Each function invocation gets its own context
 - Local vs. global scope controlled via context stacking
-

Hoisting in JavaScript

1. What Is Hoisting?

Hoisting is JavaScript's behavior during the **memory creation phase**, where:

- **Variable declarations** (with **var**) are hoisted and initialized as **undefined**.
 - **Function declarations** are hoisted with their full definitions.
- This allows you to use variables and call functions before they appear in your code—though with different outcomes.
-



2. Memory Creation Phase Explained

Before execution, JS engine builds the memory model (Execution Context):

```
Memory (before code runs):
* var x      → undefined
* function foo() { ... } → full function code
```

So calls before declaration often don't error—but be careful which declaration you're using.

3. Example 1 – Function Declarations

```
getName();           //  Logs "Namaste JavaScript"
console.log(x);      //  undefined
var x = 7;
```

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

✓ `getName()` works due to function hoisting ✓ x logs `undefined`, not an error

⚠ 4. Example 2 – Missing Declaration

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

✓ The function exists ✗ `x` isn't declared, so accessing it gives a **ReferenceError**

✗ 5. Example 3 – Function Expressions

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

✓ `getName` is a **variable declaration**, not a function declaration ✓ It's hoisted as `undefined`, so calling it yields a **TypeError**

🔑 6. Hoisting Rules Summary

Declaration Type	Hoisting Behavior
<code>var x</code>	Hoisted → <code>undefined</code>
<code>let / const</code>	Hoisted into TDZ → <code>ReferenceError</code>
<code>function fn() {...}</code>	Fully hoisted → callable anytime
<code>var fn = function()</code>	Hoisted as <code>undefined</code> → <code>TypeError</code> if called

🐉 7. Scope of Hoisting

Hoisting always occurs within its current scope—**global or function-level**:

```
function foo() {  
  console.log(a); // undefined  
  var a = 5;  
}  
foo();  
console.log(a); // ReferenceError
```

✓ Inside `foo`, `a` is hoisted ✗ Outside, `a` doesn't exist → `ReferenceError`

✓ 8. Key Takeaways

- Hoisting moves **declarations** to top during memory phase, not initializations
- `var` → hoisted as `undefined`; `let/const` → TDZ; function declarations → fully hoisted
- **Function expressions** behave like variables—no hoisting of function code
- Understanding hoisting is crucial to avoid unexpected bugs and errors

How Functions Work & Variable Environment

1. Function Invocation & Execution Context

Whenever a function is invoked, JavaScript creates a **new Functional Execution Context (FEC)** — a container similar to the global one but scoped to that function. Each FEC has:

```
Execution Context (Function)  
├─ Memory Component (Variable Environment)  
│   ├── Parameters as variables  
│   └── Local variables (initialized `undefined`)  
├─ Code Component (Thread of Execution)  
│   └── Function body executed line-by-line  
└─ Outer Lexical Environment reference
```

2. Variable Environment in Function

When a function runs:

1. **Memory Phase:**

- Parameters and local `var` variables are hoisted (set to `undefined`).

2. **Execution Phase:**

- Values are assigned, logic is executed, and function may return a value.

Example:

```
var x = 1;
a();
b();
console.log(x);

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

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

Output:

```
10
100
1
```

Execution Flow:

Global Execution Context

Memory Phase:

- x → undefined
- a → [function a]
- b → [function b]

Execution Phase:

- x = 1
- a() invoked → new FEC created

a() Execution Context

Memory:

- x → undefined

Execution:

- x = 10
- console.log(x) → 10

FEC for `a()` is then **popped off** the call stack.

◆ b() Execution Context

Memory:

- x → undefined

Execution:

- x = 100
- console.log(x) → 100

FEC for `b()` is popped off after execution.

◆ Back to Global Context

```
console.log(x); // 1
```

Global `x` is still 1 → prints 1.

🔗 Scope Isolation Insight:

Each function creates its **own local `x`**, which:

- **Shadows** the global `x` inside its own scope
 - Doesn't affect or overwrite `x` outside the function
-

📁 3. Call Stack + Function Contexts

Call Stack:

[b() Execution Context]	← after a()
[a() Execution Context]	← after global
[Global Execution Context]	

Each function call gets pushed onto the call stack and popped after it finishes.

✅ 4. Key Takeaways

- Function calls create **isolated execution contexts** with their own variables.
 - Variables like `x` are **function-scoped**, even if named the same.
 - Local variables never overwrite global ones.
 - Execution is tracked using the **call stack**.
-

📖 Scope Chain, Scope & Lexical Environment

1. Scope vs Lexical Environment

- Lxical means "in Hierarchy".
- **Scope** defines *where* a variable or function is accessible.
- A **Lexical Environment (LE)** is created with each execution context, combining:
 - Local memory (variables/functions)
 - Reference to the parent Lexical Environment.

2. Examples & Scope Chain

✓ Case 1 – Global Access

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

- `a()` finds `b` in the **Global LE** → logs **10**

✓ Case 2 – Nested Function, Global Access

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

- `c()` inside `a()` uses scope chain: `c` → `a` → **Global** → logs **10**

✓ Case 3 – Inner Shadowing

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



```
var b = 10;  
a();
```

- `c()` has its own `b` → logs **100**, not the global `b`

✓ Case 4 – Lexical Isolation

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

- `c()` logs **10**, but global `b` is not defined outside → error

🌀 3. Scope Chain & Execution Contexts

- When a function runs, a **Scope Chain** is created: local LE → parent LE → ... → global LE → `null`
- JS looks up variables *lexically* through this chain.
- If not found, it throws **ReferenceError**.

🔄 4. Call Stack & Lexical Environments

```
Call Stack: [GEC, a(), c()]  
- c() LE → pointer → a() LE  
- a() LE → pointer → Global LE  
- Global LE → pointer → null
```

Each LE stores local data + a link to its parent, forming the **scope chain**

✓ 5. Key Takeaways

- **Lexical scope**: defined by the location of functions in code.
- Inner functions access variables from parent scopes via the scope chain.
- Each function creates its own **LE with pointer to parent**, forming a chain.
- Global LE's parent is `null`, ending the chain.

`let` & `const` and Temporal Dead Zone

1. Are `let` and `const` hoisted?

Yes—but differently than `var`:

- During memory allocation:
 - `let` and `const` are reserved in memory.
 - They remain **uninitialized**—this is the **Temporal Dead Zone (TDZ)**.
-

2. Temporal Dead Zone (TDZ)

- The time between **start of scope** (when the `let` or `const` variable was first hoisted/ allocated some memory) and **initialization line** (initialized with a value) for `let/const`.
- Accessing these variables *before they're initialized* causes a **ReferenceError**.
- Example:

```
console.log(a); // undefined (var)
console.log(b); // ReferenceError (let)

var a = 100;
let b = 200;
```

3. Why TDZ matters

- Prevents use of variables before definition.
 - Encourages cleaner, more predictable code.
 - Reduces bugs caused by accidental early use.
-

4. Memory placement

- `var`: stored in **Global Execution Context**, accessible via `window.varName`.
 - `let/const`: are allocated memory but they are stored in a **separate memory space**, *not* attached to `window`. and you cannot access them before initialization.
-

5. `let` vs `const`

Feature	<code>let</code>	<code>const</code>
Reassignable?	✓ Yes	✗ No → TypeError if reassigned
Redeclarable?	✗ No	✗ No → SyntaxError if redeclared

Feature	let	const
Requires init?	✗ No	✓ Yes → SyntaxError if not initialized

✓ 6. Best Practices

- Prefer **const** by default.
- Use **let** when reassignment is needed.
- Avoid **var**—it's function-scoped and more error-prone.
- Declare variables at the **top of their scope** to shrink TDZ.

🔄 7. Visual Summary

```
[ scope start ]
  ↓
TDZ begins for b, c
  ↓
let b = 10; // TDZ ends, b initialized
const c = 20; // TDZ ends, c initialized
  ↓
[ scope end ]
```

📖 Final Take

let and **const** are hoisted but cannot be accessed before initialization due to TDZ. Use **const** for constants, **let** when needed, and avoid **var** to keep code safe, clean, and predictable.

📖 Block Scope & Shadowing

🔍 1. What is a Block?

- A **block** is a group of statements wrapped in `{...}`. It creates its own **block scope**. A block is also known as compound statement.
- We can group multiple statements in a block where javascript expects a single statement. Example:

```
if(true) {
  var a = 10;
  console.log(a);
}
```

🔑 2. Block Scope vs **var**

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

- `var a` is **function/global scoped**, so it leaks out. get's attached in the global window object.
- `let b` and `const c` are **block-scoped** and cannot be accessed outside. get's attached to a separate memory space.

🔄 3. Shadowing with `var`

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

- Inner `var a` **overwrites** the outer one because both share the same scope (a is attached to the global window object).

✅ 4. Shadowing with `let` and `const`

```
let b = 100; // Separate memory - script scoped
{
  let b = 20; // separate memory - block scoped
  const c = 30;
  console.log(b); // 20
}
console.log(b); // 100
```

- Inner `let b` **shadows** outer `b`.
- Outer `b` remains unaffected: block vs script values.

⊘ 5. Illegal Shadowing

```
let a = 20;
{
```

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

- You **cannot** declare a `var` inside a block if a `let` with the same name exists outside.
- However:
 - You *can* shadow `var` with `let` or `const`.
 - You can also shadow block-scoped variables with other block-scoped variables.

💡 6. Shadowing in Functions

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

- Functions work the same: inner `c` shadows outer `c`.

✅ 7. Summary

- NOTE: Blocks also follow the Lexical Environment Scope Chain Behaviour (Block Chain).
- All the scope rules that apply to a function, also applies same to the arrow functions.
- A **block** creates its own scope for `let` and `const`.
- `var` ignores block scope—declared at function/global level.
- **Shadowing** means inner variables with the same name hide outer ones:
 - `var` shadows `var`
 - `let/const` shadows `let/const` or `var`
 - But `var` cannot shadow `let/const`



Closures in JS

🔍 1. What Is a Closure?

- A **closure** is a function bundled together with its **lexical environment**—i.e., all the variables that were in scope when it was created.
This allows the function to **remember and access** those variables even after its outer function has finished execution.
- A **closure** is a function bundled with its lexical environment, giving it access to its **parent scopes** even after those functions have finished execution.

2. Primary Example:

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

- `x()` sets `a = 7` and returns function `y`.
- Even though `x()` has executed, `y()` still **remembers** `a` → logs 7.
- Now we can call `z()` anytime, and it's like running `y()` — with access to `var a = 7`, even though `x()` has already finished execution.

3. Nested Example:

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

- `y()` can access both `a` and `b`, thanks to its surrounding scope.*
- `y` forms a closure with its parent function `x`, and also with `x`'s parent `z`, capturing variables from both scopes.

4. Key Principle:

A closure enables a function to access variables from its outer (lexical) scope even after the outer function has returned.

5. Advantages

Closures power many useful patterns:

- **Data hiding & encapsulation**
 - **Module pattern**
 - **Function currying**
 - **Memoization**
 - Maintaining state in callbacks (e.g. `setTimeout`)
-

⚠ 6. Disadvantages

- Can cause **memory bloat** if variables are kept unintentionally alive.
 - Potential for **memory leaks** if not managed carefully.
 - Be mindful of **resource consumption** in long-lived closures.
-

Ah, that makes sense now — and yes, your updated code is **100% valid** and a classic closure gotcha. Here's the **clean and corrected version** for your `.md` notes:

🔑 Closure Gotchas:

```
function x() {  
  var a = 7;  
  return function y() {  
    console.log(a);  
  }  
  a = 100;  
}  
var z = x();  
z(); // logs: 100  
  
console.log(z); // logs: f y() { console.log(a); }
```

✓ Explanation:

- `x()` creates `a = 7`, but then immediately reassigns `a = 100`.
 - Function `y()` has the **reference** to `a`, not the value.
 - So when `z()` is called, it logs **100** — the latest value of `a` at the time `y()` runs.
 - This shows closures capture **variable bindings**, not their values at closure creation time.
-

📖 `setTimeout` + Closures Interview Trap

🔍 1. The Basic Closure in `setTimeout`

```
function x() {  
  var i = 1;  
  setTimeout(function () {
```

```

    console.log(i);
  }, 3000);
  console.log("Namaste JavaScript");
}
x();

```

🧐 What happens:

- JS encounters `setTimeout` and **registers the callback** to run after 3 seconds.
- It doesn't wait — `console.log("Namaste JavaScript")` runs **immediately**.
- Time, Tide and Javascript waits for none.
- After 3s, the callback runs and **logs i**, which is **1** (due to closure).
- The callback inside `setTimeout` forms a **closure with the scope of x()**, allowing it to access variables like **i** even after `x()` has completed execution.

✅ Output:

```

Namaste JavaScript
1 ← (after 3 seconds)

```

Closure allows the callback to **remember i**, even after `x()` has finished executing.

⚠️ 2. The Classic Interview Trap – Loop with `var`

```

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

```

✗ Output:

```

6
6
6
6
6

```

🧐 Why does it print 6 five times?

- `var` is **function-scoped**, not block-scoped.
- All 5 callbacks **share the same** `i`.
- By the time the timer fires, the loop has already finished and `i = 6`.

The callbacks don't capture **values**, they capture a **reference to** `i` — and it's 6 by the time they execute.

✓ 3. Solution 1 – Use `let` (Block Scoped)

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

✓ Output:

```
1  
2  
3  
4  
5
```

✓ Why it works:

- `let` is **block-scoped**, so each iteration has its own version of `i`.
- Each `setTimeout` callback closes over a **separate copy** of `i`.

✓ 4. Solution 2 – Use a helper function

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

✓ Output:

```
1
2
3
4
5
```

✓ Why it works:

- Each `i` is passed to a separate invocation of `close()`.
- Inside `close`, a new scope is created, so the `setTimeout` callback captures a fresh `x` every time.
- Avoids the shared reference problem of `var` inside the loop.

The closure now captures `j` (not the shared `i`), which locks the value per iteration.

⚠ 5. Important Concepts at Play

🧠 `setTimeout`:

- Doesn't block — it schedules the callback and moves on.
- Callback goes to the **Web API**, then to the **callback queue**, and runs via the **Event Loop** after the delay.

🧠 Closures:

- A function "remembers" its **lexical environment**, even after the outer function has finished.
- **Closures capture variables by reference**, not by value — unless you isolate them (via `let` or IIFE).

🧠 `var` vs `let`:

<code>var</code>	<code>let</code>
Function-scoped	Block-scoped
Shared across loop	New binding per iteration
Hoisted + initialized as <code>undefined</code>	Hoisted but in TDZ

✓ Summary

- Use `let` in loops with async code to avoid closure traps.
- `var` + closure = shared state = interview trap.
- Closures hold references, not snapshots.
- Mastering this helps in understanding timers, async logic, and safe loop patterns.

📖 CRAZY JS INTERVIEW 🤖 ft. Closures

🔍 1. Famous Closure Questions & Answers

Q1: What is a closure?

A **closure** is a function paired with its **lexical environment**, meaning it can access variables from its outer scope even after that outer function has finished.

```
```js
function outer() {
 var a = 10;
 function inner() {
 console.log(a);
 }
 return inner;
}
outer(); // logs 10
```

---

Q2: Does order matter?

No. Even if you declare the variable after the inner function:

```
function outer() {
 function inner() { console.log(a); }
 var a = 10;
 return inner;
}
outer(); // still logs 10
```

---

Q3: **const** or **let** instead of **var**?

Doesn't matter—closures work the same way:

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

---

Q4: Arguments are captured too?

Yes! Closures include function parameters:

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

---

## Q5: Multiple nested layers?

Closures can reach all the way up the chain:

```
function outest() {
 var c = 20;
 function outer(str) {
 let a = 10;
 function inner() {
 console.log(a, c, str);
 }
 return inner;
 }
 return outer;
}
outest()('hi')(); // logs 10, 20, "hi"
```

---

## Q6: What if global variables shadow inner names?

Inner scope always wins unless it's missing:

```
let a = 100;
outest()('hello')(); // still logs 10 – inner `a` shadows global
```



## 2. Why Closures Matter

- **Data hiding & encapsulation:** variables live only inside your function.

```
// Constructor Function
function Counter() {
 let count = 0;
 this.incrementCounter() {
```

```

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

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

var counter2 = counter(); // created fresh new counter
counter2().incrementCounter; // 1

```

- **Module pattern, currying, memoization:** closures keep state alive.
- **Event handlers, timers:** let functions remember data after outer execution.

### ⚠ 3. Memory & Garbage Collection

- Closures **hold references to outer variables**, which prevents them from being garbage-collected.
- If misused, this can cause **memory leaks** or bloated memory usage. (can also freeze the browser)
- Modern JS engines are optimized to clean up unused closures, but awareness is key.

#### 🗑 What is Garbage Collection in JavaScript?

**Garbage Collection (GC)** is the process by which the JavaScript engine automatically frees up memory that's no longer in use — i.e., memory that is no longer **reachable**.

#### 🧠 How It Works:

- JS uses a mechanism called **Mark-and-Sweep**:
  - The engine starts from the **root** (like global objects).
  - It recursively marks all **reachable values** (variables, objects, functions).
  - Any value **not marked** (unreachable from any live reference) is considered **garbage** and is **deleted** from memory.

#### 🔗 Closures & GC:

- Closures keep variables **alive** by maintaining references.
- If a closure is **still accessible** (e.g., via an event listener or timeout), its captured variables **won't be garbage collected**.
- Once there are **no references** to a closure (and thus no reference to its environment), **GC kicks in**.

✔ Example:

```
function outer() {
 let a = 10;
 return function inner() {
 console.log(a);
 };
}
const fn = outer(); // closure created, `a` is still in memory
// If `fn` is later set to null → closure + `a` becomes unreachable → GC clears it
```

🔒 Why This Matters:

- Preventing **memory leaks** in long-running apps.
- Understanding how closures can **retain memory longer than expected**.
- Writing more **optimized, leak-free** JavaScript.

✔ 4. Quick Summary Table

Concept	Details
Closure	Function + captured lexical environment
Order	Declaration order doesn't matter — closures capture by reference
Scoping (var/let/const)	All work fine; closure depends on lexical placement, not type
Parameters	Also captured inside closures
Nested closures	Able to access from multiple nested scopes
Shadowing	Inner variables take precedence over outer ones
Memory impact	Variables in closure aren't freed until function is out of scope

📖 First-Class Functions 🏠 ft. Anonymous Functions

🔍 1. What Are First-Class Functions?

JavaScript treats functions as **first-class citizens**, meaning:

- Functions can be **assigned** to variables.
- Functions can be **passed** as arguments to other functions.
- Functions can be **returned** from other functions.
- Functions can be **stored** in data structures (arrays, objects).

✔ Function Statement (a.k.a Function Declaration)

```
function greet() {
 console.log("Hello!");
}
greet(); // ☒ Works before this line too (due to hoisting)
```

- Declared using the **function** keyword at the top level
- ☒ Hoisted fully — both name and body
- Can be invoked **before** it's defined

---

## ☒ Function Expression

```
const greet = function () {
 console.log("Hi there!");
};
greet(); // ☒ Cannot call before this line
```

- A function is assigned to a variable
- ☒ Only the variable is hoisted, not the function body
- You **can't call it before the assignment**

---

## Anonymous Function

```
setTimeout(function () {
 console.log("Anonymous!");
}, 1000);
```

- Function **without a name**
- Mostly used in callbacks, one-liners
- Cannot be hoisted — they only exist when the line is executed

---

## Named Function Expression

```
const greet = function sayHi() {
 console.log("Hey!");
};
greet(); // ☒ works
sayHi(); // ☒ ReferenceError: sayHi is not defined
```

- Function expression **with a name**
- Name is **only visible inside** the function itself (used for recursion, debugging)

- Still not hoisted

## Function Declaration vs Expression

Feature	Function Declaration	Function Expression
Name	Required	Optional (can be anonymous)
Hoisting	✔ Fully hoisted	✗ Not fully hoisted
Call before define?	✔ Yes	✗ No
Syntax	<code>function fn() {}</code>	<code>const fn = function() {}</code>
Use Case	Reusable logic	Callbacks, conditional logic

## 2. Example – Assigning to Variables

```
function greet() {
 console.log("Hey there!");
}
const hello = greet;
hello(); // same as calling greet()
```

- `hello` now references `greet` — both point to the same function.

## 3. Passing as Arguments

```
function sayHi(fn) {
 fn();
}
sayHi(greet); // prints "Hey there!"
```

- `greet` gets passed into `sayHi`, and is called inside it.

## 4. Returning Functions

```
function outer() {
 return function inner() {
 console.log("I'm an inner function");
 };
}
const fn = outer();
fn(); // runs inner()
```



- **outer** returns a function — creating closures if it references outer variables.

---

## ✳ 5. Anonymous Functions & Callbacks

```
setTimeout(function() {
 console.log("Delayed Hello");
}, 1000);
```

- The function passed has **no name** (anonymous).
- Common pattern in callbacks — especially in async code and event listeners.

---

## ✳ 6. Array Example with Anonymous Functions

```
const arr = [1, 2, 3];
const doubled = arr.map(function(num) {
 return num * 2;
});
console.log(doubled); // [2, 4, 6]
```

- **map** receives an **anonymous function** to transform each element.

---

## 🧠 7. Why First-Class Functions Matter

- Enable **functional programming** (map, filter, reduce).
- Allow **higher-order functions**, leading to flexible abstractions.
- Facilitate **callbacks**, **event-driven code**, **promises**, and async patterns.

---

## ✓ 8. Quick Summary

Capability	Example
Assign to variable	<code>const fn = greet; fn();</code>
Pass as argument	<code>sayHi(greet);</code>
Return from function	<code>const fn = outer(); fn();</code>
Use as anonymous	<code>arr.map(num =&gt; num * 2);</code>

---

## 💡 9. Final Notes

JavaScript's power stems from treating functions like any other value. Mastering **first-class functions** is key to writing concise, expressive, and high-level code — and it's crucial for frameworks, async flows, and clean

architecture.

---

## Callback Functions & Event Listeners

---

### 1. What Is a Callback Function?

- A **callback** is a function passed as an **argument** to another function, which is **called back later**.
- Transforms synchronous code into asynchronous behavior.

---

### 2. Example: setTimeout

```
console.log("Start");
setTimeout(function callback() {
 console.log("Delayed Hello");
}, 2000);
console.log("End");
```

**Flow:**

```
Start
End
...after 2 seconds...
Delayed Hello
```

- `setTimeout` schedules the callback in the future.
- Meanwhile, the JS engine continues executing remaining code.

---

### 3. Event Listener Callback

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

- The provided function runs **only** when the event (like "click") fires.
- Acts like a callback: JS registers it and invokes it later.

---

### 4. Private Counter Using Callbacks

You can use closures to keep event-state hidden:

```
const btn = document.getElementById("btn");
(function () {
 let count = 0;
 btn.addEventListener("click", function () {
 count++;
 console.log("Clicked", count);
 });
})();
```

- `count` lives privately in a closure.
- Only the event callback can access it.
- You prevent accidental external mutations.

## ⚠ 5. Main Thread Blocking – Why Callbacks Matter

JS runs on a **single thread**. If you run a heavy task:

```
for (let i = 0; i < 1e9; i++) { /* heavy operation */ }
console.log("Done");
```

- The UI will **freeze** until that loop finishes.
- Callbacks (like for timers or events) *wait* in the queue.
- Once the loop ends, callbacks execute—keeping the UI responsive.

## 🚀 6. Visual Flow of Events & Callbacks

```
Main Thread
 ↓
Run JS:
1. Register setTimeout callback
2. Register event listener callback
3. Continue execution
Event Loop → Callback Queue:
...when timer or click happens...
 ↓
JS Engine picks callback from queue
and executes it (on the same thread)
```

## 🧠 7. Key Takeaways

- Callbacks let JS handle **asynchrony** with a single thread.
- They make code **event-driven** and responsive.
- Use closures in callbacks for **data hiding and encapsulation**.

- Avoid blocking the thread—callback patterns keep UI and logic decoupled and fluid.

---

## Garbage Collection (GC) & `removeEventListener()`

---

### What's Garbage Collection?

JavaScript automatically **cleans up memory** by removing values that are **no longer reachable** (i.e., nothing references them anymore).

If a variable/object/function is not referenced **anywhere**, it becomes a candidate for GC.

### Now... What About `addEventListener()`?

When you do this:

```
element.addEventListener('click', function handleClick() {
 console.log('clicked');
});
```

You're attaching a **function reference** to that DOM element. **Result?** That function stays in memory as long as the element does — even if:


- You don't click it
- You don't use it again in code

---

### Problem: Memory Leak Danger

If you:

- Remove that DOM element (`element.remove()` or hide it)
- But you **don't** call `removeEventListener(...)` Then that element **can't be garbage collected** (because it still holds a reference to the callback function).

 You just created a **memory leak** — the browser holds on to that DOM + function pair.

---

### Solution: Always Clean Up Listeners

```
function handleClick() {
 console.log("clicked");
}
element.addEventListener("click", handleClick);
// Later...
element.removeEventListener("click", handleClick);
```

Once `removeEventListener()` is called:

- The reference between the element and the function is broken.
  - If there are no other references to the function, it's GC'd.
  - The element is now safe to remove and clean up.
- 

## 🔗 In Closures?

If you defined the listener inside a closure:

```
(function () {
 let count = 0;
 function clickHandler() {
 count++;
 }
 btn.addEventListener("click", clickHandler);

 // Later:
 btn.removeEventListener("click", clickHandler);
})();
```

Removing the listener also **releases the closure** (and the `count` variable) for garbage collection.

---

## Bonus

### 🔗 When to call `removeEventListener()` ?

We mean **at any point in your code after the listener was added**, if:

- You **no longer need the listener**
  - You are about to **hide** (`display: none`) or **remove** (`element.remove()`) the element 📁 **That's the moment** to call `removeEventListener()`.
- 

💬 Q: If I just do `display: none` to the button, should I also call `removeEventListener()`?

✅ **Yes, if** the element won't be used again for that event.

- Hiding the element doesn't remove it from memory.
  - The event listener **is still attached** in the background.
  - So the **function is still in memory**, even if the user can't trigger it. 🔗 Especially important if you're:
  - Hiding lots of elements dynamically (like in modals, menus, SPA routing)
  - Using closures in those listeners
- 

💬 Q: If I do `element.remove()`, should I also call `removeEventListener()`?

✅ **Yes, and here's why:**

- Technically, modern browsers **try** to clean up attached listeners when you remove an element from the DOM.
- BUT:
  - If the function (listener) **closes over external variables** (closure)
  - Or you're supporting older browsers
  - Or you're attaching many dynamic elements

Then relying on the browser is risky — **call `removeEventListener()` manually** for safety. ☒ It avoids:

- Memory leaks
- Dangling logic
- Unintentional bugs

 Q: Should I call `removeEventListener()` **before** `element.remove()` or `display: none`?

 **Best Practice: Always call `removeEventListener()` before removing or hiding the element.**


```
btn.removeEventListener("click", handleClick);
btn.remove(); // or btn.style.display = "none";
```

Why?

- Makes it **explicit** and predictable
- Ensures all references are broken before the DOM node is gone
- Keeps your code maintainable and bug-free

### ☒ Final Checklist (Keep This Mental Flow):

Action	Should I remove listener first?	Why?
<code>element.style.display = "none"</code>	<input checked="" type="checkbox"/> Yes	Element still lives in memory
<code>element.remove()</code>	<input checked="" type="checkbox"/> Yes	Prevent closure/function leaks
Replacing element content	<input checked="" type="checkbox"/> Yes	Avoid zombie callbacks
SPA route change/unmount	<input checked="" type="checkbox"/> Yes	Prevent event stacking & memory bloat

 Event Listener Lifecycle – ( when to add `addEventListener` once again? )

Action	Listener Retained?	Notes
<code>display: none → block</code>	☑ Yes	Still in DOM, listener intact
<code>removeEventListener()</code>	✗ No	Must re-add manually
<code>element.remove()</code>	✗ No	DOM node & listener destroyed
<code>innerHTML = "" / .replaceWith()</code>	✗ No	Element replaced, listener gone
Detached element (but referenced)	☑ Yes	Listener stays; GC won't run

### 💡 Best Practice:

Always `removeEventListener()` before:

- `remove()`
- `replaceWith()`
- Large DOM UI changes (modals, SPAs)

## 🧠 JS vs JS Engine vs Runtime vs Browser vs Web APIs

### 📦 1. JavaScript

#### ☑ The language.

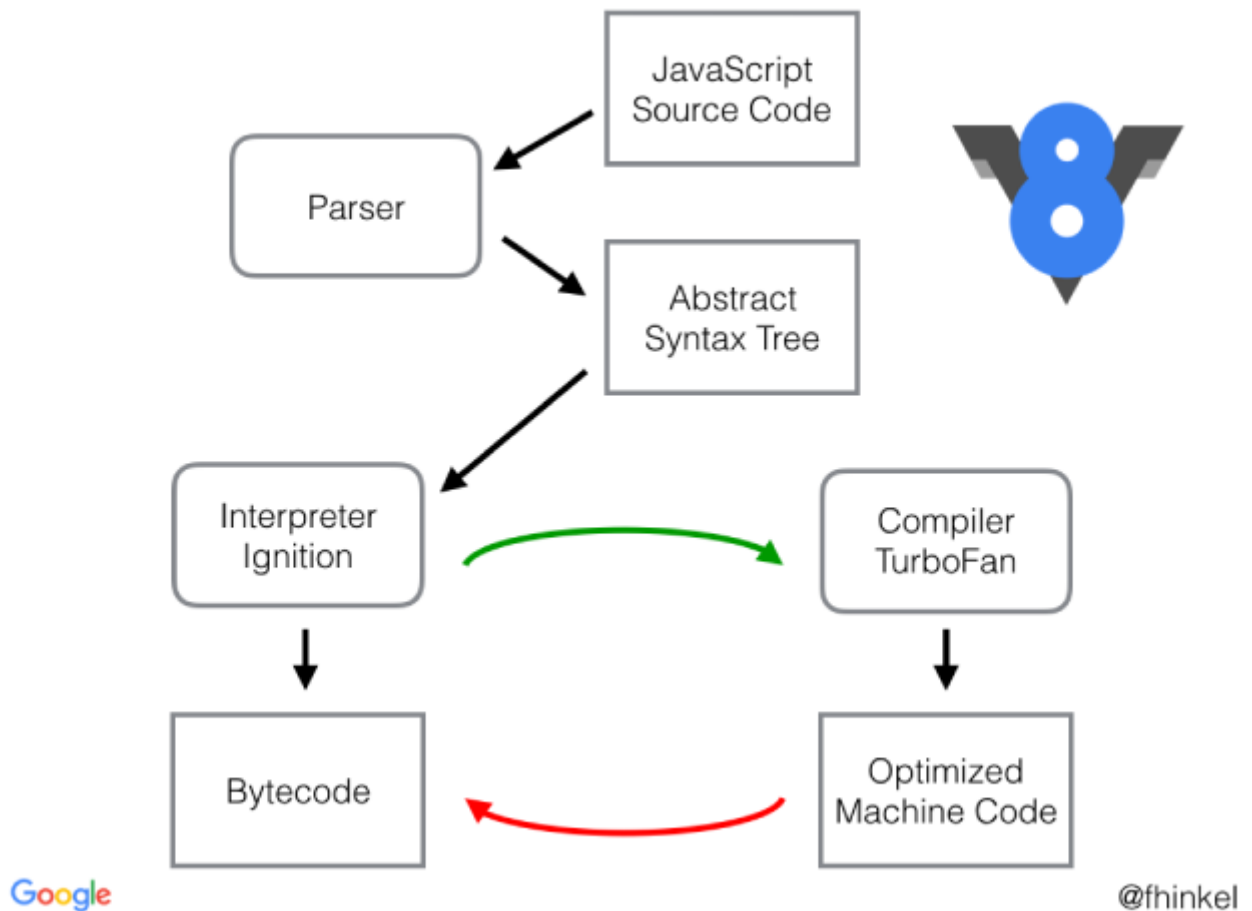
- Defines syntax, types, control structures, functions, etc.
- Does **not** include `setTimeout`, DOM, fetch, etc.
- Specified in **ECMAScript** standard (like a blueprint).

### 📦 2. JavaScript Engine

#### ☑ Runs JavaScript code.

- Parses, compiles, and executes JS.
- Examples:
  - Chrome → **V8**
  - Firefox → **SpiderMonkey**
  - Safari → **JavaScriptCore BUT:** It only knows:
- How to execute JS code
- How to manage memory, scopes, closures
- How to process Promises and microtasks ✗ **No DOM, no setTimeout, no fetch** and contains:
- |— Call Stack ☑
- |— Memory Heap ☑
- |— Execution Thread ☑

- V8 JS ENGINE:



### ◆ 3. JavaScript Runtime

✓ **Engine + Everything Else You Need to Actually Run JS Code** | Runtime = | JavaScript Engine + Web APIs + Event Loop | | ----- | ----- | | In browsers | V8 (or similar) + DOM + Timers + Fetch + Event Loop | | In Node.js | V8 + fs + process + timers + HTTP + Event Loop | **Runtime makes JavaScript feel powerful & async — but these extras aren't in the language itself.**

### ◆ 4. Web APIs (in browsers)

✓ Provided by **the browser**, NOT JavaScript. Includes:

- `setTimeout`, `setInterval`
- DOM APIs
- `fetch()`
- `localStorage`, `cookies`
- `addEventListener`
- `location`, `console.log` etc. When you do:

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



That `setTimeout` is a **browser feature**, not a JS one.

◆ 5. Are `Promise`, `then`, `async/await` Part of JS?

✔ YES. These **are** part of the **JavaScript language spec**.

- `Promise` is part of ECMAScript since ES6
- `async/await` is just syntactic sugar over Promises
- `Promise.then(...)` uses the **Microtask Queue**, which **is handled by the engine** So: | Feature | Belongs To | Queue Used | | ----- | ----- | ----- | | `Promise` | JS Engine (ES spec) | Microtask queue | | `async/await` | JS Engine | Microtask queue | | `setTimeout` | Web API (Browser) | Macrotask queue | | `fetch` | Web API (Browser) | Macrotask (callback) + Microtask (promise) |

◆ 6. Is JS Engine = JS Runtime?

✗ No.

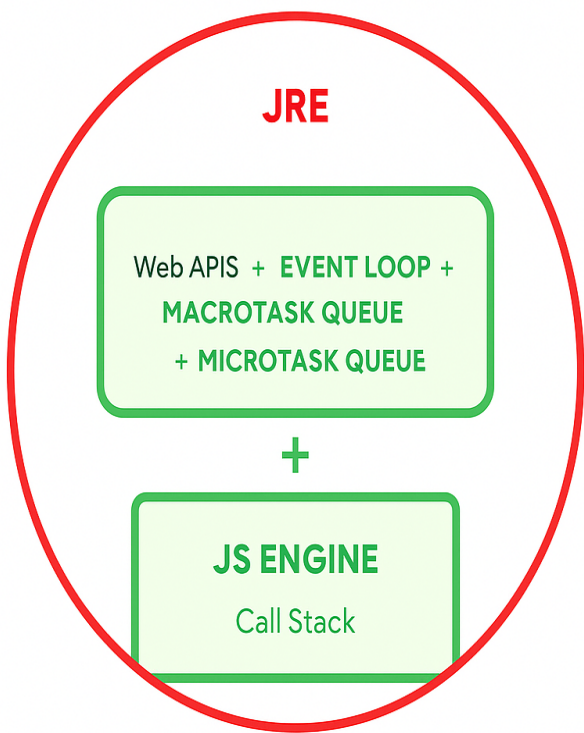
Term	Includes
JS Engine	Just the JS interpreter/compiler (e.g. V8)
JS Runtime	Engine + APIs + Event Loop + Queues

You can't run a full JS app with only the engine.

✔ TL;DR — Clear Answers to Your Questions:

Question	Answer
Is JavaScript synchronous?	✔ Yes. It's a sync language with async features via runtime
Is JS Engine = Runtime?	✗ No. Engine runs JS; Runtime = Engine + APIs + Event Loop
Is <code>setTimeout</code> part of JS?	✗ No. It's a <b>Web API</b> (browser/Node provides it)
Is <code>Promise/then</code> part of JS?	✔ Yes. They're in the language itself
Are <code>async/await</code> features of JS or browser?	✔ JS. They compile to Promises
Is async behavior handled by the JS Engine?	☑ Partially: Promises are; timers, <code>fetch</code> , etc., are handled by runtime
Are Web APIs and browser the same thing?	☑ Kind of. Web APIs are <b>features exposed by the browser</b>
Are Promises async because of browser features?	✗ No. They are async <b>within JS engine</b> using the microtask queue

# BROWSER



## Full Execution Flow: Async Code in JavaScript

Let's say your JS program contains:

- `setTimeout`
- `fetch`
- `setInterval`
- `eventListeners`
- `Promise`
- `async/await`
- Callbacks (sync + async)



### First: The Components Involved

Layer	Responsible For
JS Engine	Executes sync code, handles Promises, async/await, microtask queue
JS Runtime	Provides APIs: <code>setTimeout</code> , <code>fetch</code> , DOM, <code>setInterval</code> , events
Call Stack	Executes functions (sync or async callbacks)
Web APIs	Handles async API execution in background
Microtask Queue	<code>Promise.then()</code> , <code>catch</code> , <code>async/await</code>

Layer	Responsible For
Macrotask Queue	<code>setTimeout</code> , <code>setInterval</code> , DOM events
Event Loop	Coordinates between stack + queues

## 🧩 0. Setup

When your script is loaded:

- **Global Execution Context (GEC)** is created and pushed to the **Call Stack**
- Memory phase: variables/functions are hoisted
- Execution phase: code runs top-down

## ⚙️ 1. Sync Code Execution (JS Engine)

- JS engine runs all **synchronous code line by line**
- Functions like `console.log`, math, DOM access run **immediately** via the **Call Stack**
- If a **Promise** is encountered, its executor function runs **immediately**, but `.then()` gets scheduled
- `async/await` is converted to Promises under the hood

## ⌚ 2. Encountering Async APIs (Handled by Runtime)

Code Line	What Happens
<code>setTimeout(cb, 1000)</code>	Registered in <b>Web API</b> → timer starts
<code>setInterval(cb, 2000)</code>	Registered in <b>Web API</b> → runs on interval
<code>fetch(...)</code>	Sent to Web API → HTTP request sent in background
<code>addEventListener</code>	Listener is registered in Web API → waits for user event

These functions are **NOT JS features** — they're delegated to **browser APIs (runtime)**.

## 🔄 3. As Async APIs Complete

- When timer finishes, HTTP response comes, or event occurs: → the **callback** is placed into the **Macrotask Queue** (aka Callback Queue) ✅ But it will **NOT execute immediately**.

## 📦 4. Promises & Microtasks (JS Engine's Domain)

- When a **Promise** is resolved or rejected: → Its `.then()` or `.catch()` is placed into the **Microtask Queue**
- Same for `await` — it pauses execution, schedules the next step in **Microtask Queue** ✅ Microtasks are JS-engine-native — they don't involve Web APIs or timers.

## 5. The Event Loop's Role

The **Event Loop** is the orchestrator. It does this infinitely:

Event Loop Cycle:

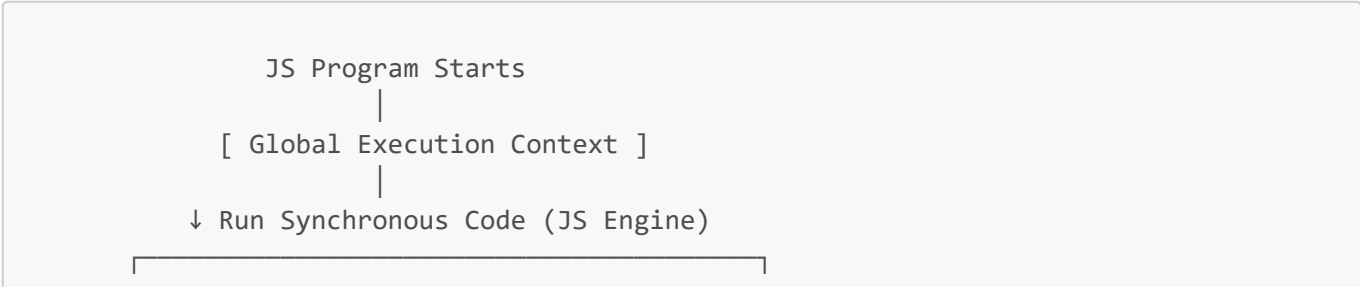
```
while (true) {
 if (Call Stack is empty) {
 Run all Microtasks in order (drain Microtask Queue)
 Run one Macrotask (from Callback Queue)
 }
}
```

✓ This ensures **microtasks always have higher priority** than macrotasks.

## 6. Summary of What Touches What

Code Type	Handled By	Goes To	Runs When
Sync Code	JS Engine	Call Stack	Immediately
<code>Promise.then()</code>	JS Engine	Microtask Queue	After current stack, before macrotask
<code>async/await</code>	JS Engine	Microtask Queue (await resumption)	Same as Promise
<code>setTimeout</code>	JS Runtime (Web API)	Macrotask Queue	After delay, when stack is empty
<code>fetch().then(...)</code>	Web API (fetch) + JS Engine (then)	Microtask Queue for <code>.then()</code>	After response arrives
<code>setInterval</code>	JS Runtime	Macrotask Queue (repeated)	Every interval after stack clears
<code>eventListener</code>	Web API	Macrotask Queue (on event)	When event occurs
<code>callback</code> (sync)	JS Engine	Call Stack	Immediately if sync

## Visual Flow



```
| Async APIs Found → Offloaded to Web APIs |
```

```
↓ JS Engine handles Promises → Microtask Queue
```

```
[Event Loop Starts Watching]
```

```
📌 Loop:
```

```
→ Is Call Stack empty?
```

```
→ Yes? Drain Microtask Queue
```

```
→ Then: Pull one Macrotask
```

```
🕒 Timer/Fetch/Event/Interval Callbacks run
```

```
New ECs pushed to Call Stack
```

```
↓
Functions execute
```

## 🎯 End Result

- JS engine runs sync code + internal async logic
- JS runtime (browser or Node) runs timers, events, HTTP
- **Event loop bridges the gap**
- Microtask queue is **priority 1**, macrotask queue is **priority 2**
- Promises and async/await = handled internally
- Web APIs = handled externally

## 🏠 TRUST ISSUES with setTimeout()

### 🧐 Understanding setTimeout()

```
console.log("Start");
setTimeout(() => {
 console.log("Inside Timeout");
}, 0);
console.log("End");
```

### 📄 Output:

```
Start
End
Inside Timeout
```

## 🔗 Why not "Inside Timeout" right after "Start"?

Even with `0ms`, `setTimeout()` doesn't run **immediately** — it gets pushed to the **macrotask queue** (a.k.a. callback queue), waiting for the call stack to clear.

---

## 🔄 Behind the Scenes

1. Global Execution Context (GEC) is created.
2. Code starts executing line by line (Call Stack).
3. `setTimeout()` is handed over to Web API (Browser/Runtime).
4. After `delay` ms, callback goes into **macrotask queue**.
5. **Event Loop** constantly checks:
  - Is Call Stack empty? ☒
  - Then it pulls callback from Macrotask Queue → Stack

---

## 🔑 Important Facts about `setTimeout()`

Fact	Explanation
<code>setTimeout()</code> is <b>async</b>	Runs via browser's Web API
Callback is queued	In <b>macrotask queue</b> , not immediately
Delay $\neq$ exact timing	Minimum wait time, <b>not a guarantee</b>
Blocked by long sync code	JS is single-threaded. Sync code delays the async.

---

## 🔧 Example: Heavy Code Blocks Timeout

```
setTimeout(() => {
 console.log("Hello");
}, 0);
for (let i = 0; i < 1e9; i++) {} // Heavy sync code
console.log("Done");
```

Output:

```
Done
Hello
```

☞ Even though `setTimeout` was 0ms, it waited until the **blocking loop finished**. Because **JS is single-threaded**.

---

## 🕒 `setTimeout()` ≠ Precise Timing

- 0ms delay means: "Run it **as soon as possible**, but **after current execution** finishes."
  - It never skips the **event loop** and **call stack rules**.
- 

## 🧠 How Delay Affects Things?

```
console.log("Start");
setTimeout(() => console.log("Timeout 1"), 100);
setTimeout(() => console.log("Timeout 2"), 0);
console.log("End");
```

Output:

```
Start
End
Timeout 2
Timeout 1
```

Even though `Timeout 1` has a higher delay, `Timeout 2` is still **not instant**, but gets queued first.

---

## 📊 `setTimeout` vs Promises

```
console.log("Start");
setTimeout(() => console.log("setTimeout"), 0);
Promise.resolve().then(() => console.log("Promise"));
console.log("End");
```

Output:

```
Start
End
Promise
setTimeout
```

🧠 **Promise** goes to **microtask queue** → has higher priority. 🕒 `setTimeout` → **macrotask queue** → runs later.

## ⚠ Gotcha: Delay Stacking

```
setTimeout(() => {
 console.log("First");
 setTimeout(() => {
 console.log("Second");
 }, 0);
, 0);
```

output:  
First  
Second

Even though both are 0ms, **nested setTimeout** causes a **tick delay** between executions.

## 🔑 Bonus

Even **1000s of 0ms setTimeout()** will queue up and execute **one per event loop tick**, never truly "parallel".

```
for (let i = 0; i < 1000; i++) {
 setTimeout(() => console.log(i), 0);
}
```

output: 0 to 1000 printed

⊖ JS is not multithreaded (unless you explicitly use Web Workers). All callbacks wait their turn.

## 🔑 Key Takeaways

- `setTimeout` is NOT a guaranteed timer — it's a **scheduling mechanism**.
- Callback execution always respects the **event loop model**.
- Know your **task queues**: macrotask vs microtask.
- Great for **non-blocking async logic**, but not for exact timing (e.g., animations → use `requestAnimationFrame`).
- Avoid relying on `setTimeout()` for accurate delays.

## Higher-Order Functions

### 🔄 What is a Function in JavaScript?

JS treats functions as **first-class citizens**:



- Functions can be:
  - Assigned to variables ☒
  - Passed as arguments ☒
  - Returned from other functions ☒ This forms the **core of functional programming** in JS.

## ▲ Higher-Order Functions (HOF)

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

1. Takes another function as an **argument**, or
2. Returns a function from within.

```
function hof(callback) {
 console.log("Inside HOF");
 callback();
}
hof(function () {
 console.log("I'm a callback");
});
```

### ☒ Real-World HOF Examples:

Built-in Method	Description
map()	Transforms elements
filter()	Filters based on condition
reduce()	Reduces to a single value
forEach()	Loops through each element
setTimeout()	Takes a function to run after delay
addEventListener()	Passes a callback on event trigger

## 💡 Why Are HOFs Powerful?

- **Abstraction:** Encapsulate behavior and logic.
- **Reusability:** Generic functions handle multiple use-cases.
- **Clean Code:** Reduce loops, nested logic.

## 🔧 Example: Custom map() Implementation

```
function customMap(arr, logic) {
 const result = [];
 for (let i = 0; i < arr.length; i++) {
```

```
 result.push(logic(arr[i]));
 }
 return result;
}
const doubled = customMap([1, 2, 3], (x) => x * 2);
console.log(doubled); // [2, 4, 6]
```

## Functions Returning Functions

```
function greet(greeting) {
 return function (name) {
 console.log(`${greeting}, ${name}!`);
 };
}
const sayHi = greet("Hi");
sayHi("Sid"); // Hi, Sid!
```

This is a **closure** + **HOF** combo — powerful tool in advanced JS.

## map () implementation

```
Array.prototype.calculate = function(logic) {
 const output = [];
 for(let i= 0; i < this.length; i++) {
 output.push(logic(this[i]));
 }
 return putput;
}
const radius = [3, 2, 4];
console.log(radius.calculate((r) => 2 * MATH.PI * r));
```

## map(), filter(), and reduce() in JavaScript

### 1. What are these?

JavaScript provides **functional array methods** that accept **callbacks** and return **new arrays** or values without mutating the original array.

Method	Purpose	Returns
map()	Transforms each element	New array
filter()	Filters elements based on a condition	New array

Method	Purpose	Returns
<code>reduce()</code>	Reduces array to a single value	Any data type

## 2. `map()` – Transform Each Element

 Use:

Create a **new array** by applying a function to **every element**.

```
const nums = [1, 2, 3, 4];
const doubled = nums.map(num => num * 2);
console.log(doubled); // [2, 4, 6, 8]
```

- Non-mutating.
- Returns **same length** array.
- Ideal for **data transformation**.

## 3. `filter()` – Select Elements

 Use:

Create a **new array** by keeping only elements that **pass the condition**.

```
const nums = [1, 2, 3, 4];
const evens = nums.filter(num => num % 2 === 0);
console.log(evens); // [2, 4]
```

- Non-mutating.
- Returns **fewer or same number** of elements.
- Ideal for **cleaning or narrowing data**.

## 4. `reduce()` – Boil Down to a Single Value

 Use:

Use a function to **accumulate/combine** all values into one.

```
const nums = [1, 2, 3, 4];
const sum = nums.reduce((acc, curr) => acc + curr, 0);
console.log(sum); // 10
```

 Syntax:

```
arr.reduce((accumulator, currentValue) => {
 return updatedAccumulator;
}, initialValue);
```

- Powerful for **totals, averages, objects, arrays, even nested reduction**.
- You can build `map()` and `filter()` using `reduce()`.

---

## Real-World Examples

### ◆ Map Names to Uppercase

```
const names = ["sid", "vex", "vish"];
const upper = names.map(name => name.toUpperCase());
// ["SID", "VEX", "VISH"]
```

### ◆ Filter Out Falsy Values

```
const arr = [0, null, "hello", undefined, 42];
const truthy = arr.filter(Boolean);
// ["hello", 42]
```

### ◆ Reduce to Count Occurrences

```
const chars = ['a', 'b', 'a'];
const count = chars.reduce((acc, char) => {
 acc[char] = (acc[char] || 0) + 1;
 return acc;
}, {});
// { a: 2, b: 1 }
```

---

## Why These Matter

Concept	Value
Declarative	Write <b>what</b> to do, not <b>how</b>
Immutability	Safer & easier to reason about
Chainable	Combine multiple methods cleanly
Interview Gold	Common Qs in React, Node, data & logic rounds

---

## Bonus Tip

You can chain them for more power:

```
const result = [1, 2, 3, 4, 5]
 .filter(x => x % 2 === 0) // [2, 4]
 .map(x => x * 10) // [20, 40]
 .reduce((a, b) => a + b); // 60
```

---

## Shortest JS Program, `window`, & `this`

---

### 1. Shortest JS Program

- An **empty .js file** → no code → still executes.
- JS engine creates:
  - **Global Execution Context (GEC)**
  - **Global object** (`window` in browsers) - created inside global scope.
  - `this` in global scope points to `window`
  - global variables and functions are attached to window object.

---

### 2. `window` – The Global Object

- Represents the global scope in browsers.
- Automatically created when code runs.
- All **global variables/functions** attach to `window`.

```
var a = 5;
function c() { return 10; }

console.log(window.a); // 5
console.log(window.c()); // 10
```

- Variables/functions inside functions are **not** on `window`.

---

### 3. `this` in Global Scope

- Outside any function, `this === window` in browsers.

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

- `this` and `window` are interchangeable globally.

---

## 4. Key Takeaways

- **Empty script** still triggers:
  - Global Execution Context
  - Creation of `window` and `this` linking to it
- Global variables/functions attach to `window`.
- `this` in global scope references `window`.

---

## Summary Table

Concept	Behavior in Browser Global Scope
Global EC	Always created, even if file is empty
<code>window</code>	Global object that holds globals
<code>this</code>	Same as <code>window</code> outside any function

---

## `undefined` vs “not defined” in JS

---

### 1. Memory Allocation in JavaScript

- Before running any code, JS performs a **memory creation phase**:
  - Variables (with `var`, `let`, `const`) are allocated memory.
  - Uninitialized variables are set to `undefined`.
  - Functions get their full definitions in memory.

### 2. `undefined` vs. “not defined”

- **`undefined`**: Memory exists but no value is assigned.
- **`ReferenceError: x is not defined`**: No memory reserved — variable was never declared.

---

### 3. Code Examples

```
var a;
console.log(a); // undefined
console.log(b); // ReferenceError: b is not defined
```

- `a` → exists, but no value → **`undefined`**
- `b` → never declared → **`not defined`**

## ⚠️ 4. `undefined` ≠ empty or `null`

- `undefined` is its own type and reserved keyword.
- It's a placeholder until you assign a real value.
- Best practice: **don't manually assign `undefined`** — it confuses intent.

## ⚠️ Why Not to Manually Assign `undefined`

- JS **automatically assigns `undefined`** to uninitialized variables.
- Manually doing `var a = undefined` creates confusion:
  - Was it set by the dev or left uninitialized?
- ☒ Use `null` if you want to **intentionally indicate "no value"**.
- you can also consider using empty string or boolean

---

## 📋 5. Behavior Summary

Situation	Result
Declared, no value	<code>undefined</code>
Never declared	ReferenceError: not defined
Manually set to <code>undefined</code>	Allowed, but discouraged

---

## ✅ 6. Why It Matters

- Understanding the distinction helps in debugging:
  - Logs `undefined` → variable exists but isn't set.
  - Throws ReferenceError → you likely made a typo or forgot to declare it.
- Avoids pitfalls with conditional checks and default values.

---

## 🔄 7. Visual Flowchart

```
[var a;]
 ↓
[a: undefined] -> console.log(a) // prints undefined

[console.log(b)]
 ↓
ReferenceError: b is not defined

🧠 Callback Hell

🤔 What is a Callback?
A callback is a function passed as an argument to another function, which
is then invoked later, usually after an async operation.
```

```
``js
function loadData(callback) {
 setTimeout(() => {
 console.log("Data loaded");
 callback();
 }, 1000);
}
loadData(() => console.log("Callback triggered"));
```

---


## What is Callback Hell?

When callbacks are **nested inside callbacks**, forming a **pyramid** or "**right-angled Christmas tree**" structure, it becomes hard to:


- **Read**
- **Maintain**
- **Handle errors**
- **Scale**

### Example of Callback Hell:

```
setTimeout(() => {
 console.log("1");
 setTimeout(() => {
 console.log("2");
 setTimeout(() => {
 console.log("3");
 }, 1000);
 }, 1000);
}, 1000);
```

 Output:


```
1
2
3
```

 Problem: The deeper it goes, the harder it is to **track logic**, **handle errors**, or **refactor**.

---

## Why This Happens?

JS is **single-threaded** — it uses **async constructs like callbacks** to handle time-based and I/O operations **without blocking** the main thread. To sequence operations, we **nest callbacks**. But...

Nesting = coupling = code chaos 



## Callback Hell Symptoms


Symptom	Pain Point
Deep nesting	Visually hard to parse
Inverted control	You don't control flow, callback does
Error handling mess	<code>try/catch</code> doesn't work in async calls
Repetition	Same logic scattered everywhere

## Solutions to Callback Hell

Technique	Description
Named functions	Break down logic into smaller named callbacks
Promises	Flatten async logic using <code>.then()</code> chains
<code>async/await</code>	Modern syntax — looks like sync, runs async
Error-first pattern	Pass <code>err</code> , <code>result</code> in callbacks for clarity

## Rewriting with Named Functions:

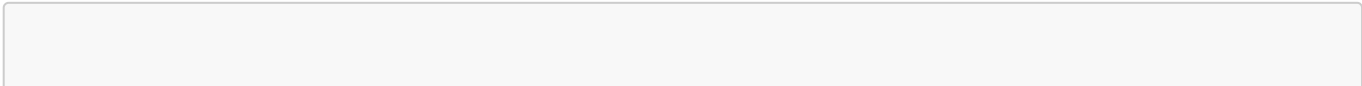
```
function step3() {
 console.log("3");
}
function step2() {
 console.log("2");
 setTimeout(step3, 1000);
}
function step1() {
 console.log("1");
 setTimeout(step2, 1000);
}
setTimeout(step1, 1000);
```

Clean, flat, readable 

## Callback Hell Breakdown

### The "Pyramid of Doom"

When callbacks are **nested within callbacks**, the code visually forms a **triangular or pyramid shape**:



```
doStep1(() => {
 doStep2(() => {
 doStep3(() => {
 doStep4(() => {
 // ...
 });
 });
 });
});
```

This pattern is hard to:

- **Read**
- **Debug**
- **Extend**

The deeper the pyramid, the darker the hell 😈

## Inversion of Control

You're handing **control of your logic** to another function. When you pass a callback to a function (e.g., `setTimeout`, `fs.readFile`, `doSomethingAsync`), you **trust** it to:

- Call your callback once
- Call it with the right data
- Handle failures correctly But if it misbehaves? You're screwed.

### Real Risk:

- Your callback might be called **twice**
- It might **never be called**
- You **lose control** over error handling and execution flow

## Summary

- ? Callback = Function passed & executed later
- 🗑️ Callback Hell = Nested async callbacks that are hard to maintain
- 🔄 Happens due to JS async nature + sequencing
- ✅ Solve with named functions, Promises, async/await

## Promises in JavaScript

### What is a Promise?

A **Promise** is an object that represents the eventual **completion (or failure)** of an asynchronous operation and its resulting value. It acts as a **placeholder** for a value that's not available *yet* but will be resolved in the future.

## 🧐 Why Promises?

Before Promises:

- We had **callback hell** 🌀 — messy, hard to maintain, deeply nested functions.
- Lack of **readability**, **composability**, and **error handling**.

## ✅ Promises Solve:

Problem	Promise Solution
Inversion of control	You decide when <code>.then()</code> runs, not API
Callback hell (pyramid of doom)	Promises allow clean chaining
No error propagation	Promises use <code>.catch()</code> for error flow

## 🔬 Promise Lifecycle

```
const promise = new Promise((resolve, reject) => {
 // async task
 if (success) {
 resolve(data); // fulfilled
 } else {
 reject(error); // rejected
 }
});
```

**States:**

- 🕒 *Pending* — initial state
- ✅ *Fulfilled* — `resolve()` was called
- ❌ *Rejected* — `reject()` was called
- 🗑️ *Settled* — either fulfilled or rejected

## 🦄 Consuming Promises

```
promise
 .then((data) => {
 // handle success
 })
 .catch((err) => {
```

```
// handle error
})
.finally(() => {
 // always runs
});
```

- `.then()` — handles **success**
- `.catch()` — handles **failure**
- `.finally()` — always runs (cleanup)

---

## Promise Chaining

```
doSomething()
 .then(result => doNext(result))
 .then(nextResult => finalStep(nextResult))
 .catch(err => handleError(err));
```

- Each `.then()` returns a **new promise**.
- Allows linear, readable async flows.

---

## Common Gotcha

If you **return nothing** in a `.then()`, the next `.then()` gets **undefined**.

```
fetchData()
 .then(res => {
 console.log(res);
 // no return here → next .then gets undefined
 })
 .then(data => console.log(data)); // undefined
```

---

## One-Time Use

Once a Promise is settled (resolved or rejected), its state is **immutable**.

```
const p = new Promise((res, rej) => {
 res("done");
 rej("fail"); // ignored
});
```

---

## Real Example

```
const cart = ["shoes", "pants"];
createOrder(cart)
 .then(orderId => proceedToPayment(orderId))
 .then(paymentInfo => showOrderSummary(paymentInfo))
 .catch(err => handleError(err));
```

✅ Each function returns a promise    ✅ Clean flow from creation → payment → summary

---

## 🔧 Problem Without Promises

```
createOrder(cart, function(orderId) {
 proceedToPayment(orderId, function(paymentInfo) {
 showOrderSummary(paymentInfo);
 });
});
```

### ⚠️ Inversion of Control:

- You're handing over control to the API.
  - If `createOrder` internally calls the callback multiple times, or with the wrong data — you're **screwed**.
  - Promises let **you control the chain**.
- 

## 🧠 Takeaways

- Promises are **contracts**: "I'll give you a result later."
  - They bring back **control, predictability, and composability**.
  - Don't just consume them — **understand how they're structured**.
- 

## Promise Practice

```
console.clear();

let cart = ["shoes", "pants"];
// let cart = [];

createOrder(cart)
 .then(orderId => proceedToPayment(orderId))
 .then(paymentInfo => showOrderSummary(paymentInfo))
 .then(orderSummary => updateWallet(orderSummary))
 .then(walletBalance => console.log(walletBalance))
 .catch(error => console.log(error))

function validateCart(cart) {
 if (cart.length) return true;
```

```
 return false;
 }

 function createOrder(cart) {
 let pr = new Promise((resolve, reject) => {
 if (!validateCart(cart)) {
 let err = new Error("your cart is empty!");
 reject(err);
 }
 const orderId = "12345";
 if (orderId) {
 setTimeout(() => {
 resolve(orderId);
 }, 2000);
 }
 });
 return pr;
 }

 function proceedToPayment(orderId) {
 console.log(orderId);
 let pr = new Promise((resolve, reject) => {
 setTimeout(() => {
 resolve("Payment is Successful!");
 }, 4000);
 // reject("Payment Failed!");
 });
 return pr;
 }

 function showOrderSummary(paymentInfo) {
 console.log(paymentInfo);
 let pr = new Promise((resolve, reject) => {
 setTimeout(() => {
 resolve("Order Summary");
 }, 1000);
 });
 return pr;
 }

 function updateWallet(orderSummary) {
 console.log(orderSummary);
 let pr = new Promise((resolve, reject) => {
 setTimeout(() => {
 resolve("Balance = 2 Billin Dollars.");
 }, 1000);
 });
 return pr;
 }
}
```

---

## ✳ Creating Promises, Chaining & Error Handling

## 1 Use Case: E-commerce Checkout Flow

- You have a `cart = ['shoes', 'pants', 'kurta']`.
- Goal:
  1. Validate cart
  2. Create order → get `orderId`
  3. Proceed to payment → get `paymentInfo`
  4. Handle errors cleanly along the way

## 2 Producer: Defining `createOrder(cart)`

```
function createOrder(cart) {
 return new Promise((resolve, reject) => {
 if (!validateCart(cart)) {
 reject(new Error("Cart is not valid")); // error path
 }
 const orderId = '12345';
 resolve(orderId); // success path
 });
}
```

- Uses `new Promise((resolve, reject) => { ... })`
- Calls `resolve()` or `reject()` only once, ensuring predictable behavior

## 3 Consumer: Handling the Promise

```
createOrder(cart)
 .then(orderId => proceedToPayment(orderId))
 .then(paymentInfo => console.log(paymentInfo))
 .catch(err => console.error("Error:", err));
```

- `.then()` receives resolved value
- `.catch()` handles any rejection in the chain

## 4 Flat Chaining vs. Nesting

- ☒ Correct (flat, returns promise in each `.then()`):

```
createOrder(cart)
 .then(orderId => {
 return proceedToPayment(orderId);
 })
```

```
.then(paymentInfo => {
 console.log(paymentInfo);
})
.catch(err => console.error(err));
```

Each `.then()` returns a new promise; the next `.then()` waits for it

- ✘ Wrong (chaining without `return` = breaks flow):

```
.then(orderId => {
 proceedToPayment(orderId);
})
```

Without `return`, `paymentInfo` in next `.then()` becomes `undefined`

---

## 5 Advanced: Multiple `.catch()` Positions

```
createOrder(cart)
 .then(orderId => {
 return proceedToPayment(orderId);
 })
 .catch(err => {
 console.warn("Payment failed, but continue:", err);
 })
 .then(() => {
 // Continues even if payment failed
 return showOrderSummary();
 })
 .catch(err => console.error("Fatal error:", err));
```

- Placing `.catch()` mid-chain **handles certain errors locally**, and the chain continues
- Good for partial error recovery (e.g., payment failed, but still show summary)

---

### ✓ (Bonus Insight)

1. **Promises fix Inversion of Control** – unlike callbacks, handing `.then()` keeps control in *your* hands
2. **Flat structure = readable + maintainable**
3. **Return every promise in `.then()`** – essential for proper chaining
4. **Catch placement matters:**
  - A single `.catch()` at end captures *all errors*
  - Mid-chain `.catch()` allows *partial recovery and continued flow*

---

## 🔑 Key Takeaways

- Always **create** promises using `new Promise(...)`



- Always **return inside** `.then()` when chaining to preserve chain
- Use `.catch()` **smartly** — at the end for fatal errors, or mid-chain to recover
- Promise chains are powerful tools to serialize async steps clearly

---

## Promise APIs in JavaScript

---

### ✓ Topics Covered:

- What are the 4 major Promise combinators?
- Use-cases & behavior in real-world async tasks
- Performance, behavior, and pitfalls
- Interview-level clarity

---

### 1. `Promise.all()`

#### What It Does:

Takes an array of promises → waits for all to resolve → returns an array of results in order. If **any** promise **rejects**, the whole `Promise.all()` **fails immediately** (Fail Fast).

#### Use-case:

Perfect for parallel API calls where you want **all results** to proceed.

#### Catch:

If one fails, others don't stop executing, but their result is ignored.

#### ✓ Example:

```
const p1 = new Promise(resolve => setTimeout(() => resolve('P1'), 3000));
const p2 = new Promise(resolve => setTimeout(() => resolve('P2'), 1000));
const p3 = new Promise(resolve => setTimeout(() => resolve('P3'), 2000));
Promise.all([p1, p2, p3])
 .then(results => console.log(results)) // ['P1', 'P2', 'P3']
 .catch(err => console.error(err));
```

#### ✗ Error Scenario:

```
const p2 = new Promise((_, reject) => setTimeout(() => reject('P2 Failed'),
1000));
Promise.all([p1, p2, p3])
 .then(results => console.log(results))
 .catch(err => console.error(err)); // 'P2 Failed' after 1s
```

---


## 2. `Promise.allSettled()`

### What It Does:

Waits for **all promises to settle** (resolve/reject) → returns status+value/reason of each.

### Use-case:

When you want the result of every promise regardless of success/failure.

 Safer alternative to `.all()` when rejection shouldn't break the chain.

### Example:

```
Promise.allSettled([p1, p2, p3])
 .then(results => console.log(results));
```

```
// Output:
[
 { status: 'fulfilled', value: 'P1' },
 { status: 'rejected', reason: 'P2 Failed' },
 { status: 'fulfilled', value: 'P3' }
]
```

---

## 3. `Promise.race()`

### What It Does:

Returns result of the **first promise** that settles (either resolve or reject).

### Use-case:

Timeouts, fallback APIs, race conditions, etc.

### Caveat:

If the **first** one fails → the entire `.race()` fails.

### Example:

```
Promise.race([p1, p2, p3])
 .then(result => console.log(result))
 .catch(err => console.error(err)); // Depends on fastest one
```

```
// p2 resolves in 1s → Output: "P2"
// If p3 rejected in 2s before others resolved → Output: "P3 Failed"
```

## 4. Promise.any()

### What It Does:

Returns the result of the **first successful promise**.

### Use-case:

When **only one** success is required, and rejections can be ignored (initially).

### Example:

```
Promise.any([p1, p2, p3])
 .then(result => console.log(result)) // returns first fulfilled one
 .catch(err => {
 console.error(err); // AggregateError if all failed
 console.error(err.errors); // ['P1 Fail', 'P2 Fail', 'P3 Fail']
 });
```

### Note:

- It only **fails** if **all promises fail**.
- Introduced in ES2021.

## Key Differences Recap:

Method	Returns When?	Success Condition	Fails When?
Promise.all	All resolved	All promises must resolve	Any one fails (fast fail)
Promise.allSettled	All settled (resolve/reject)	Always resolves	Never fails
Promise.race	First to settle (resolve or reject)	Any one settles	On first rejection (if it comes first)
Promise.any	First to resolve	Any one resolves	Only if all are rejected (AggregateError)

## Advanced Insight: Execution Timing

Even though `await` causes execution to pause within the async function, **timers for promises start when declared**, not at the `await` line. So this:

```
const p1 = new Promise(resolve => setTimeout(() => resolve("p1"), 5000));
const p2 = new Promise(resolve => setTimeout(() => resolve("p2"), 10000));
async function handle() {
 const val1 = await p1;
 const val2 = await p2;
 console.log(val1, val2);
}
```

⌚ Will take **10 seconds**, not 15 — because both `p1` and `p2` started ticking at declaration. But this will take 15s:

```
async function handle() {
 const p1 = new Promise(resolve => setTimeout(() => resolve("p1"), 5000));
 const val1 = await p1;
 const p2 = new Promise(resolve => setTimeout(() => resolve("p2"), 10000));
 const val2 = await p2;
 console.log(val1, val2);
}
```

⌚ Because `p2` doesn't start until after `p1` resolves.

---

## 🔗 Real-World Application Patterns

Pattern	Use-Case
<code>Promise.all()</code>	Fetch multiple product data in parallel
<code>Promise.allSettled()</code>	Form submission with optional fields
<code>Promise.race()</code>	Show fallback loader if API takes too long
<code>Promise.any()</code>	Fastest mirror server from a list

---

## 🔧 Interview-Level Tips

- `Promise.all()` is fast but brittle. One fail, all fail.
- `allSettled()` is best for audit logs, batch jobs.
- `race()` is great for adding timeouts.
- `any()` is best when only one success is enough.

### 🔧 Bonus:

- Use `AbortController` + `race()` for fetch timeouts.
- Always add `.catch()` or use `try-catch` in `async/await`.

---

## Summary

```
// Most safe:
Promise.allSettled()
// Fastest result (success/fail):
Promise.race()
// Fastest successful result:
Promise.any()
// Wait for all but risky if one fails:
Promise.all()
```

---

## `async/await` in JavaScript

### What is `async`?

- `async` keyword is used to define a function that always returns a **Promise**, even if it returns a simple value.
- It allows use of the `await` keyword inside it.

```
async function getData() {
 return "Hello";
}
getData().then(console.log); // Promise resolved with "Hello"
```

### What is `await`?

- Used **only inside async functions**.
- Pauses the function execution at that line until the awaited Promise resolves.
- Makes asynchronous code look synchronous — increasing readability.

```
async function fetchData() {
 const result = await somePromise;
 console.log(result);
}
```

---

## How `async/await` works behind the scenes?

### `async/await` is **syntactic sugar over Promises**

When JS encounters `await`:

1. The async function execution is **suspended**.
2. The JS engine **removes** the function from the call stack.
3. The rest of the code runs.
4. When the awaited Promise resolves, the function is **resumed from the same line**. ⊖ It doesn't block the main thread. ✓ No freezing happens — call stack is free.

## Real Example

```
const p1 = new Promise(res => setTimeout(() => res("P1 Done"), 5000));
const p2 = new Promise(res => setTimeout(() => res("P2 Done"), 10000));
async function handle() {
 console.log("Start");
 const res1 = await p1;
 console.log(res1); // after 5s
 const res2 = await p2;
 console.log(res2); // after total 15s
}
```

### ⚠ Common Confusion:

"Shouldn't this take 15s total if p2 starts when the line is reached?" No. Because the timer for **p2** started *when it was defined*, **not when await is reached**. ⚙ **Timer starts at declaration** So:

- If both **p1** and **p2** are declared outside the async function: ✓ **Parallel execution** → total 10s
- If you move the declaration of **p2 after await p1**, the timer starts later ✗ **Sequential execution** → total 15s

```
async function handle() {
 const p1 = new Promise(res => setTimeout(() => res("P1"), 5000));
 const res1 = await p1;
 const p2 = new Promise(res => setTimeout(() => res("P2"), 10000));
 const res2 = await p2;
 console.log(res1, res2); // total 15s
}
```

## 🧐 Error Handling in async/await

### ✓ try...catch block

```
async function fetchData() {
 try {
 const response = await fetch("https://api.github.com/users/sidharthjuyal");
 const data = await response.json();
 console.log(data);
 } catch (err) {
```

```
 console.error("Something went wrong", err);
 }
}
```

## Async/Await vs .then()/ .catch()

Feature	.then() / .catch()	async/await
Readability	Less readable with multiple chains	Cleaner, looks like synchronous
Error Handling	With .catch()	With try...catch
Blocking	Non-blocking	Also non-blocking (execution is suspended, not blocked)

## Deep Concept: Is JS really “waiting”?

No. When `await` is encountered:

- The `async` function is **paused**.
- JS engine continues executing other code.
- Once the Promise resolves, function is **resumed from where it paused**.💡 This is what makes JS appear synchronous, while actually staying non-blocking.

## Summary – `async/await`

- `async` marks a function to always return a Promise.
- `await` pauses execution until a Promise resolves.
- Execution context is **suspended**, not blocked.
- **Error handling** is done using `try...catch`.
- For true **parallel execution**, declare promises **before awaiting** them.

## `this` Keyword in JavaScript

The `this` keyword refers to **the object that is executing the current function**. Its value depends entirely on the **execution context**, i.e., **how a function is called, not where it's defined**.

### 1. `this` in Global Context

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

- In the global space (top-level code), **this** points to the **global object**.
    - In browser → **window**
    - In Node.js → **global**
- 



## 2. **this** Inside Regular Functions

```
function test() {
 console.log(this);
}
test();
```

- In **non-strict mode**, **this** refers to the **global object**.
- In **strict mode**, **this** is **undefined**.

```
"use strict";
function test() {
 console.log(this); // undefined
}
test();
```



### **this** Substitution Rule:

In **non-strict mode**, if **this** is **null** or **undefined**, it is **automatically substituted** with the global object.

---



## 3. **this** in Object Methods

```
const user = {
 name: "Vex",
 greet() {
 console.log(this); // refers to `user` object
 console.log(this.name); // Vex
 }
};
user.greet();
```

- When a function is called as a method of an object, **this** refers to that **object**.
- 



## 4. **this** with **call**, **apply**, and **bind**

These methods are used to explicitly **set the value of this** in a function.

---



```
const person = {
 name: "Sid",
 printName() {
 console.log(this.name);
 }
};
const person2 = { name: "Vex" };
person.printName(); // Sid
person.printName.call(person2); // Vex
```

- `call(obj)` → invokes the function immediately with `this = obj`.
- `apply(obj, argsArray)` → same as `call`, but takes arguments as an array.
- `bind(obj)` → returns a **new function** with `this = obj`.

```
const boundFn = person.printName.bind(person2);
boundFn(); // Vex
```

## 5. `this` in Arrow Functions

Arrow functions do **not have their own `this`**. They **inherit `this` from their lexical (outer) context**.

```
const obj = {
 name: "Alok",
 arrowFn: () => {
 console.log(this.name); // undefined (global or window)
 }
};
obj.arrowFn();
```

Compare with:

```
const obj = {
 name: "Alok",
 regularFn() {
 const arrowFn = () => {
 console.log(this.name); // Alok
 };
 arrowFn();
 }
};
obj.regularFn();
```

 Remember:

Arrow functions capture **this** from **where they are defined**, not where they are called.

## 6. **this** in Event Listeners / DOM

```
<button onclick="console.log(this)">Click Me</button>
```

- this** refers to the **DOM element** that triggered the event (i.e., the button). Equivalent in JS:

```
document.querySelector("button").addEventListener("click", function () {
 console.log(this); // the <button> element
});
```

But with arrow function:

```
document.querySelector("button").addEventListener("click", () => {
 console.log(this); // refers to the enclosing lexical `this` (not the button!)
});
```

## Common Pitfalls

Situation	<b>this</b> Value
Global code	<b>window</b> (in browser)
Inside regular function (non-strict)	<b>window</b>
Inside regular function (strict)	<b>undefined</b>
Method in object	The object itself
Arrow function	Lexical (outer) <b>this</b>
DOM event (regular function)	The HTML element
DOM event (arrow function)	Outer <b>this</b> (not DOM element)
With <b>.call()</b> , <b>.apply()</b> , <b>.bind()</b>	Explicitly set by user

## Interview-Level Examples

Q: What's printed here?

```
const user = {
 name: "Sid",
```

```
greet: function () {
 setTimeout(function () {
 console.log(this.name);
 }, 1000);
}
};
user.greet();
```

A: `undefined` (because `this` inside `setTimeout` is global, not `user`) ☒ Fix with arrow function:

```
setTimeout(() => {
 console.log(this.name); // Sid
, 1000);
```

---

## ✨ Key Takeaways

- Always understand the **calling context** to predict `this`.
  - Avoid `this` inside nested regular functions — prefer arrow functions when appropriate.
  - Use `bind` to fix `this` in callbacks.
  - Arrow functions are **not suited** for methods or event listeners where `this` is required.
-