

## Execution Context in JavaScript

Everything in JavaScript happens inside the Execution Context. Think of it as a sealed-off container where JavaScript runs. It is an abstract concept that holds information about the environment in which the current code is being executed.

Inside the execution context, there are two main components:

### 1. Memory Component (Variable Environment)

- Stores variables and function declarations in key-value pairs.
- During the creation phase, memory is allocated for variables (**undefined** initially) and functions (stored entirely).

### 2. Code Component (Thread of Execution)

- Executes the code one line at a time.
- JavaScript follows a synchronous execution model, meaning it runs one command at a time in a specific order.
- It is also known as the Thread of Execution.

## JavaScript: Synchronous & Single-Threaded

- Synchronous: Executes code line by line, in the order it appears.
  - Single-threaded: Can only run one task at a time in a single execution thread.
- 

## How JavaScript is Executed & The Call Stack

When a JavaScript program runs, a **Global Execution Context (GEC)** is created. The execution context is created in two phases:

### 1. Memory Creation Phase (Creation Phase)

- JS allocates memory to variables and functions.

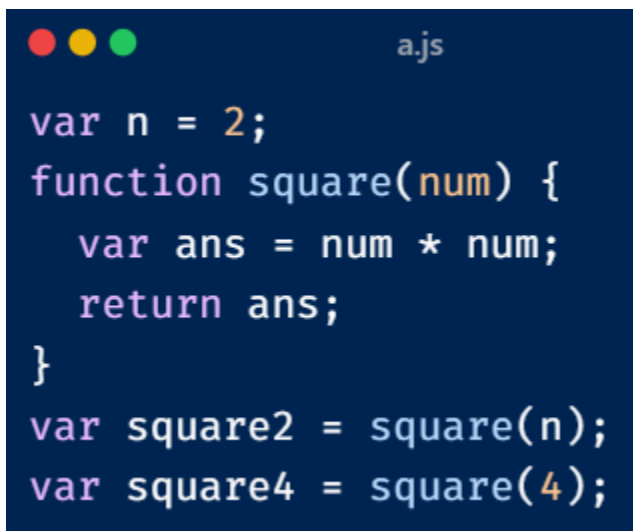
- Variables are assigned **undefined**, while functions are stored entirely in memory.

## 2. Code Execution Phase

- JS executes the code line by line, updating variable values and calling functions.

---

## Example & Execution Steps



```
a.js
var n = 2;
function square(num) {
  var ans = num * num;
  return ans;
}
var square2 = square(n);
var square4 = square(4);
```

### Step 1: Memory Creation Phase

- Line 1: **n** is allocated memory and initialized as **undefined**.
- Line 2-5: **square** function is stored in memory (entire function body).
- Line 6 & 7: **square2** and **square4** are allocated memory and initialized as **undefined**.

### After Memory Creation Phase:



```
1  n → undefined
2  square → function definition
3  square2 → undefined
4  square4 → undefined
```

## Step 2: Code Execution Phase

- Line 1: `n = 2` (updates value of `n`).
- Line 6: `square(n)` is invoked → a new Function Execution Context (FEC) is created.
  - Memory Creation (inside FEC): `num` and `ans` are allocated memory (`undefined`).
  - Code Execution (inside FEC): `num = 2`, then `ans = 2 * 2 = 4`.
  - Function returns `4`, assigns `square2 = 4`, and execution context is destroyed.
- Line 7: `square(4)` is invoked → a new Function Execution Context (FEC) is created.
  - Memory Creation (inside FEC): `num` and `ans` are allocated (`undefined`).
  - Code Execution (inside FEC): `num = 4`, then `ans = 4 * 4 = 16`.
  - Function returns `16`, assigns `square4 = 16`, and execution context is destroyed.

## Final State Before Global Execution Context is Removed:



```
1  n → 2
2  square → function definition
3  square2 → 4
4  square4 → 16
```

---

## Call Stack & Execution Context


JavaScript manages execution context creation and deletion using the Call Stack.

- Call Stack is a mechanism that keeps track of the execution order of functions.
- It follows LIFO (Last In, First Out):
  1. When a function is called, a new execution context is pushed onto the stack.
  2. When the function finishes, its execution context is popped off the stack.

### Call Stack Execution for the Example:

1. Global Execution Context (GEC) is created.
  2. `square(n)` is called → new Function Execution Context (FEC) is pushed.
  3. `square(n)` finishes → FEC is popped.
  4. `square(4)` is called → new Function Execution Context (FEC) is pushed.
  5. `square(4)` finishes → FEC is popped.
- Global Execution Context (GEC) is removed after execution ends.

### Call Stack at Different Stages:



```
1  [Global Execution Context]  // Initial
2  [Global → square(n)]       // square(n) called
3  [Global]                   // square(n) finished
4  [Global → square(4)]       // square(4) called
5  [Global]                   // square(4) finished
6  // Call Stack is empty → Program ends
7
```

---

## Hoisting in JavaScript (Variables & Functions)

### Example 1: Hoisting Behavior

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

#### Explanation:

- In many programming languages, calling a function or accessing a variable before declaration would result in an error.
- However, in JavaScript, due to Hoisting, this does not cause an error.
- During the Memory Creation Phase of the Execution Context:
  - Variables declared with `var` are assigned `undefined`.
  - Function declarations are stored in memory with their complete definition.
- During Execution Phase:
  - `getName()` executes successfully because the function is already stored in memory.
  - `console.log(x)` prints `undefined` because `x` exists in memory but hasn't been assigned `7` yet.

### Hoisting Concept

Hoisting is JavaScript's behavior where variables and function declarations are moved to the top of their scope before execution.

### Key Takeaways:

- ✓ Variables (**var**) are hoisted but initialized as **undefined** until assigned a value.
  - ✓ Function declarations are fully hoisted, meaning they can be called even before they are defined.
  - ✓ **let** and **const** variables are hoisted but not initialized (they remain in a Temporal Dead Zone).
- 

### Example 2: Hoisting with var and Function Expressions

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

### Explanation:

- Difference Between Function Declaration & Function Expression:
    - A Function Declaration (**function getName() {}**) is hoisted entirely.
    - A Function Expression (**var getName = function() {}**) is hoisted only as a variable, meaning it is initialized as **undefined**.
  - During Memory Creation Phase:
    - **var getName** is hoisted and assigned **undefined**.
  - During Execution Phase:
    - **getName();** throws **TypeError: getName is not a function** because **getName** is still **undefined** at that point.
- 

### Example 3: Hoisting with **let** and **const**



```
1 console.log(a); // ReferenceError: Cannot access 'a' before initialization
2 let a = 5;
```

## Explanation:

- Unlike `var`, variables declared with `let` and `const` are hoisted but not initialized.
- They remain in a Temporal Dead Zone (TDZ) until they are assigned a value.
- Accessing `a` before its declaration results in `ReferenceError`.

---

## Summary of Hoisting Rules:

Type	Hoisted?	Initializ ed?	Accessible Before Declaration?
<code>var</code>	✓ Yes	✓ As undefined	✓ Allowed (but undefined)
<code>let</code>	✓ Yes	✗ No	✗ ReferenceError
<code>const</code>	✓ Yes	✗ No	✗ ReferenceError

Function Declaration	✓ Yes (Entire Function)	✓ Yes	✓ Allowed
Function Expression ( <code>var fn = function() {}</code> )	✓ Yes (Only Variable)	✓ As <code>undefined</code>	✗ TypeError

```

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

function a() {
  var x = 10; // Local scope due to separate execution context
  console.log(x);
}

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

```

### Key Takeaways:

- ✓ Functions declared using `function` are fully hoisted.
- ✓ Variables declared with `var` are hoisted but initialized as `undefined`.



- ✓ Function expressions and arrow functions using `var` behave like variables and are hoisted as `undefined`.
  - ✓ Variables declared with `let` and `const` are hoisted but not initialized, causing a `ReferenceError` if accessed before declaration.
- 

## Functions and Variable Environments in JavaScript

### Example Code & Output

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

function a() {
  var x = 10; // Local scope due to separate execution context
  console.log(x);
}

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

### Output:

```
10
100
1
```

## Code Flow in Terms of Execution Context

When the program runs, JavaScript follows these steps:

### 1. Global Execution Context (GEC) is Created

- GEC is pushed onto the Call Stack.
- The execution context has two phases:
  - Memory Creation Phase (Phase 1):
    - `x` is allocated memory and initialized as `undefined`.
    - `a` and `b` store their entire function definitions.
  - Code Execution Phase (Phase 2):
    - `x = 1` is assigned in the global scope.
    - `a()` is called.

Call Stack at this point: [GEC]

---

### Execution of `a()`

- A new Execution Context (EC) for `a()` is created.
- It follows the same two-phase process:
  - Phase 1:
    - A new `x` is allocated in `a`'s local memory (initialized as `undefined`).
  - Phase 2:
    - `x = 10` is assigned inside `a()`.
    - `console.log(x);` prints `10`.
- After execution, `a()`'s Execution Context is removed from the Call Stack.

Call Stack at this point: [GEC]

---

### Execution of `b()`

- A new Execution Context (EC) for `b()` is created.

- Again, two-phase execution happens:
  - Phase 1:
    - A new `x` is allocated in `b`'s local memory (initialized as `undefined`).
  - Phase 2:
    - `x = 100` is assigned inside `b()`.
    - `console.log(x);` prints `100`.
- After execution, `b()`'s Execution Context is removed from the Call Stack.

Call Stack at this point: [GEC]

---

Final Step: Global Execution Resumes

- `console.log(x);` executes in the Global Execution Context, printing `1`.
- Finally, the Global Execution Context is removed, and the Call Stack is empty.

Final Call Stack: [Empty] // Program ends

---

## Key Takeaways

- ✓ Each function call creates a new Execution Context with its own variable environment.
- ✓ Local variables inside functions do not affect the global variable `x` (different execution contexts).
- ✓ Execution contexts are managed using the Call Stack (LIFO principle).
- ✓ Functions are hoisted in memory before execution starts, allowing calls before their definitions.

---

## Shortest JavaScript Program, window & this Keyword

### Shortest JavaScript Program

- The shortest JS program is an empty file.
- Even with an empty file, the JavaScript engine executes some default processes:
  1. Creates the Global Execution Context (GEC).
  2. Defines the `window` object (in browsers).
  3. Defines the `'this'` keyword, which points to `'window'` at the global level.

---

### The `window` Object

- The `window` object is automatically created in the global scope in browsers.
- It acts as a global container for built-in functions, variables, and objects (e.g., `console`, `setTimeout`, `alert`).
- In different JavaScript environments, the global object has different names:
  - Browsers: `window`
  - Node.js: `global`
  - Web Workers: `self`

---

### The `this` Keyword at Global Scope

- In the global scope (outside functions), `this` refers to the global object.

In browsers:

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

- Any variable declared using `var` at the global level automatically becomes a property of `'window'`.

Example:

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

- Here, `x` is attached to `window`, meaning `this.x` and `window.x` both output `10`.
- 

## Key Takeaways

- ✓ The shortest JS program (empty file) still creates a Global Execution Context (GEC).
  - ✓ The `window` object is the global object in browsers.
  - ✓ `this` refers to `window` in the global scope in browsers.
  - ✓ Global `var` variables attach to `window`, but `let` and `const` do not.
- 

## Undefined vs Not Defined in JavaScript

### 1. `undefined` in JavaScript

- During the memory allocation phase, JavaScript assigns `undefined` to variables that are declared but not assigned a value.
- `undefined` means the variable exists in memory but has no value yet.

Example:

```
console.log(x); // undefined (Memory allocated, but no value yet)
var x = 25;
console.log(x); // 25
```

Here, `x` is hoisted with an initial value of `undefined`, then later assigned `25`.

---

## 2. `not defined` in JavaScript

- If a variable is never declared, trying to access it throws a `ReferenceError`.
- "Not defined" means the variable doesn't exist in memory at all.

Example:

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

Since `a` was never declared, JavaScript doesn't recognize it and throws an error.

---

### Key Differences: `undefined` vs `not defined`

Concept	Meaning
<code>undefined</code>	Variable is declared but not assigned a value.
<code>not defined</code>	Variable is not declared anywhere in the code.

Throws Error?	<code>undefined</code> → No error. <code>not defined</code> → <code>ReferenceError</code> .
---------------	---

---

### 3. JavaScript is a Loosely Typed Language

- Variables are not bound to a specific data type.
- You can reassign different types to the same variable.

Example:

```
var a = 5;
a = true;
a = "hello";
console.log(a); // "hello"
```

---

Best Practice

- ✗ Never manually assign `undefined` to a variable.
  - ✓ Let JavaScript handle it during memory allocation.
- 

## Scope Chain, Scope & Lexical Environment in JavaScript

### 1. Scope in JavaScript

Scope determines where variables and functions can be accessed in the code. It is directly related to the Lexical Environment.

---

## 2. Lexical Environment

A Lexical Environment consists of:

- ✓ Local memory (variables and functions declared inside a function).
- ✓ A reference to the Lexical Environment of its parent (outer scope).

- Lexical means "in order" → Code structure determines variable access.
  - Every Execution Context has a Lexical Environment.
  - Scope chain follows the Lexical Environment hierarchy to find variables.
- 

## 3. Scope Chain & Execution Context

When a variable is accessed, JavaScript searches:

- 1 Current function's local memory (if declared inside function).
  - 2 Parent function's Lexical Environment (if not found locally).
  - 3 Global Execution Context (GEC) (if not found in any parent scope).
  - 4 Throws ReferenceError if the variable is not defined anywhere.
- 

## 4. Examples & Outputs

### Case 1: Function Accessing a Global Variable

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

- ✓ Output: 10
- ✓ Explanation: Function `a()` accesses `b` from the global scope.



---

### Case 2: Nested Function Accessing a Global Variable

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

✓ Output: 10

✓ Explanation: `c()` has access to `b` from the global scope due to the scope chain.

---

### Case 3: Local Variable Overriding Global Variable

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

✓ Output: 100

✓ Explanation: Local variable `b = 100` inside `c()` takes precedence over global `b`.

---

#### Case 4: Function Cannot Access Variables from Another Execution Context

```
function a() {  
  var b = 10;  
  c();  
  function c() {  
    console.log(b); // 10  
  }  
}  
a();  
console.log(b); // ReferenceError: b is not defined
```

✓ Output:

- 10 (inside `c()`)
  - Error: `b is not defined` (outside function).  
 ✓ Explanation:
    - `b` exists inside function `a()`, not in the global scope.
    - Trying to access `b` globally results in `ReferenceError`.
- 

## 5. Call Stack & Lexical Environment Representation

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

### Call Stack:

[GEC] → [a()] → [c()]

### Lexical Environment Mapping:

- ✓ c() = [b:10, lexical parent → a()]
- ✓ a() = [b:10, c(), lexical parent → GEC]
- ✓ GEC = [a(), lexical parent → null]

### Scope Chain Process:

- ✓ c() looks for **b** in its local scope → Not found.
- ✓ Moves to **a()**'s Lexical Environment → **b = 10** found ✓.
- ✓ Prints **10**.

---

## 6. Key Takeaways

- ✓ **Scope Chain:** Inner functions can access variables from outer functions.
- ✓ **Lexical Scope:** A function's scope is determined by where it is defined, not where it is called.
- ✓ **Global Scope:** Variables in the global scope can be accessed from anywhere.
- ✓ **Local Scope:** Variables inside a function cannot be accessed outside it.

---

## 7. TL;DR

- 📌 Lexical Environment = Local Memory + Parent's Lexical Environment.
  - 📌 Scope Chain follows parent scopes to find variables.
  - 📌 Inner functions have access to outer function variables (but not vice versa).
- 

## let & const in JavaScript & Temporal Dead Zone (TDZ)

### 1. Difference Between var, let, and const in Hoisting

- ✅ All variables (**var**, **let**, **const**) are hoisted but behave differently.
- ✅ **var** is initialized as **undefined** in the global execution context.
- ✅ **let** and **const** are hoisted but remain in the **Temporal Dead Zone (TDZ)** until assigned a value.

Example:

```
console.log(a); // ❌ ReferenceError: Cannot access 'a' before initialization
console.log(b); // ✅ undefined (var is hoisted)
let a = 10;
var b = 15;
console.log(a); // ✅ 10
console.log(window.a); // ✅ undefined (let does not attach to window)
console.log(window.b); // ✅ 15 (var attaches to window)
```

- ✅ **var b = 15;** is stored in the global memory (window object).
  - ✅ **let a = 10;** is stored in a separate script memory and isn't accessible until initialized.
  - ✅ Accessing **a** before initialization leads to a **ReferenceError** (TDZ issue).
-

## 2. Temporal Dead Zone (TDZ)

TDZ is the period between the variable's hoisting and its initialization.

Example:

```
console.log(a); // ❌ ReferenceError (TDZ)
let a = 10;
console.log(a); // ✅ 10
```

### Explanation:

- 1 `let a` is hoisted but not initialized → Stays in TDZ.
- 2 Accessing `a` before initialization → `ReferenceError`.
- 3 Once `a` is assigned a value (`a = 10`), it exits the TDZ and can be used.

---

## 3. Common Errors with `let` & `const`

Error	Cause	Example
ReferenceError	Accessing <code>let</code> before initialization (TDZ issue)	<code>console.log(a);</code> <code>let a = 10;</code>
SyntaxError	Redeclaring a variable in the same scope	<code>let a = 10;</code>  <code>let a = 100;</code>
SyntaxError	<code>const</code> must be initialized at declaration	<code>const x;</code>

TypeError	Reassigning a <code>const</code> variable	<code>const x = 10; x = 20;</code>
-----------	---	------------------------------------

#### 4. `const` is Stricter Than `let`

✓ `let` allows declaration without initialization

```
let x;
x = 10; // ✓ No error
console.log(x); // ✓ 10
```

✓ `const` requires immediate initialization

```
const y; // ✗ SyntaxError: Missing initializer in const declaration
y = 10;
```

✓ Reassigning a `const` variable throws a `TypeError`

```
const z = 100;
z = 200; // ✗ TypeError: Assignment to constant variable
```

#### 5. Best Practices

- ✓ Use `const` by default
- ✓ If reassigning is needed, use `let`
- ✓ Avoid `var` completely

✓ Declare and initialize variables at the top to shrink the TDZ

---

## 6. TL;DR

📌 `var` is hoisted with `undefined`, while `let` & `const` remain in TDZ.

📌 TDZ = time from hoisting until initialization.

📌 `const` must be initialized at declaration and cannot be reassigned.

📌 Use `const` whenever possible, otherwise use `let`, avoid `var`.

---

## Block Scope & Shadowing in JavaScript

### 1. What is a Block?

✓ A block (or compound statement) groups multiple JavaScript statements together using `{ ... }`.

✓ Variables declared inside a block have different scopes:

- `var` is hoisted globally (function/global scope).
- `let` and `const` are hoisted in block scope.

### Example:

```
{  
  var a = 10;  
  let b = 20;  
  const c = 30;  
}  
  
console.log(a); // ✅ 10 (accessible, since var is globally scoped)  
console.log(b); // ❌ ReferenceError: b is ↓ defined  
console.log(c); // ❌ ReferenceError: c is not defined
```

### 🔧 Explanation:

- ✅ **var a** is stored in the global scope, so it remains accessible after the block.
  - ❌ **let b** and **const c** are block-scoped, meaning they exist only within {}.
- 

## 2. What is Shadowing?

✅ Shadowing occurs when a variable in a nested scope has the same name as a variable in the outer scope.

✅ The inner variable temporarily "overrides" the outer one within its scope.

Example (with **var**):



```
var a = 100;
{
  var a = 10; // Overrides global 'a'
  let b = 20;
  const c = 30;
  console.log(a); // ✅ 10 (inner 'a' replaces global 'a')
}
console.log(a); // ✅ 10 (global 'a' was modified)
```

### 🔧 Why does this happen?

- `var a = 100` is declared in the global scope.
  - Inside `{}` when `var a = 10` is declared, it does not create a new variable in block scope (since `var` is global).
  - Instead, it overwrites the global `a`.
- 

### 3. Shadowing with `let` & `const` (Safer Approach)

- ✅ `let` and `const` do NOT override outer variables.
- ✅ Each `let/const` exists in its own scope.

Example (with `let`):

```
let b = 100;
{
  var a = 10;
  let b = 20;
  const c = 30;
  console.log(b); // ✅ 20 (inside block scope)
}
console.log(b); // ✅ 100 (original `b` remains unchanged)
```

### 🔧 Why does this happen?

- `let b = 20` inside `{}` only exists within the block scope.
- `let b = 100` outside the block remains untouched.

✅ Same logic applies to functions:

```
const c = 100;
function x() {
  const c = 10;
  console.log(c); // ✅ 10 (function `c` is used)
}
x();
console.log(c); // ✅ 100 (global `c` is untouched)
```

---

## 4. Illegal Shadowing

💣 Illegal Shadowing happens when `var` tries to override a `let` or `const` variable within the same scope.

### Example (Throws Error)

```
let a = 20;
{
  var a = 30; // ❌ SyntaxError: Identifier 'a' has already been declared
}
```

✅ However, shadowing `var` with `let` is valid:

```
var a = 20;
{
  let a = 30; // ✅ Allowed
  console.log(a); // ✅ 30 (block-scoped `a`)
}
console.log(a); // ✅ 20 (global `a` remains unchanged)
```

---

## 5. Function Scope & Shadowing

✅ Functions follow the same scope rules as blocks. ✅ **var** is function-scoped, so it does NOT cause illegal shadowing inside functions.

Example:

```
let a = 20;
function x() {
  var a = 30; // ✅ No error, function scope allows it
}
```

---

## 6. TL;DR

📌 **var** is globally/function-scoped & can be overridden within blocks.

📌 **let** and **const** are block-scoped & prevent accidental overwrites.

📌 Shadowing is allowed with **let/const**, but **var** can cause issues.

📌 Illegal shadowing happens when **var** tries to override a **let/const**.

🚀 Best Practice: Avoid **var**, prefer **let** or **const** for predictable scoping!

---

## Closures in JavaScript

### What is a Closure?

A closure is a function that remembers the variables from its outer scope even after the outer function has finished executing.

### Example of Closure:

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

### Explanation:

- `y()` retains access to `a` even after `x()` has been executed.
- `z` holds the function `y` along with its lexical scope.
- When `z()` is called, it still remembers `a` from `x()`.

---

## Lexical Scope & Closures

### JavaScript follows lexical scoping, meaning:

1. A function first looks for variables in its local scope.
2. If not found, it looks at its parent function's scope.

3. This continues until it reaches the global scope.

### Nested Closures Example

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

- `y()` remembers both `a` (from `x()`) and `b` (from `z()`).
- This is because `y()` carries its lexical scope.

---

### Advantages of Closures

Closures are widely used in JavaScript to maintain state, encapsulate data, and optimize performance.

#### 1. Module Pattern (Encapsulation)



- ✓ Used to create private variables and methods.

```
const authModule = (function () {  
  let loggedInUser = null;  
  
  function login(username) {  
    loggedInUser = username;  
  }  
  
  function logout() {  
    loggedInUser = null;  
  }  
  
  function getUserInfo() {  
    return loggedInUser;  
  }  
}
```

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

*// Usage*

```
authModule.login('john_doe');  
console.log(authModule.getUserInfo()); // 'john_doe'  
authModule.logout();  
console.log(authModule.getUserInfo()); // null
```

-  Prevents global variable pollution.
-  Keeps `loggedInUser` private.
-  Provides controlled access using returned methods.

---

## 2. Currying (Functional Programming)

✓ Transforms a function with multiple arguments into a series of functions.

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

const addSalesTax = calculateTotalPrice(8);
console.log(addSalesTax(100)); // 108
```

- ✓ Allows partial function application.
  - ✓ Improves code flexibility.
- 

### 3. Memoization (Performance Optimization)

✓ Caches function results to avoid redundant 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
```

- ✓ Optimizes recursive functions.
  - ✓ Reduces redundant calculations.
- 

### 4. Data Hiding & Encapsulation

✓ Hides internal data using closures.

```
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 must be declared in an enclosing class
```

- ✅ Prevents direct access to `#name`.
  - ✅ Encapsulates data safely.
- 

## 5. Using Closures with `setTimeout`

✅ Closures allow functions to remember values even after a delay.

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

showMessage('Hello, world!', 2000); // Logs after 2 seconds
```

- ✅ Used in asynchronous programming.
  - ✅ Useful in animations and event handling.
- 

## Disadvantages of Closures

🔥 Closures can lead to performance issues if not handled properly.



### 1. Increased Memory Consumption 🛑

- Unused variables stay in memory, leading to higher memory usage.

### 2. Memory Leaks 🛑

- If a closure references a large object, it prevents garbage collection.

### 3. Browser Freezing 🛑

- Closures in loops without careful handling can cause infinite memory retention.

#### Example of a Memory Leak

```
function createLeak() {  
  let largeData = new Array(1000000).fill("Memory Leak!");  
  return function () {  
    console.log(largeData.length);  
  };  
}  
  
const leakyFunction = createLeak(); // largeData stays in memory
```

✅ Solution: Avoid unnecessary closures or explicitly set variables to `null`.

---

#### 🚀 Best Practices for Closures

- ✅ Use closures wisely to prevent memory leaks.
  - ✅ Release memory when done (`null` assignment).
  - ✅ Use them in modular design patterns for maintainability.
  - ✅ Use `let/const` to prevent accidental overwrites in loops.
-

## Real-World Analogy

Think of a closure as a packed lunch. You make a sandwich at home (the outer function), put it in a lunchbox (the closure), and take it to work (the inner function). Even though you're not at home anymore, you can still eat the sandwich because it's with you.

## 🎯 Summary

📌 Closures allow functions to "remember" variables even after execution.

📌 They enable encapsulation, memoization, currying, and async operations.

📌 Improper use may cause memory leaks & performance issues.

📌 Use closures strategically for efficient JavaScript code! 🚀

---

## First-Class Functions & Anonymous Functions in JavaScript

### Functions: The Heart ♥ of JavaScript

In JavaScript, functions are first-class citizens, meaning they can be:

- ✓ Assigned to variables
- ✓ Passed as arguments
- ✓ Returned from other functions

---

## Function Statement (Function Declaration)

A function statement is a standard way of defining a function.

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

✅ Hoisted: Available before execution.

---

## Function Expression

A function can be assigned to a variable, making it a function expression.

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

✅ Function as a value.

❌ Not Hoisted: Cannot be used before declaration.

---

## Difference: Function Statement vs. Function Expression

```
a(); // ✅ "Hello A"  
b(); // ❌ TypeError: b is not a function  
  
function a() {  
  console.log("Hello A");  
}  
  
var b = function () {  
  console.log("Hello B");  
};
```

Why?

- `a()` is hoisted with its function definition.

- `b()` is stored as `undefined` during memory allocation, leading to an error when called before definition.
- 

## Anonymous Function

A function without a name.

```
function () {  
    console.log("Anonymous");  
}; // ❌ Syntax Error (Function statement needs a name)
```

✓ Used inside function expressions

```
var b = function () {  
    console.log("I am anonymous!");  
};  
b(); // ✅ Output: I am anonymous!
```

---

## Named Function Expression

A function expression with a name.


```
var b = function xyz() {  
    console.log("b called");  
};  
  
b(); // ✅ "b called"  
xyz(); // ❌ ReferenceError: xyz is not defined
```

Why?

- `xyz` is not available in the global scope.

- It is only accessible inside the function itself.
- 

## Parameters vs. Arguments

```
var b = function (param1, param2) {  
  console.log("Parameters: ", param1, param2);  
};  
  
b("arg1", "arg2"); //  Arguments: "arg1", "arg2"
```


- ✓ Parameters: Defined in function declaration (param1, param2).
  - ✓ Arguments: Passed when calling the function ("arg1", "arg2").
- 

## First-Class Functions (First-Class Citizens)

JavaScript treats functions like any other value:

- ① Functions can be passed as arguments
- ② Functions can be returned from other functions

### Example 1: Passing a Function as an Argument

```
var b = function (param1) {  
  console.log(param1); // Logs function itself  
};  
  
b(function () {}); //  Passing an anonymous function
```

### Example 2: Assigning a Named Function

```
var b = function (param1) {  
    console.log(param1);  
};  
  
function xyz() {}  
b(xyz); // ✅ Passing a named function
```

### Example 3: Returning a Function

```
var b = function () {  
    return function () {  
        console.log("Returned function");  
    };  
};  
  
console.log(b()); // ✅ Logs the function itself  
b(); // ✅ "Returned function"
```

✅ High-order functions (HOFs) are built using first-class functions.

---

### 🎯 Summary

- 📌 Function statement = Function declaration (Hoisted)
  - 📌 Function expression = Function stored in a variable (Not Hoisted)
  - 📌 Anonymous function = Function without a name (Used in expressions)
  - 📌 Named function expression = Function expression with a name
  - 📌 First-Class Functions = Functions can be assigned, passed, and returned
-

## Callback Functions in JavaScript ft. Event Listeners

### Callback Functions

A callback function is a function passed as an argument to another function. The receiving function can call it later, enabling asynchronous behavior in JavaScript.

```
setTimeout(function () {  
    console.log("Timer");  
}, 1000); // Callback function executed after 1 sec
```

JavaScript is **synchronous and single-threaded**, meaning it executes code line by line. However, callbacks allow us to perform asynchronous operations, like handling API calls or event listeners.

### Example: Callbacks in Action

```
function x(y) {  
    console.log("x");  
    y(); // Invoking the callback  
}  
  
x(function y() {  
    console.log("y");  
});
```

✓ Output:

x

y

---

## Callbacks & setTimeout

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

✓ Expected Output:

```
x  
y  
(timer appears after 5 seconds)
```

How it works:

- ① `x()` is executed immediately.
- ② `y()` is called inside `x()`, so "x" and "y" are printed.
- ③ The `setTimeout` callback waits 5 seconds before executing.

---

## Blocking the Main Thread

JavaScript has a **single call stack (main thread)**. If a function takes too long, it blocks further execution.



```
function longRunningTask() {  
  for (let i = 0; i < 1e9; i++) {} // Simulating a long task  
}  
console.log("Start");  
longRunningTask();  
console.log("End");
```

✅ **Problem:** If this function takes 30 seconds, JavaScript will be stuck and unresponsive.

🚀 **Solution:** Use async callbacks like `setTimeout` or Promises.

---

## Callback Hell Example

Nested callbacks lead to messy and hard-to-maintain code.

```
function printStr(str, cb) {  
  setTimeout(() => {  
    console.log(str);  
    cb();  
  }, Math.floor(Math.random() * 100) + 1);  
}  
  
function printAll() {  
  printStr("A", () => {  
    printStr("B", () => {  
      printStr("C", () => {});  
    });  
  });  
}  
  
printAll(); // Ensures A → B → C execution
```

✓ **Fix:** Use Promises or Async/Await to avoid deeply nested callbacks.

---

## Event Listeners

Event listeners let us execute callback functions when an event occurs, such as a button click.

### Basic Example

html

Copy

Edit

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

js

Copy

Edit

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

- ◆ When the button is clicked, "Button clicked" is logged.
- 

## Counter Example (With & Without Closures)

### ✗ Using a Global Variable (Bad Practice)

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

### ✓ Issue:

♦ Anyone can modify `count` globally, leading to unpredictable behavior.

### ✓ Using Closures (Better)

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

### ✓ Why is this better?

- `count` is private inside `attachEventList()`.
- Encapsulation: No external access to `count`.

---

## Garbage Collection & removeEventListener

### ♦ Why Remove Event Listeners?

- Event listeners consume memory because they form closures.
- Even after elements are removed, listeners stay in memory, leading to memory leaks.

### ♦ Removing an Event Listener

```
function eventHandler() {  
  console.log("Button clicked");  
}  
  
document.getElementById("clickMe").addEventListener("click", eventHandler);  
  
// Remove event listener when it's no longer needed  
document.getElementById("clickMe").removeEventListener("click", eventHandler);
```

## ✅ Best Practices:

- ✓ Remove listeners when no longer needed
  - ✓ Avoid too many event listeners (e.g., `onScroll`, `onResize`)
- 

## 🎯 Summary

- 📌 Callbacks allow async behavior in JavaScript
- 📌 `setTimeout()` uses a callback to delay execution
- 📌 Callback hell happens with deeply nested callbacks
- 📌 Event Listeners execute a callback when an event occurs
- 📌 Closures help encapsulate data inside event listeners
- 📌 `removeEventListener` prevents memory leaks 🚀

🔥 Mastering callbacks unlocks JavaScript's async power!

---

## Asynchronous JavaScript & Event Loop

### Understanding the JavaScript Execution Model

#### JavaScript Runtime Components

JavaScript runs in a browser environment that consists of:

- 1 **Call Stack** → Executes synchronous code.
  - 2 **Web APIs** → Browser-provided features (e.g., `setTimeout`, `fetch`, `DOM API`).
  - 3 **Callback Queue (Task Queue)** → Holds callbacks from Web APIs (`setTimeout`, `event listeners`).
  - 4 **Microtask Queue** → Holds high-priority tasks like Promises & `MutationObserver`.
  - 5 **Event Loop** → Moves tasks from queues to the Call Stack.
- 

### Web APIs in JavaScript

🚀 JavaScript does NOT have `setTimeout`, `fetch`, or even `console.log()`! These are part of the Web APIs (provided by the browser).

### ✅ Common Web APIs:

- `setTimeout()`, `setInterval()` → Timer functions.
- **DOM APIs** → Manipulate HTML (`document.getElementById()`).
- `fetch()` → Makes network requests.
- `localStorage` → Store data.
- `console.log()` → Outputs to the console.

Example:

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

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

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

### ✅ Output:

```
Start
End
Timer Done (after 5s)
```

### 🚀 Why?

- 1 `console.log("Start")` executes immediately.
  - 2 `setTimeout(cb, 5000)` registers `cb` inside the Web API environment.
  - 3 `console.log("End")` executes next.
  - 4 After 5 seconds, `cb()` moves to the Callback Queue.
  - 5 Event Loop moves `cb()` to the Call Stack only when it's empty.
-

## Event Loop & Callback Queue

### What is the Event Loop?

The Event Loop constantly checks:

- If the Call Stack is empty → It moves a task from the Callback Queue or Microtask Queue to the Call Stack.

Example: Event Loop in Action

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

document.getElementById("btn").addEventListener("click", function cb() {
  console.log("Button Clicked");
});

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

### ✓ Execution Flow:

- ① "Start" is printed.
  - ② Event listener (**cb**) is registered in the Web API environment.
  - ③ "End" is printed.
  - ④ When the button is clicked, **cb()** moves from the Callback Queue → Call Stack → executes.
- ♦ Event Listeners persist in the Web API until removed.

---

## Microtask Queue vs. Callback Queue

### Example: Fetch & Microtask Queue

```

console.log("Start");

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

fetch("https://api.netflix.com").then(function cbF() {
  console.log("CB Netflix");
});

console.log("End");

```

✓ Output (assuming fetch takes 2 sec):

```

Start
End
CB Netflix (from Microtask Queue)
CB Timeout (from Callback Queue)

```

🚀 Why?

- `fetch()` registers `cbF` in the Web API environment.
- `setTimeout()` registers `cbT` (5s timer starts).
- "Start" & "End" are printed immediately.
- After 2s, `cbF` moves to the Microtask Queue.
- After 5s, `cbT` moves to the Callback Queue.
- Event Loop Priority → Microtask Queue runs first!

✓ Microtask Queue Always Runs Before Callback Queue

- ✓ `cbF` executes first (Microtask Queue).
- ✓ `cbT` executes next (Callback Queue).

● Microtask Queue Starvation

If a task keeps adding more tasks inside the Microtask Queue, the Callback Queue may never get a chance to execute (Starvation).

---

## Important Questions & Concepts

### ? When does the Event Loop start?

💡 Always running, checking Call Stack & Queues.

### ? Are all callbacks stored in the Web API environment?

- ◆ No! Only asynchronous callbacks (`setTimeout`, `fetch`).
- ◆ Synchronous callbacks (e.g., `map`, `filter`) execute immediately in the Call Stack.

### ? Does `setTimeout(0)` execute immediately?

🔴 No! Even with `setTimeout(cb, 0)`, `cb` waits in the Callback Queue until the Call Stack is empty.

- ◆ If the Call Stack is busy, `cb` could be delayed beyond 0ms!

Example:

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

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

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

✅ Output:

```
Start
End
Timeout // Still waits until Call Stack is empty
```

---



## 🔥 Summary

- ✓ JavaScript is Single-Threaded (one Call Stack).
- ✓ Web APIs provide async capabilities (e.g., `setTimeout`, `fetch`).
- ✓ Event Loop moves tasks from Callback/Microtask Queues to Call Stack.
- ✓ Microtask Queue (Promises, MutationObserver) has higher priority than Callback Queue.
- ✓ Event Listeners persist in Web API until removed.
- ✓ `setTimeout(0)` does not execute immediately!

🔥 Mastering Asynchronous JavaScript is key to writing efficient web applications! 🚀

---

## V8 Architecture (Google Chrome's JS Engine) 🚀

V8 is Google's JavaScript engine written in C++ and used in Chrome, Node.js, and Deno. It compiles JavaScript into machine code instead of interpreting it line by line, making execution faster.

---

### 💡 Key Components of V8

- 1 **Parser** → Converts JavaScript code into AST (Abstract Syntax Tree).
  - 2 **Ignition (Interpreter)** → Converts AST into Bytecode (Intermediate Code).
  - 3 **TurboFan (Compiler)** → Optimizes & converts bytecode into highly optimized machine code.
  - 4 **Orinoco (Garbage Collector)** → Manages memory efficiently using the Mark-and-Sweep algorithm.
  - 5 **Memory Heap** → Stores objects & variables.
  - 6 **Call Stack** → Manages function execution.
-

## ⚡ How V8 Executes JavaScript

1. **Parsing:** JS code → Tokens → AST (using a Syntax Parser).
  2. **Compilation (JIT - Just-in-Time Compilation):**
    - Ignition → Converts AST to Bytecode (Interpreted first).
    - TurboFan → Optimizes & compiles bytecode into Machine Code.
  3. **Execution:** Machine code runs, and V8 optimizes it further during runtime.
- 

## 🔥 Why V8 is Fast?

- ✓ **JIT Compilation** → Uses both interpretation & compilation.
  - ✓ **Optimized Garbage Collection** → Reduces memory leaks.
  - ✓ **Inline Caching** → Stores function calls for faster execution.
  - ✓ **Hidden Class Optimization** → Efficient object property access.
- 

## 🔧 Where is V8 Used?

- Chrome, Edge (Blink engine uses V8)
- Node.js (Server-side JS runtime)
- Deno (Secure runtime for JS & TS)

🚀 V8 makes JavaScript super fast and efficient!

---

## 🚨 Trust Issues with `setTimeout()` in JavaScript 🚨

`setTimeout()` does not guarantee exact execution time! Even if you set a timer for `5s`, the callback may run after 6, 7, or even 10 seconds.

---

## 🔍 Why Doesn't `setTimeout(5000)` Execute Exactly After 5s?

1. Global Execution Context (GEC) is Created and pushed to the Call Stack.

2. "Start" is printed.

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

3. `setTimeout(cb, 5000)` is encountered:

- The callback function (`cb`) is registered in Web APIs.
- Timer starts counting in the background (separate from JS execution).

4. "End" is printed, and JS moves to the next code execution.

5. While the main thread is still executing (e.g., 1M lines of code), the timer expires.

6. `cb()` moves to the Callback Queue after 5s.

7. Event Loop checks if the Call Stack is empty.

8. If the Call Stack is still busy (e.g., executing 1M lines of code for 10s), `cb()` must wait.

9. Only when the Call Stack is empty, `cb()` is pushed and executed.

 Conclusion: Even though `setTimeout(5000)` was set for 5s, it may execute after 10s if the main thread is busy!

---

## Rule of JavaScript

 Never block the main thread!

JavaScript is single-threaded → If you execute heavy computations, the Call Stack stays blocked, delaying asynchronous callbacks.

---

⚡ What Happens with `setTimeout(cb, 0)`?

```
console.log("Start");
setTimeout(function cb() {
  console.log("Callback");
}, 0);
console.log("End");
```

✓ Output:

```
Start
End
Callback
```

Even though timer is `0s`, `cb()` still enters the Callback Queue and waits for the Call Stack to clear before execution.

✓ Use case: Defer a function without blocking execution (e.g., to let high-priority tasks finish first).

---

## 🚀 Key Takeaways

- `setTimeout()` guarantees a minimum delay, NOT an exact delay.
- If the Call Stack is busy, `setTimeout()` waits longer.
- `setTimeout(0)` still goes through the Callback Queue and executes after synchronous code.
- JavaScript is single-threaded → Avoid blocking the main thread for smooth performance.

This is the Concurrency Model of JavaScript 🔥.

## 🚀 Higher-Order Functions (HOF) in JavaScript

### ♦ What is a Higher-Order Function?

A Higher-Order Function is a function that:

- ✓ Takes another function as an argument OR
- ✓ Returns a function as its result.

```
function x() {  
  console.log("Hi");  
}  
  
function y(callback) { // HOF (takes function as an argument)  
  callback();  
}  
  
y(x); // Output: Hi
```

👉 Here, **y** is a Higher-Order Function and **x** is a callback function.

---

### ♦ Real-World Example: Optimizing Repetitive Code

### ✗ First Approach: Naive Implementation

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

### 👉 Problem?

If we now need circumference, we must write another function → violates DRY (Don't Repeat Yourself) Principle.

---

## ✓ Better Approach: Using Higher-Order Functions

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

// Logic to calculate area
const area = radius => Math.PI * radius * radius;

// Logic to calculate circumference
const circumference = radius => 2 * Math.PI * radius;

// Higher-Order Function to apply any calculation
const calculate = function(radiusArr, operation) {
  return radiusArr.map(operation);
}

console.log(calculate(radiusArr, area));           // Output: [3.14, 12.56, 28.27, 50.26]
console.log(calculate(radiusArr, circumference)); // Output: [6.28, 12.56, 18.85, 25.13]
```

💡 Why is this better?

- ✓ No code repetition
- ✓ Reusable functions
- ✓ Scalable for new operations

---

### ♦ Polyfill of map()

The `calculate` function is actually a custom implementation of `.map()`.

👉 Let's create a custom `map()` method:

```

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

console.log(radiusArr.calculate(area));           // Same as `radiusArr.map(area)`
console.log(radiusArr.calculate(circumference)); // Same as `radiusArr.map(circumference)`

```

- ✓ Now, `.calculate()` works like `map()`
- ✓ Reusable for any transformation

## Key Takeaways

- ✓ Higher-Order Functions make code clean, reusable, and modular.
- ✓ `map()`, `filter()`, and `reduce()` are built-in HOFs in JS.
- ✓ Writing a custom `map()` function improves understanding of functional programming.

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

### ♦ `map()` - Transforming an Array

- ✓ `map()` creates a new array by applying a function to each element.
- ✓ It does not modify the original array.

### Example 1: Double the Array Elements

```

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

const doubleArr = arr.map(x => x * 2);
console.log(doubleArr); // [10, 2, 6, 4, 12]

```

## Example 2: Convert Elements to Binary

```
const binaryArr = arr.map(x => x.toString(2));  
console.log(binaryArr); // ["101", "1", "11", "10", "110"]
```

---

### ♦ `filter()` - Selecting Elements from an Array

✓ `filter()` creates a new array containing only elements that satisfy a condition.

✓ Returns elements where the callback function evaluates to true.

## Example 1: Filter Odd Numbers

```
const oddArr = arr.filter(x => x % 2 !== 0);  
console.log(oddArr); // [5, 1, 3]
```

## Example 2: Filter Even Numbers

```
const evenArr = arr.filter(x => x % 2 === 0);  
console.log(evenArr); // [2, 6]
```

---

### ♦ `reduce()` - Condensing an Array into a Single Value

✓ `reduce()` applies a callback function to reduce an array into a single value.

✓ It takes two arguments:

- **Accumulator (acc)** → Stores the result
- **Current Element (curr)** → Current element being processed

## Example 1: Sum of Array Elements

```
const sum = arr.reduce((acc, curr) => acc + curr, 0);  
console.log(sum); // 17
```



## Example 2: Find Maximum Value

```
const max = arr.reduce((max, curr) => (curr > max ? curr : max), 0);
console.log(max); // 6
```

---

### ◆ Real-World Use Cases

## Example 1: Get Full Names using `map()`

```
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 },
];

const fullNameArr = users.map(user => `${user.firstName} ${user.lastName}`);
console.log(fullNameArr);
// ["Alok Raj", "Ashish Kumar", "Ankit Roy", "Pranav Mukherjee"]
```

## Example 2: Count Users by Age using `reduce()`

```
const report = users.reduce((acc, curr) => {
  acc[curr.age] = (acc[curr.age] || 0) + 1;
  return acc;
}, {});

console.log(report);
// { 23: 1, 29: 2, 50: 1 }
```

---

### ◆ Function Chaining (Combining `map()`, `filter()`, and `reduce()`)

## ✓ Example: First Names of Users Under 30

```
const output = users
  .filter(user => user.age < 30)
  .map(user => user.firstName);

console.log(output); // ["Alok", "Ashish", "Ankit"]
```

### ✓ Same Example Using reduce()

```
const outputReduce = users.reduce((acc, curr) => {  
  if (curr.age < 30) acc.push(curr.firstName);  
  return acc;  
}, []);  
  
console.log(outputReduce); // ["Alok", "Ashish", "Ankit"]
```

---

### Key Takeaways

- ✓ **map()** → Used for transforming an array (returns new array).
  - ✓ **filter()** → Used for selecting elements that satisfy a condition.
  - ✓ **reduce()** → Used for aggregating data into a single value.
  - ✓ **Function chaining** → Combining **map()**, **filter()**, and **reduce()** makes code cleaner and more efficient.
- 

### Callbacks in JavaScript

#### ♦ What is a Callback?

A callback is a function that is passed as an argument to another function and is executed later.

- ✓ Callbacks are essential in handling asynchronous operations (e.g., API calls, database queries, file handling).
  - ✓ **Problem:** Callbacks can lead to Callback Hell and Inversion of Control.
- 

#### ♦ Synchronous JavaScript Example

```
console.log("Namaste");  
console.log("JavaScript");  
console.log("Season 2");
```

```
// Output:  
// Namaste  
// JavaScript  
// Season 2
```

✓ JavaScript executes code line-by-line (single-threaded, synchronous).

---

### ♦ Using Callbacks for Asynchronous Operations

```
console.log("Namaste");  
setTimeout(() => {  
  console.log("JavaScript");  
}, 5000);  
console.log("Season 2");  
  
// Output:  
// Namaste  
// Season 2  
// (after 5 sec) JavaScript
```

✓ `setTimeout()` delays execution without blocking the rest of the code.

---

### ♦ Callbacks in an e-Commerce Web App

Scenario: User places an order → Proceeds to payment → Shows order summary → Updates wallet.

#### ✓ Step 1: Without Callbacks (Synchronous)

```
const cart = ["shoes", "pants", "kurta"];  
  
api.createOrder();  
api.proceedToPayment();
```

🚨 Problem: `proceedToPayment()` executes before `createOrder()` is completed.

---

## ✅ Step 2: Using Callbacks

```
api.createOrder(cart, function () {  
  api.proceedToPayment();  
});
```

✓ Now, `proceedToPayment()` is executed only after `createOrder()` completes.

---

## ✅ Step 3: Handling Multiple Dependencies

```
api.createOrder(cart, function () {  
  api.proceedToPayment(function () {  
    api.showOrderSummary(function () {  
      api.updateWallet();  
    });  
  });  
});
```

💣 Problem: Callback Hell (aka Pyramid of Doom)

● Code is hard to read, debug, and maintain.

---

## ♦ Inversion of Control

💣 Risk: We pass `proceedToPayment()` to `createOrder()`, trusting it to execute properly.

● What if:

- `createOrder()` never calls the callback?
- `createOrder()` calls it twice?
- Another developer modifies `createOrder()` and breaks functionality?

✓ We lose control over execution!

---

## ♦ Key Takeaways

✓ Callbacks enable asynchronous programming.

✓ Issues with Callbacks:

- Callback Hell → Code becomes nested and unreadable.
  - Inversion of Control → We trust external functions to call our callback correctly.
    - Next Step: Promises & Async/Await
  - Fixes Callback Hell
  - Gives back control over execution
- 

## 🚀 JavaScript Promises Explained

### ♦ What is a Promise?

A Promise is an object that represents the eventual completion (or failure) of an asynchronous operation.

### ✓ Before Promises (Using Callbacks)

🔥 Problems:

- Inversion of Control: We rely on external functions to execute callbacks.
- Callback Hell: Code becomes nested, unreadable, and hard to maintain.

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

● This is the "Pyramid of Doom".

---

### ◆ How Promises Solve This?

#### ✓ Solution 1: Avoid Inversion of Control

- ✓ Instead of passing a callback, we attach a function using `.then()`.
- ✓ Promise guarantees that the attached function will be called only once when the data is ready.

```
const promiseRef = createOrder(cart); // Returns a Promise

promiseRef.then(function (orderId) {
  proceedToPayment(orderId);
});
```

---

### ◆ How Promises Work Internally?

- ① When `createOrder(cart)` is called, it returns a Promise.
  - ② Initially, the Promise state is `"pending"` and its value is `undefined`.
  - ③ After execution completes, the Promise is fulfilled with `orderId`.
  - ④ The `.then()` callback is triggered only when the Promise is resolved.
-

### ♦ Real Example: Fetching Data from API

```
const URL = "https://api.github.com/users/alok722";  
  
const user = fetch(URL); // Returns a Promise  
console.log(user); // Promise {<Pending>}  
  
// Attaching a callback to handle response when Promise is resolved  
user.then(function (response) {  
  console.log(response);  
});
```

- ✓ `fetch()` makes an API call and immediately returns a Promise in "Pending" state.
  - ✓ When the request completes, the Promise state changes to "Fulfilled" with the response data.
- 

### ♦ Promise States

A Promise can be in one of three states:

- 1 Pending → Initial state, operation is not complete yet.
- 2 Fulfilled → Operation completed successfully.
- 3 Rejected → Operation failed.

- ✓ Promises are immutable: Once a Promise is fulfilled or rejected, it cannot change.
- 

### ♦ Fixing Callback Hell with Promise Chaining

- 🚀 Promise chaining ensures a clean, readable structure.

```
createOrder(cart)  
  .then(orderId => proceedToPayment(orderId))  
  .then(paymentInfo => showOrderSummary(paymentInfo))  
  .then(balance => updateWalletBalance(balance));
```

- ✓ Each `.then()` returns a Promise, making the result available to the next `.then()`.
- ✓ Common Mistake: Forgetting to return a Promise in chaining.

### ✓ Fixed Version

```
createOrder(cart)
  .then(orderId => {
    return proceedToPayment(orderId);
  })
  .then(paymentInfo => {
    return showOrderSummary(paymentInfo);
  })
  .then(balance => {
    return updateWalletBalance(balance);
  });
```

- ✓ Better Readability with Arrow Functions
- 

### ♦ Key Takeaways

- ✓ Promises solve Inversion of Control and Callback Hell.
- ✓ A Promise is a placeholder for a future value.
- ✓ Promise chaining improves readability and maintainability.
- ✓ Always return a Promise inside `.then()` to maintain the chain.
- ✓ Promises can only resolve or reject once.

🚀 Next Step: Learn `async/await` for even cleaner async handling!