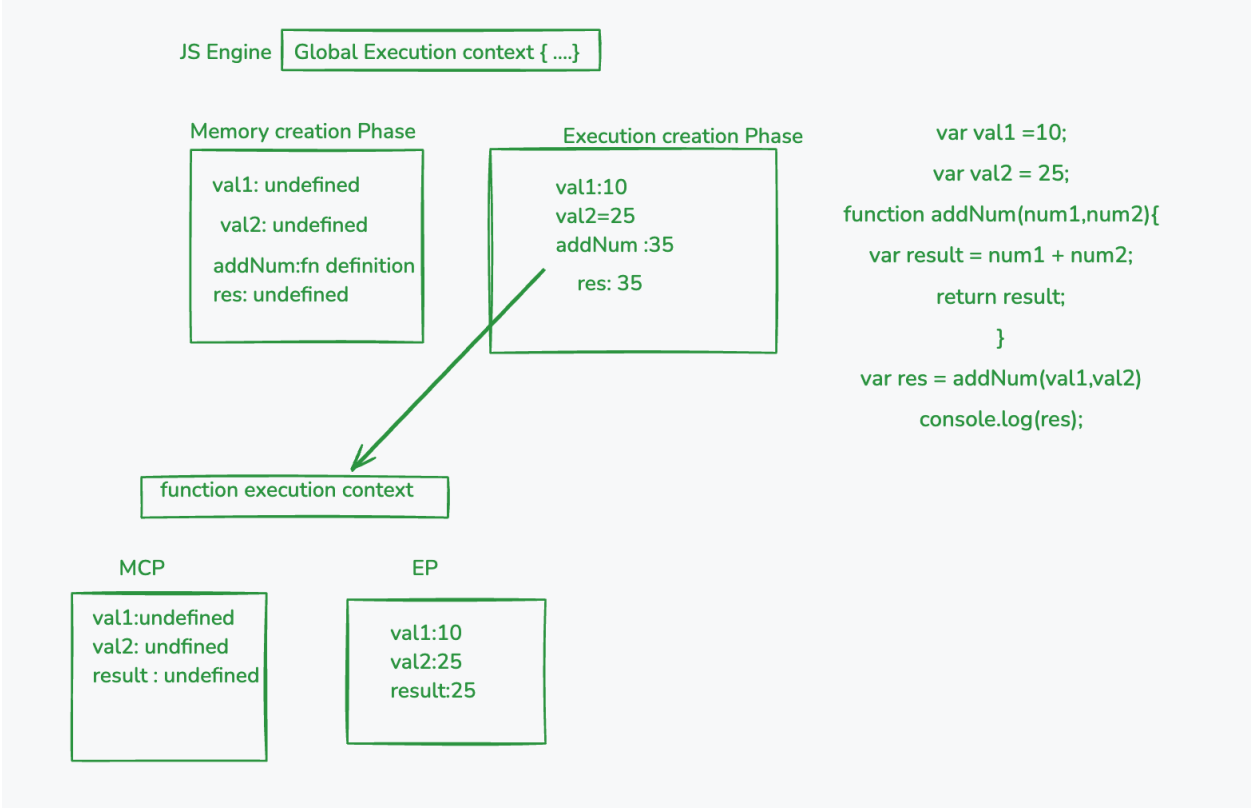
**Lecture 1: JavaScript Code Execution & Global Execution Context**

1. **Global Execution Context (GEC)**: Created first when the code is run.
2. **Phases of Execution**:
    - **Memory Creation Phase**: Allocates memory for variables and functions. Variables are assigned an undefined value, and entire functions are stored.
    - **Execution Phase**: Executes the code line by line and assigns actual values to variables. For functions, a new Function Execution Context (FEC) is created.
3. **Function Execution Context (FEC)**: Similar phases occur within each function call (memory allocation and execution).
4. **Execution Flow**: The code executes sequentially, with the GEC at the bottom of the call stack and FECs pushed and popped as functions are invoked and returned.



**Phases of Execution in simple words**

**Memory Creation Phase:** Allocates memory for variables and functions. Variables are assigned an undefined value, and entire functions are stored.

**Execution Phase:** Executes the code line by line and assigns actual values to variables. For functions, a new Function Execution Context (FEC) is created.

**Function Execution Context (FEC):** Similar phases occur within each function call (memory allocation and execution).

**Execution Flow:** The code executes sequentially, with the GEC at the bottom of the call stack and FECs pushed and popped as functions are invoked and returned.

**Lecture 2: Event Loop, Call Stack & Concurrency in JavaScript**

1. **Call Stack**: Operates in LIFO (Last In, First Out) order, managing function execution. Functions are pushed when invoked and popped when they return.

2. **Event Loop**: Manages asynchronous operations by continuously checking the call stack and the task queues (macro and microtask queues).

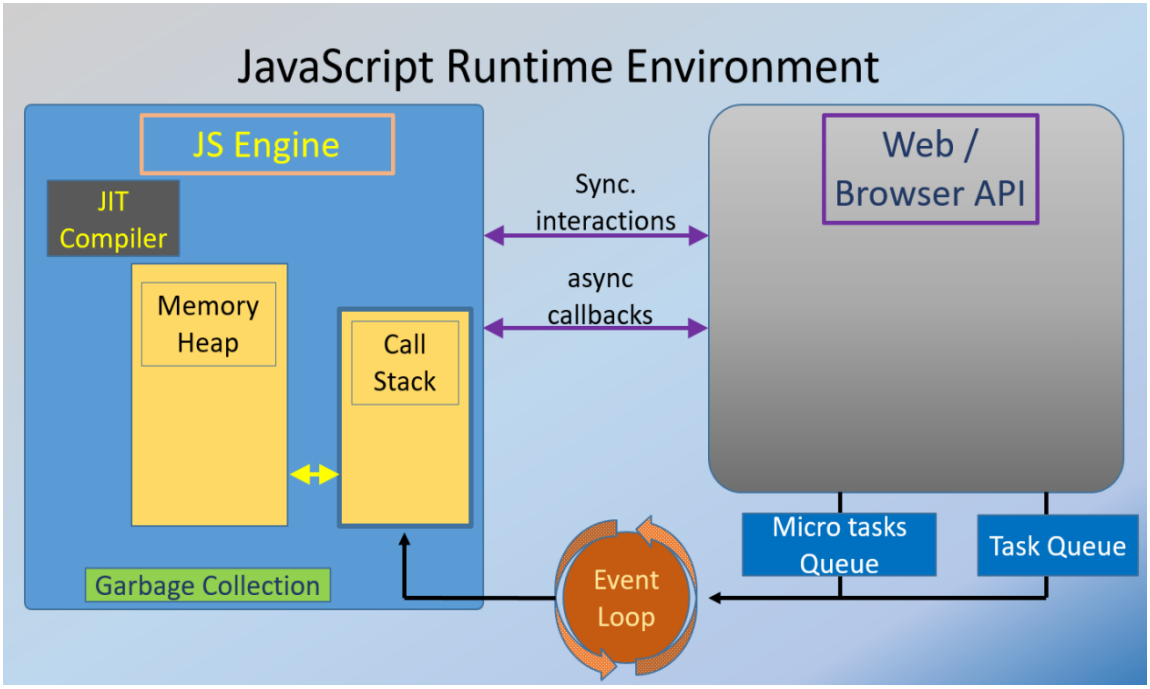
3. **Concurrency**: Enhances user experience by allowing non-blocking operations. It appears that multiple tasks run simultaneously, even though they are managed by the event loop.

4. **Queues**:

    - **Task Queue**: Holds callbacks from setTimeout, setInterval, etc.

    - **Microtask Queue**: Holds callbacks from promises and mutation observers, and it has higher priority than the task queue

5. **JIT Compiler & Garbage Collector**: The Just-In-Time compiler optimizes the code execution, and the garbage collector frees up memory by removing unused variables.



**Lecture 3: Hoisting Explained: Variables & Functions**

**Definition:** Hoisting is JavaScript's default behavior of moving declarations to the top of the current scope (script or function) before code execution.

When we say the declaration moves to the top, it doesn't become visible at the top of the code. it just means it gets stored in memory beforehand.

1. **Var Hoisting:** Variables declared with var are hoisted and initialised to undefined. They can be accessed before their declaration, but will return undefined until assigned.

2. **let and const Hoisting:** These are hoisted too, but they stay in the Temporal Dead Zone (TDZ) until their declaration is reached. Accessing them beforehand results in a ReferenceError.

3. **Function Hoisting** Function declarations are completely hoisted, meaning you can call the function even before its actual declaration in the code.

4. **Function Expression Hoisting** In function expressions (e.g., assigning a function to a variable), the variable is hoisted but not the function itself. If declared with var, it's hoisted and initialised as undefined, so calling it before assignment causes an error.

**Lecture 4: Var vs Let vs Const: Best Practices**

**var**:

- Function-scoped, meaning it's only scoped to the function where it's defined.
- Can be re-declared and updated within the same scope.
- Gets hoisted and initialized with undefined during the memory creation phase.

**let**:

- Block-scoped, meaning it's confined to the block (like loops or if statements) where it's defined.
- Can be updated but not re-declared in the same scope.
- Gets hoisted but not initialized, leading to a "temporal dead zone" until it's defined.

**const**:

- Block-scoped like let.
- Must be initialized at the time of declaration and cannot be re-assigned.
- Also gets hoisted but remains in the "temporal dead zone" until it's defined.

**Lecture 5: Arrow Functions vs Traditional Functions: A Deep Dive into the 'this' Keyword**

Arrow Function vs Normal Function:

- **Normal Functions:** Have their own this context, which depends on how they are called.
- **Arrow Functions:** Do not have their own this; they inherit this from their surrounding (lexical) scope.

**this Behavior in Objects:**

- **Normal Functions in Objects:** this refers to the object itself, so object properties are accessible.
- **Arrow Functions in Objects:** this does not refer to the object, but to the outer scope (e.g., global scope), which can lead to unexpected results.

**Global Objects:**

- **Browser Environment:**
  - Global object is window.
  - Variables declared with var in the global scope become properties of window.
  - Variables declared with let or const do not attach to the window object.
- **Node.js Environment:**
  - Global object is global.
  - Variables declared with var, let, or const in the global scope do NOT become properties of the global object

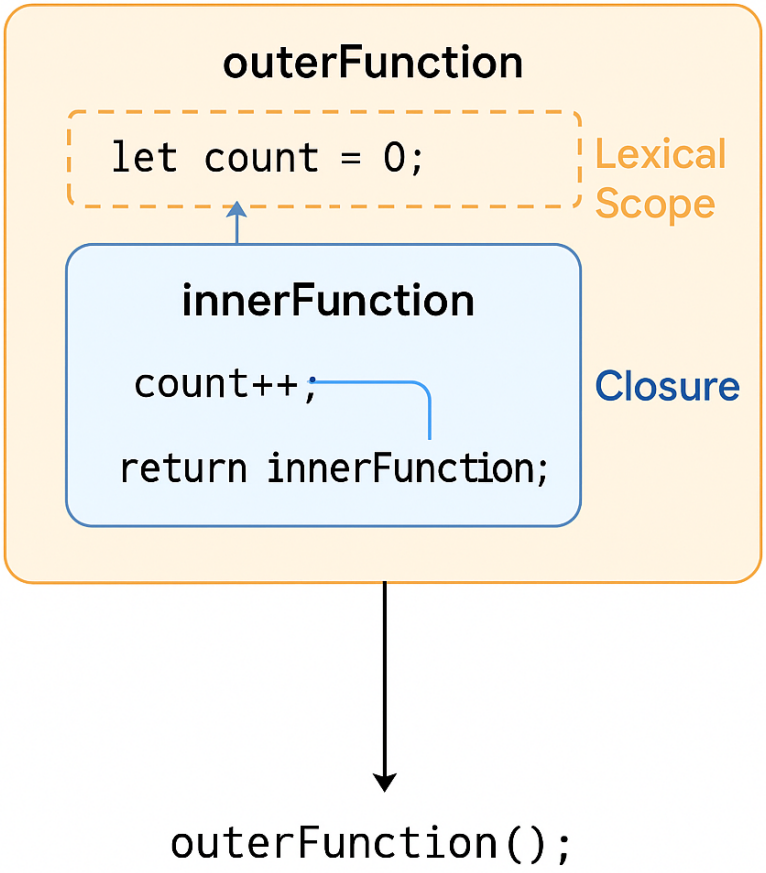
**Lecture 6: Closure & Lexical Scope in JavaScript**

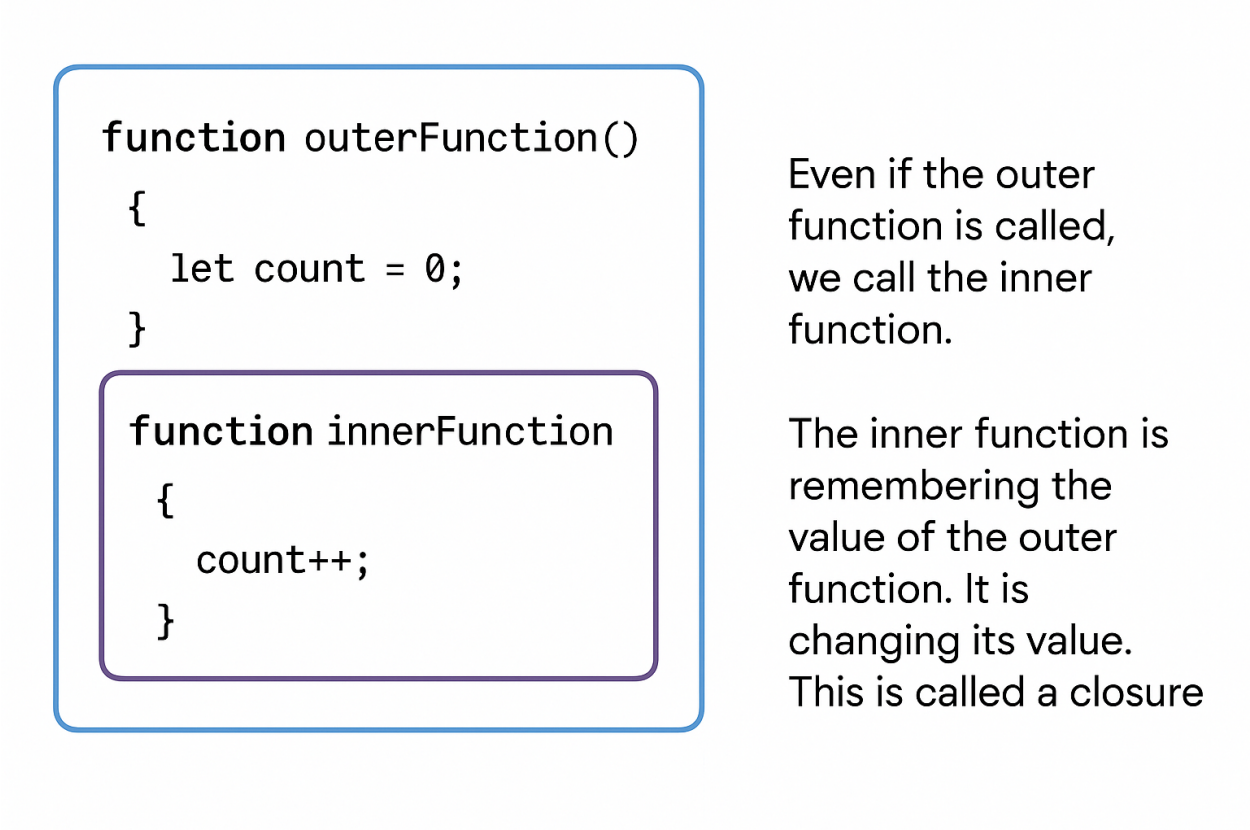
**Lexical Scope:**

- Definition: Inner functions can access variables from their outer function's scope.
- Simplified: An inner function remembers and can access the variables from where it was originally defined.

**Closure:**

- Definition: A function that retains access to its outer function's variables even after the outer function has completed execution.
- Simplified: Even after the outer function is done, the inner function still has access to those outer variables.




function outerFunction()
{
let count = 0;
}
function innerFunction
{
count++;
}
Even if the outer function is called, we call the inner function.
The inner function is remembering the value of the outer function. It is changing its value. This is called a closure

**Lecture 7: Classes, Objects, and Getters/Setters in JavaScript**

1. A **class** is a template used to create objects.

2. An **object** is created from a class and holds real data and functions.

3. In JavaScript, all properties are **public by default**.

4. You can **directly access or change** a property (e.g., person.name = "Ajay").

**5. Getters** are used to safely **read** a property.

**6. Setters** are used to safely **update** a property.

7. The _ (underscore) in property names (like _name) is just a **naming rule** to show it's internal or private.

8. You can still **access _name directly**, but it's not recommended.

**9. Getters and setters** help you add **validation or control** before reading or changing values.

10. In small or simple scripts, you can skip getters/setters if no extra logic is needed.

11. **#name** is a private field — it can't be accessed from outside the class.

```
class A {
  #name; // private field

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

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

const obj = new A();
console.log(obj.getName());  // ✅ Ajay
console.log(obj.#name);      // ❌ Error
```

**Lecture 8: Working with Objects in JavaScript**

Object: A collection of key-value pairs, where keys are strings (or symbols) and values can be any data type.

Creating Objects:

- Object Literal: { key: value }
- Constructor: new Object()

Accessing Properties:

- Dot Notation: object.key
- Bracket Notation: object["key"]

Checking Property Existence:

- in Operator: Checks if a property exists in the object (including inherited properties).
- hasOwnProperty() Method: Checks if a property exists directly on the object (not inherited).

**In Simple words**

1. Object Creation – Objects in JavaScript are created using key-value pairs.
2. Everything Is an Object – In JavaScript, almost everything (arrays, functions, etc.) is derived from the base Object.
3. Object Methods – Use Object.keys(), Object.values(), and Object.entries() to get keys, values, and key-value pairs respectively.

4. Looping Through Objects –
   Use for...in to loop through keys.
   Use for...of with Object.entries() to loop through both keys and values.

**Please Note : -** Primitive Types (like number, string, Boolean, null, undefined, symbol, bigint) are not objects.

**Lecture 9: Optional Chaining (?.), Nullish Coalescing (??), and Logical OR (||)**

1. Optional Chaining (?.)

- Safely accesses nested object properties.
- Prevents runtime errors when something is undefined or null.
- If the property doesn't exist, it simply returns undefined.
- Commonly used when working with dynamic or API data.

2. Nullish Coalescing (??)

- Provides a fallback value only if the original value is null or undefined.
- Does not trigger for values like 0, false, or empty string.
- Ideal when 0 or "" are valid and should not be replaced.

---

3. Logical OR (||)

- Provides fallback for any falsy value.
- This includes false, 0, "", null, and undefined.
- Can unintentionally replace valid values like 0 or false.

**Lecture 10: Destructuring in JavaScript (Object & Array)**

**1. What is Destructuring?**

- A shorthand syntax to extract values from objects and arrays.
- Makes your code cleaner and more readable.
- Commonly used in function parameters, API data, and React props.

---

**Object Destructuring**

- Allows you to extract values from objects using their property names.
- Variable names must match the object's keys.
- You can rename variables while destructuring.
- You can set default values if a key is missing.
- Often used when working with API responses or config objects.

---

**Array Destructuring**

- Let's you extract values based on their position in the array.
- Useful when you need the first few elements quickly.
- You can skip values using commas.
- Very popular with functions that return arrays (e.g. React useState).
- Also used for swapping variables without a temporary value.

**Lecture 11: Shallow & Deep Copy in JavaScript**

**Shallow Copy:**

- Definition: Creates a new object whose top-level properties are copied, but nested objects are still references to the original.
- Mutability: Changes to nested objects affect both the original and the copied object.
- Common Methods: Using the spread operator (...) or Object.assign() for objects.

**Deep Copy:**

- Definition: Creates a new object with entirely independent copies of all nested objects and properties.
- Mutability: Changes to the copied object do not affect the original, and vice versa.
- Common Methods: Using JSON.parse(JSON.stringify(obj)) or libraries like Lodash's cloneDeep().

**In simple words, with respect to JS & React**

**Shallow Copy :** When you create a shallow copy of an object, the top-level properties get a new reference. So, if you change a top-level property (like changing the name from "Ajay" to "Vijay"), that change will not affect the original object. However, if you change a nested property (like changing the city inside an address object), the original object will reflect that change because the nested properties share the same reference.

**Deep Copy:** When you create a deep copy of an object, every level of the object gets its own new reference. This means that any change you make, whether it's a top-level property or a nested property, will not affect the original object at all. Each part of the copied object is completely independent from the original.

**Shallow Copy in React:** When you create a shallow copy of a state object and update a top-level property, React detects that the reference has changed. This triggers a re-render, and the UI updates accordingly. However, if you change a nested property in a shallow copy, React might not detect that change automatically since the nested reference remains the same.
**Deep Copy in React:** When you make a deep copy of a state object, every part of the object has a new reference. This means any change, whether it's top-level or nested, will be detected by React, ensuring that the UI updates as expected.

**Lecture 12 : Real-World Power of JS Arrays: map, filter, join, split, reduce, find & find Index**

**map()**: Transforms each element in an array and returns a new array.

**filter()**: Creates a new array with all elements that pass a test.

**join()**: Joins all elements of an array into a single string, separated by a specified separator.

**split()**: Splits a string into an array of substrings using a specified delimiter.

**reduce()**: Reduces the array to a single value by executing a reducer function on each element.

**find()**: Returns the first element that satisfies a given condition.

**findIndex()**: Returns the index of the first element that satisfies a given condition.

**Lecture 13: Mastering call, apply, and bind in JavaScript**

**call() – Key Points:**

- Executes the function **immediately**.
- Accepts thisArg and **arguments as comma-separated values**.
- Used for **function borrowing** and changing this context.
- Syntax: func.call(thisArg, arg1, arg2, ...)

---

**apply() – Key Points:**

- Executes the function **immediately**.
- Accepts thisArg and **arguments as an array**.
- Useful when arguments are already in array format.
- Syntax: func.apply(thisArg, [arg1, arg2, ...])

---

**bind() – Key Points:**

- **Does NOT execute immediately**.
- Returns a **new function** with bound this and optional preset arguments.
- Great for **delayed execution**, like in event listeners or callbacks.
- Syntax: const fn = func.bind(thisArg, arg1, arg2, ...)

**Lecture 14: JavaScript Array Methods – slice vs splice**

### slice()

- Returns a shallow copy of a portion of the array.
- Takes two arguments: start (inclusive), end (exclusive).
- Does NOT modify the original array (non-destructive).
- Supports negative indices (counts from end).
- Commonly used for viewing or extracting data safely.
- Syntax: array.slice(start, end)

---

### splice()

- Modifies the original array by adding/removing/replacing elements.
- Takes arguments: startIndex, deleteCount, and optional ...itemsToAdd.
- Returns an array of deleted items.
- Can be used to insert elements at any position.
- Can be used to replace or delete elements.

**Lecture 15: Mastering Spread & Rest Operators in JavaScript**

Spread Operator (...)

- Used to unpack elements from arrays or properties from objects.
- Mostly used in function calls, array/object cloning, and merging.
- It works outside of function parameters.
- Does not mutate original data (when cloning/merging).

**Examples:**

- [...arr] – clones an array
- {...obj} – clones an object
- func(...arr) – spreads array values as individual arguments

---

Rest Operator (...)

- Used to gather multiple arguments or values into a single array or object.
- Works in function parameters and destructuring.
- Allows handling of variable number of arguments.
- Always returns an array (or object in case of object destructuring).

**Simple words**

Spread Operator (...)

- Used to unpack elements from arrays or properties from objects.
- Mostly used in function calls, array/object cloning, and merging.
- Does not mutate original data (when cloning/merging).

Examples:

- [...arr] – clones an array

- {...obj} – clones an object
- func(...arr) – spreads array values as individual arguments

Rest Operator (...)

- Used to gather multiple arguments or values into a single array or object.
- Works in function parameters and destructuring.
- Allows handling of variable number of arguments.

## Lecture 16: Ternary Operator & Short-Circuiting

### Ternary Operator (? :)

- A concise alternative to if...else statements.
- Syntax: condition ? valueIfTrue : valueIfFalse
- Returns one of the two expressions based on the condition.
- Useful for inline decisions and UI logic.
- Keeps code clean and readable when handling simple conditions.

### Short-Circuiting ( || and && )

**(OR Operator ||)**

- Returns the first truthy value.
- Commonly used to assign default values.
- Skips evaluation of right-hand side if left is truthy.

**(AND Operator &&)**

- Returns the first falsy value (or last if all are truthy).
- Used for conditional execution.
- Skips evaluation of right-hand side if left is falsy.

**Lecture 17: Boolean Values in JS**

- JavaScript automatically converts values to Boolean in conditional statements (like if, while, etc).
- Values are either treated as truthy or falsy.

**Falsy Values (Evaluated as false in Boolean context):**

- 0
- "" (empty string)
- null
- undefined
- NaN
- false (of course)

**Truthy Values (Evaluated as true in Boolean context):**

- Any non-zero number → e.g. 1, 42, -7
- Any non-empty string → "hello", "false"
- Empty array []
- Empty object {}

Converting to Boolean Explicitly:

Use the Boolean() function to convert any value to true/false.

**Lecture 18: JS Prototype & Inheritance**

1. What is a Prototype?

- JavaScript objects have an internal link to another object called their prototype.
- If a property or method is not found on the object itself, JavaScript looks up the prototype chain.

Prototype is like a fallback object for methods/properties.

Inheritance allows one object to use the methods of another.

**You can do inheritance using:**

- **Object.create()**
- **Function constructors**
- **class and extends (modern syntax)**

**Prototypes and __proto__ Property:** Every JavaScript object has a hidden __proto__ property that points to its prototype. This prototype object can have its own __proto__, forming a prototype chain.

**Traditional Inheritance Using Constructor Functions:** Before ES6, inheritance was achieved using constructor functions. A constructor function is a regular function used with the new keyword. Methods shared by all instances are attached to the constructor's prototype.

**Lecture 19: Understanding Set & Map in JavaScript**

**What is a Set?**

- A Set is a special type of collection that stores only **unique values**.
- It automatically removes duplicates.
- The order of insertion is maintained.
- Useful when you need a list of distinct items.

**Common Use-Cases for Set**

- Removing duplicate values from an array.
- Tracking unique items (like selected tags, IDs, etc.).
- Efficient membership check (has() is fast).

**Key Features of Set**

- No duplicate values allowed.
- Fast lookups and deletions.
- You can iterate over a Set easily using loops.

**What is a Map?**

- A Map is a collection of **key-value pairs**.
- Unlike regular objects, **Map keys can be of any type**, including objects, functions, etc.
- Maintains insertion order.

**Common Use-Cases for Map**

- Storing and retrieving data using keys.

- Caching results.
- Mapping IDs to data or configuration settings.

**Key Features of Map**

- Keys can be strings, numbers, or objects.
- Maintains the exact order of insertion.
- More powerful than plain objects for key-value storage.

**Lecture 20: Event Delegation for Performance Optimization & Event Bubbling**

Event Bubbling in JavaScript

- Events in JavaScript propagate from child to parent (bottom to top).
- When an event is triggered on a child element, it bubbles up through its ancestors unless stopped.
- This allows parent elements to listen for events fired on their descendant elements.
- Example: Clicking a button inside a  can also trigger a click event on the .

**Event Delegation**

- **Technique** where a parent element handles events for its **child elements**.
- Useful when:
    - You have **many child elements**.
    - Elements are **dynamically added or removed**.
- Instead of attaching event listeners to each child, attach **one listener to the parent**.
- Use event.target to determine which child triggered the event.
- Improves **performance** and **maintainability**.

```
document.getElementById("parent").addEventListener("click", function(event) {
  if (event.target.matches(".child")) {
    console.log("Child clicked:", event.target.textContent);
  }
});
```

**Lecture 21: JavaScript Promises: The Foundation of Async Code**

What is a Promise in JavaScript?

- A Promise is an object representing the eventual completion (or failure) of an asynchronous operation.
- Think of it as a placeholder for a value that is not available yet but will be in the future.

**States of a Promise**

1. Pending – Initial state, neither fulfilled nor rejected.
2. Fulfilled – Operation completed successfully.
3. Rejected – Operation failed with an error.

**Syntax:-**

```
promise
  .then((data) => {
    // success block
  })
  .catch((error) => {
    // error block
  })
  .finally(() => {
    // always runs
  });
```

**Lecture 22: Async & Await in JavaScript**

**What is async/await?**

- Async/await is **syntactic sugar over Promises**.
- Makes asynchronous code look and behave like **synchronous code**.
- Helps write **cleaner, readable, and easier-to-debug** async code.

**Key Concepts**

- A function declared with async always **returns a Promise**.
- Inside an async function, you can use await before a Promise.
- await makes JavaScript **wait** until the Promise resolves or rejects.

```
async function fetchData() {
  try {
    const response = await fetch(url);
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error("Error:", error);
  }
}
```

**Behind the Scenes**

- **await pauses execution of the async function.**
- **Once the Promise resolves, execution resumes.**
- **Errors are caught using try...catch.**

**Promise.all:** This method allows you to handle multiple promises at once. It returns an array of results when all the promises are resolved. If any promise is rejected, then Promise.all immediately rejects with that error.

## Lecture 23: Memoization in JavaScript

What is Memoization?

- Memoization is a performance optimization technique.
- It caches the result of a function call based on its input.
- If the function is called again with the same input, the cached result is returned instead of recomputing it.

### Why Use Memoization?

- Saves time on repetitive and expensive function calls.
- Especially useful in:
    - Recursive functions (e.g., Fibonacci)
    - Heavy computations
    - React components (e.g., useMemo, React.memo)

**Use Case:**

Memoization is particularly useful for functions with heavy computations or recursive functions like those used in calculating Fibonacci numbers.

**Lecture 24: ES6 Modules in JavaScript**

What are ES6 Modules?

- ES6 Modules allow code to be divided across multiple files.
- They enable better code organization, reusability, and maintainability.
- Modules use import and export statements

Why Use Modules?

- Keeps code modular and readable.
- Avoids polluting the global scope.
- Allows easy sharing of code between files and teams.
- Works seamlessly with modern tools (React, Vite, Webpack).

**Lecture 25: Understanding export default vs Named Exports**

Named Exports

- Used to export multiple values from a file.
- Each exported item has a name, and must be imported using that same name.
- You can export variables, functions, or constants.
- Multiple named exports are allowed per file.

Default Export

- Used to export a single main value from a file.
- Can be a function, class, object, or primitive.
- You can give any name when importing.
- Only one default export is allowed per file.

Use named exports when your file exports multiple helpers. Use default export when your file exports one main thing, like a React component, a config, or an entry point.

**Lecture 26: Debouncing & Throttling in JavaScript**

**Debouncing:**
Definition: Debouncing ensures that a function is only executed after a certain period of inactivity. It's useful for events that might fire frequently, such as keystroke events during typing or window resizing.

How It Works: When the event is triggered, a timer is set. If the event is triggered again before the timer ends, the timer resets. The function only executes after the timer completes without interruption.

**Throttling:**
Definition: Throttling ensures that a function is only executed at most once in a specified time interval. It's useful for controlling the rate of execution for events like scrolling or resizing.

How It Works: When the event is triggered, the function executes immediately and then ignores any subsequent triggers until the specified time interval has passed.

**Use Cases:**
Debouncing is commonly used for search input fields, where you only want to fetch results after the user has stopped typing.

Throttling is often used for events like window resizing or infinite scrolling, where you want to limit how often a function runs.

**********Thank You**********