**Closures**

Closures are one of the most powerful and elegant features in JavaScript. They allow functions to "remember" the environment in which they were created—even after that environment is gone.

🔐 What Is a Closure?

A **closure** is a function that retains access to its **outer scope**, even after the outer function has finished executing. This is possible because of **lexical scoping**, where scope is determined by where variables and functions are declared in the source code.

In simple terms: **a closure lets an inner function access variables from an outer function even after the outer function has returned**.

🧪 Example of a Closure

function outer() { let outerVar = "I'm in the outer scope!"; function inner() { console.log(outerVar); // Accesses outerVar even after outer() is done } return inner; } const closure = outer(); // outer() runs and returns inner() closure(); // Logs: "I'm in the outer scope!"

Even though outer() has finished executing, inner() still remembers outerVar. That’s a closure in action.

🔁 Closures for Private Variables

Closures are often used to create **private variables**:

function counter() { let count = 0; return function () { count++; return count; }; } const increment = counter(); console.log(increment()); // 1 console.log(increment()); // 2 console.log(increment()); // 3

Here, count is not accessible from outside counter(), but increment() can still modify and access it.

⚙️ Closures in Asynchronous Code

Closures are especially useful in asynchronous programming:

function createTimers() { for (let i = 1; i <= 3; i++) { setTimeout(function () { console.log(`Timer ${i}`); }, i \* 1000); } } createTimers(); // Output: Timer 1, Timer 2, Timer 3 (each one second apart)

Each setTimeout callback retains access to its own i value thanks to closures.

**DOM (Document Object Model) :**

The **DOM (Document Object Model)** is a programming interface for web documents. It represents the structure of a webpage as a **tree of objects**, allowing developers to manipulate HTML and CSS using JavaScript.

🌳 DOM Structure Overview

* The DOM treats every HTML element as an **object**.
* These objects are organized in a **tree-like hierarchy**:
* The root is the document object.
* Child nodes include elements like <html>,<body>,<img> etc.
* Each element can have attributes, text nodes, and other child elements.

🔧 Common DOM Objects

| **DOM Object** | **Description** |
| --- | --- |
| document | Represents the entire HTML document. |
| window | Represents the browser window; contains the document object. |
| element | Represents an HTML element (e.g.,  ,  , **Error! Filename not specified.**). |
| node | Represents any node in the DOM tree (element, text, comment, etc.). |
| event | Represents events like clicks, keypresses, etc. |
| navigator | Provides information about the browser. |
| location | Contains info about the current URL. |
| history | Allows navigation through browser history. |

🛠️ DOM Manipulation with JavaScript

You can use JavaScript to interact with DOM objects:

// Access an element const heading = document.getElementById("main-heading"); // Change its content heading.textContent = "Welcome to the DOM!"; // Add a class heading.classList.add("highlight"); // Respond to an event heading.addEventListener("click", () => { alert("Heading clicked!"); });

Here’s a simple and clear example that uses var, let, and const together in one JavaScript snippet to highlight their differences:

🧪 Example: Comparing var, let, and const

function demoVariables() {

var x = 10; // Function-scoped

let y = 20; // Block-scoped

const z = 30; // Block-scoped and cannot be reassigned

if (true) {

var x = 100; // Re-declared and updated (still in function scope)

let y = 200; // New block-scoped variable (separate from outer y)

// z = 300; // ❌ Error: Assignment to constant variable

console.log("Inside block:");

console.log("x (var):", x); // 100

console.log("y (let):", y); // 200

console.log("z (const):", z); // 30

}

console.log("Outside block:");

console.log("x (var):", x); // 100 (updated)

console.log("y (let):", y); // 20 (original y)

console.log("z (const):", z); // 30

}

demoVariables();

🔍 Key Takeaways

* var is **function-scoped** and can be **re-declared and reassigned**.
* let is **block-scoped**, can be **reassigned**, but **not re-declared** in the same scope.
* const is **block-scoped**, **cannot be reassigned**, and **must be initialized**.

**ECMA script (ES6 + features)**

ES6 (ECMAScript 2015) introduced a major update to JavaScript, bringing powerful new syntax and features that make code more concise, readable, and maintainable. Here's a curated list of the most useful ES6 features with examples:

🚀 1. let and const

Block-scoped variable declarations.

let count = 1; const PI = 3.14159;

* let allows reassignment but is block-scoped.
* const is also block-scoped but cannot be reassigned.

🏹 2. Arrow Functions

Shorter syntax for writing functions.

const add = (a, b) => a + b;

* Implicit return for single expressions.
* Lexically binds this.

🧩 3. Destructuring Assignment

Extract values from arrays or objects.

const person = { name: "Raj", age: 25 }; const { name, age } = person; const colors = ["red", "blue"]; const [firstColor, secondColor] = colors;

🌈 4. Template Literals

String interpolation using backticks.

const name = "Raj"; console.log(`Hello, ${name}!`);

📦 5. Spread and Rest Operators (...)

Spread: expands arrays/objects. Rest: collects remaining elements.

const nums = [1, 2, 3]; const moreNums = [...nums, 4, 5]; // Spread function sum(...args) { return args.reduce((a, b) => a + b); }

🔁 6. for...of Loop

Iterates over iterable objects.

const fruits = ["apple", "banana"]; for (const fruit of fruits) { console.log(fruit); }

🗺️ 7. Maps and Sets

New data structures.

const map = new Map();

map.set("a", 1);

const set = new Set([1, 2, 2, 3]); // Unique values

🏛️ 8. Classes

Syntactic sugar for constructor functions.

class Animal { speak() { console.log("The animal makes a sound"); } } const dog = new Animal(); dog.speak();

📤 9. Modules

Encapsulation and reuse via import/export.

// math.js export const add = (a, b) => a + b; // app.js import { add } from './math.js';

⏳ 10. Promises

Handle asynchronous operations.

const fetchData = () => new Promise((resolve) => setTimeout(() => resolve("Done"), 1000)); fetchData().then(console.log);

Functional programming (FP) in JavaScript is a paradigm that treats computation as the evaluation of mathematical functions and avoids changing state or mutable data. It emphasizes **pure functions**, **immutability**, and **function composition**, leading to cleaner, more predictable code.

🧪 Core Concepts of Functional Programming

* **Pure Functions**: Always return the same output for the same input, with no side effects.
* **Immutability**: Data is never modified; instead, new data structures are returned.
* **Higher-Order Functions**: Functions that take other functions as arguments or return them.
* **Function Composition**: Combining simple functions to build complex ones.
* **Declarative Style**: Focuses on *what* to do rather than *how* to do it.

✅ Example: Filtering Even Numbers from an Array

Here's a simple functional approach using map, filter, and pure functions:

const arr = [1, 2, "3", "4", 5, 6, "7", 8, "9"]; // Pure function to convert to number const toNumber = el => Number(el); // Pure function to check even const isEven = el => el % 2 === 0; // Compose functions const result = arr.map(toNumber).filter(isEven); console.log(result); // Output: [2, 4, 6, 8]

This code:

* Converts all elements to numbers using map.
* Filters out even numbers using filter.
* Avoids mutating the original array.