**Closures and Scope in JavaScript**

Closures and scope are fundamental concepts in JavaScript, particularly when dealing with **functions and variable access**.

**1. Understanding Scope in JavaScript**

**Types of Scope:**

1. **Global Scope** – Variables declared outside any function.
2. **Function (Local) Scope** – Variables declared inside a function.
3. **Block Scope (ES6)** – Variables declared with let or const inside {}.

**Example: Different Scopes**

// Global scope

let globalVar = "I'm global";

function exampleFunction() {

// Function scope

let localVar = "I'm local";

console.log(globalVar); // ✅ Can access global variable

}

exampleFunction();

console.log(localVar); // ❌ ERROR: localVar is not defined

**2. What is a Closure?**

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

**Example of a Closure**

function outerFunction() {

let outerVar = "I'm from outer function";

return function innerFunction() {

console.log(outerVar); // ✅ Still has access to outerVar

};

}

const myClosure = outerFunction();

myClosure(); // Output: "I'm from outer function"

📌 **Key Concept**:  
Even though outerFunction has finished executing, innerFunction **remembers** outerVar because of **closure**.

**3. Practical Uses of Closures**

**3.1 Encapsulation (Data Privacy)**

Closures help create **private variables** that cannot be accessed directly.

function counter() {

let count = 0; // Private variable

return {

increment: function () {

count++;

console.log("Count:", count);

},

decrement: function () {

count--;

console.log("Count:", count);

}

};

}

const myCounter = counter();

myCounter.increment(); // Count: 1

myCounter.increment(); // Count: 2

myCounter.decrement(); // Count: 1

console.log(myCounter.count); // ❌ Undefined (private variable)

📌 **Key Concept**: count is private and can only be modified using increment or decrement.

**3.2 Function Factory (Generating Custom Functions)**

Closures can **customize functions dynamically**.

function multiplier(factor) {

return function (num) {

return num \* factor;

};

}

const double = multiplier(2);

const triple = multiplier(3);

console.log(double(5)); // Output: 10

console.log(triple(5)); // Output: 15

📌 **Key Concept**: multiplier(2) creates a function that **remembers** factor = 2.

**3.3 Event Listeners with Closures**

function attachEventListener() {

let count = 0;

document.getElementById("clickBtn").addEventListener("click", function () {

count++;

console.log("Button clicked", count, "times");

});

}

attachEventListener();

📌 **Key Concept**: The event handler **remembers** count because of the closure.

**3.4 Memoization (Performance Optimization)**

Closures can store **cached results** to speed up function calls.

function memoizedAdd() {

let cache = {};

return function (num) {

if (num in cache) {

console.log("Fetching from cache...");

return cache[num];

} else {

console.log("Calculating result...");

let result = num + 10;

cache[num] = result;

return result;

}

};

}

const add10 = memoizedAdd();

console.log(add10(5)); // Calculating result... 15

console.log(add10(5)); // Fetching from cache... 15

📌 **Key Concept**: cache persists across function calls.

**4. Summary**

| **Concept** | **Example** |
| --- | --- |
| **Lexical Scope** | Inner functions remember outer variables |
| **Basic Closure** | function inner() { console.log(outerVar); } |
| **Data Privacy** | counter() with increment() & decrement() |
| **Custom Functions** | multiplier(2) returns num \* 2 |
| **Event Handling** | Event listeners retain access to variables |
| **Memoization** | Caching function results for performance |

Would you like a **real-world project using closures**? 🚀