In JavaScript, a **function within a function** is known as a **nested function** or sometimes an **inner function**.

**Example:**

function outerFunction() {

console.log("I am the outer function");

function innerFunction() {

console.log("I am the inner function");

}

innerFunction(); // Call the inner function from within the outer function

}

outerFunction();

**Key Points:**

* The inner function can access variables declared in the outer function (**lexical scoping**).
* The outer function cannot access variables declared inside the inner function.
* This is commonly used in **closures**, where the inner function "remembers" the environment in which it was created.

Here's a **simple and clear note** on **Lexical Scope** and **Closures** in JavaScript, along with practical **examples** and explanation.

**📘 LEXICAL SCOPE**

**✅ Definition:**

**Lexical scope** means **a function can access variables from its outer (parent) scope** where it was defined — not where it was called.

JavaScript uses **lexical scoping**, which means scope is determined by the **position of functions and blocks in the code during writing**, not at runtime.

**🧠 Think of it like:**

Inner functions can "see" variables of outer functions, but not the other way around.

**✅ Example:**

function outer() {

let name = "Sanjib";

function inner() {

console.log("Hello " + name); // inner can access 'name' from outer

}

inner();

}

outer();

**🧾 Output:**

Hello Sanjib

👉 inner() has access to the name variable, because it's defined **inside** outer() — **lexical scope**.

**🔒 CLOSURE**

**✅ Definition:**

A **closure** is created when a function **remembers** and **has access to variables from its lexical scope**, even after the outer function has finished executing.

**📌 In other words:**

Closure = Function + Outer Variables it remembers.

**✅ Example:**

function outer() {

let count = 0;

return function inner() {

count++;

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

};

}

let counter = outer(); // outer runs once, returns inner

counter(); // Count is 1

counter(); // Count is 2

counter(); // Count is 3

**🧾 Output:**

Count is 1

Count is 2

Count is 3

**💡 What's happening?**

* outer() runs and returns inner().
* inner() still **remembers** the count variable even though outer() has finished.
* This is a **closure** — inner() "closes over" count.

**🔍 Summary**

| **Term** | **Meaning** | **Key Point** |
| --- | --- | --- |
| **Lexical Scope** | Scope is determined by code position | Inner function can access outer variables |
| **Closure** | Function + remembered variables | Allows data to persist across calls |

**🔧 Real-Life Use Case of Closures:**

**✅ Example 1: Click Counter on a Button**

Imagine you have a website with a button, and you want to **count how many times** it is clicked. You don’t want to expose the counter globally — so you use a **closure**.

**✅ Code:**

<button id="clickBtn">Click Me</button>

<p id="output"></p>

<script>

function clickCounter() {

let count = 0;

return function () {

count++;

document.getElementById("output").innerText = `Button clicked ${count} times`;

};

}

const handleClick = clickCounter(); // Closure is created

document.getElementById("clickBtn").addEventListener("click", handleClick);

</script>

**🧠 What’s Happening:**

* clickCounter() runs once and returns the inner function.
* That returned function is used as the event listener.
* Each time the button is clicked, the function **remembers the count** from the previous call.
* This memory is possible because of **closure**.

**🧰 Example 2: Private Variables**

JavaScript doesn’t have built-in "private" variables, but closures let us **hide data** from outside.

function createBankAccount() {

let balance = 1000;

return {

deposit: function (amount) {

balance += amount;

console.log("Deposited:", amount);

},

checkBalance: function () {

console.log("Balance:", balance);

}

};

}

const myAccount = createBankAccount();

myAccount.deposit(500); // Deposited: 500

myAccount.checkBalance(); // Balance: 1500

✅ balance is not directly accessible — only through the returned functions.  
This is like creating **encapsulated (private)** data using **closures**.

**🔁 Why Are Closures Useful?**

| **Use Case** | **How Closures Help** |
| --- | --- |
| **Encapsulation** | Hide variables like balance in a bank account |
| **Event Listeners** | Retain data for click counters or toggles |
| **Functional Programming** | Return functions with specific behavior (e.g., math operations) |
| **Async Programming** | Preserve state inside setTimeout, setInterval, etc. |