## WEBLAB ASSIGNMENT - JAVASCRIPT CLOSURES

NAME: NIHAREEKA MOHANTY

## Q. What is a Closure in JavaScript?

A closure in JavaScript is formed when a function is declared inside another function and continues to have access to the outer function’s variables even after the outer function has finished executing. This behavior is due to JavaScript’s lexical scoping.

Whenever a function is created inside another, it captures and “closes over” its surrounding environment, allowing it to remember and access those variables even when it's invoked later, outside the original scope.

Example:  
function createCounter() {  
 let count = 0;  
 return function() {  
 count++;  
 return count;  
 };  
}

const counter = createCounter();  
console.log(counter()); //1  
console.log(counter()); //2

Here, the returned function retains access to count, even after createCounter() finishes running—this is a closure in action.

## Q. How does a closure relate to lexical scope?

Closures are built on top of lexical scoping—a rule that states the scope of a variable is defined by where it appears in the source code. When a function is defined, it remembers the environment in which it was created.

Even if an inner function is returned and executed in a different context, it still retains access to the variables from the scope where it was originally declared.

Example:  
function outer() {  
 let outerVar = 'Hello from outer scope';  
 return function inner() {  
 console.log(outerVar);  
 };  
}

const greet = outer();  
greet(); // "Hello from outer scope"

The inner() function remembers the value of outerVar because of lexical scope, even though outer() has already completed.

## Q. Provide one real-world example where closures are useful (e.g., data privacy, event handlers).

Closures are often used for data encapsulation—to keep certain variables private and expose only specific behaviors.

A real-world scenario is creating a private counter that can’t be modified directly from outside the function but can be accessed through a controlled interface.

Example:  
function createCounter() {  
 let count = 0;  
 return function() {  
 count++;  
 return count;  
 };  
}

const counter = createCounter();  
console.log(counter()); // 1  
console.log(counter()); // 2

Here, the variable count is private to createCounter(). External code can't access it directly, but can interact with it via the returned function.

Closures like this are useful in:  
- Maintaining internal state  
- Avoiding global variables  
- Creating modules or factory functions

## Q. Explain how closures are different from global or local variables. Why is it beneficial to use closures instead of global variables in the counter example from Exercise 1?

Global variables are accessible throughout the entire program and persist as long as the application runs. They're vulnerable to accidental overwrites and can cause naming conflicts.  
Local variables, on the other hand, exist only inside the function where they're declared and are discarded once that function ends.  
Closures fall somewhere in between: they allow a function to "remember" its local variables by bundling them with the function, even after the outer function exits.

In the counter example:  
Using a closure instead of a global variable:  
- Hides the internal count variable from outside interference  
- Allows multiple independent counters without shared state  
- Promotes better modularity and safety in code

Closures provide encapsulated and private state management—something that global variables cannot offer.

## Q. In Exercise 3, how does the closure ensure that each multiplier function retains its own factor value? What would happen if the factor was stored in a global variable instead?

In Exercise 3, each call to multiplierFactory(factor) returns a new function that remembers its specific factor value. This is possible because the returned function forms a closure over the factor variable at the time it was created.

Example:  
function multiplierFactory(factor) {  
 return function(number) {  
 return number \* factor;  
 };  
}

const double = multiplierFactory(2);  
const triple = multiplierFactory(3);

console.log(double(5)); // 10  
console.log(triple(5)); // 15

Each multiplier function maintains its own copy of factor through closure, allowing them to work independently.

If factor was global:  
- All multiplier functions would rely on the same shared factor  
- Changing the global variable would break all functions  
- There would be no separation of state, leading to bugs and unpredictable behavior

Closures eliminate such issues by preserving individual values in isolated scopes, making each function consistent and reliable.