Javascript Assignment 9

1. Carefully observe this example.

a) Is the InnerFunction() a closure?

b) What is output of this program

function OuterFunction()

{ var outerVariable = 100;

function InnerFunction() {

alert(outerVariable);

}

return InnerFunction;

}

var innerFunc = OuterFunction();

innerFunc();

Ans:

a) Yes, **InnerFunction()** is a closure. A closure is a function that has access to its own scope, the scope of the outer function (**OuterFunction()** in this case), and the global scope. In this code, **InnerFunction()** has access to the **outerVariable** defined in the outer function, which makes it a closure.

b) The output of this program will be an alert dialog that displays the value of **outerVariable**, which is 100. When **innerFunc()** is called, it invokes **InnerFunction()**, and within **InnerFunction()**, **outerVariable** is accessed and its value (100) is alerted. So, you will see an alert dialog with the number 100 as the output of the program.

2. What is the difference between a closure and a scope ?

Ans:

Closures and scopes are related concepts in JavaScript, but they have different meanings and purposes:

1. Scope:
   * Scope refers to the context in which variables and functions are defined and accessed in JavaScript.
   * JavaScript has function scope and block scope. Variables declared inside a function are typically limited to that function's scope (function scope), while variables declared within a block (e.g., within an **if** statement or a **for** loop) are limited to that block's scope (block scope) in modern JavaScript with **let** and **const**.
   * Scope determines the visibility and lifetime of variables and functions. Variables declared in a narrower scope are not directly accessible from a broader scope.
2. Closure:
   * A closure is a function that has access to variables from its own scope, the scope in which it was defined (outer function's scope), and the global scope, even after the outer function has finished executing.
   * Closures are created when a nested function references variables or functions from its containing (outer) function, and it "captures" those variables, effectively preserving their values.
   * Closures are often used to create private variables, data encapsulation, and to maintain state in functions, even when the outer function has completed its execution.
3. What is a lexical scope and how is it related to closure?

Ans:

Lexical scope, also known as static scope, is a concept in programming languages that determines how variable names are resolved in nested functions based on the physical structure of the code. In lexical scope, the scope of a variable is determined by its location in the source code at the time of writing, not at runtime. This means that variables are bound to the scope in which they are declared in the source code, and this binding remains fixed during the program's execution.

In JavaScript, lexical scope is closely related to closures. When a function is defined within another function, it creates a closure. Closures in JavaScript have access to variables from their containing (outer) function's scope, and this access is determined by the lexical scope. In other words, closures "capture" variables from their lexical scope, allowing them to be used even after the outer function has finished executing.

function outer() {

const outerVar = "I am from outer!";

function inner() {

console.log(outerVar); // Accessing outerVar from the lexical scope

}

return inner;

}

const closureFunc = outer(); // outer() returns inner() as a closure

closureFunc(); // Calls the closure, which still has access to outerVar

4. Output of following closure ?

for (var i = 0; i < 3; i++) {

setTimeout(function log() {

console.log(i); // What is logged?

}, 1000);}

The output of this code is:

3

3

3

To achieve the expected behavior where each log statement prints a different value from the loop (0, 1, and 2), you can use a block-scoped variable like **let** inside the loop:

for (let i = 0; i < 3; i++) {

setTimeout(function log() {

console.log(i); // This will log 0, 1, 2

}, 1000);

}