**NAME: SODOLAMU AYOMIDE FOLADAYO**

**MATRIC NO: 23/0001**

**DEPARTMENT: INFORMATION TECHNOLOGY (GRP C)**

**ASSIGNMENTS**

**HIGHER ORDER FUNCTIONS:** Are functions that can take other functions as arguments or return them as results. This ability makes them powerful in functional programming, allowing for more abstract and reusable code.

1.**SET TIMEOUT is a higher-order function that takes two parameters: a callback function to execute after a specified delay (in milliseconds) and the delay duration.**

**For example:**

**function greet() {**

**console.log("Hello, world!");**

**}**

**// Using setTimeout as a higher-order function**

**setTimeout(greet, 1000); // Logs "Hello, world!" after 1 second**

**Here, setTimeout accepts the greet function as an argument, making it a higher-order function.**

**You can also define the callback function inline, which is common in JavaScript:**

**setTimeout(() => {**

**console.log("Hello after 1 second");**

**}, 1000);**

**2,SETINTERVAL : is another higher-order function that repeatedly calls a function at specified intervals until you clear it.**

**For example:**

**function sayHello() {**

**console.log("Hello, again!");**

**}**

**// Using setInterval as a higher-order function**

**const intervalId = setInterval(sayHello, 2000); // Logs "Hello, again!" every 2 seconds**

**// To stop the interval after some time**

**setTimeout(() => clearInterval(intervalId), 10000); // Stops after 10 seconds**

**Here, setInterval takes the sayHello function as an argument and repeats it every 2 seconds.**

**Since setTimeout and setInterval both take a callback function and invoke it later, they fall under the definition of higher-order functions. They allow us to defer or schedule the execution of other functions, which is a powerful way to control asynchronous code.**

**ASSIGNMENT 2**

**Closures in JavaScript : can be defined as a fundamental concept, enabling functions to access variables from an outer function even after that outer function has finished executing. A closure is created when a function is defined within another function and retains access to the outer function’s variables.**

**Closures are particularly useful for creating private variables, maintaining state across function calls, and building functions with specific configurations. For examples:**

**1. Basic Closure Example**

**function outerFunction() {**

**let outerVariable = "I'm outside!";**

**function innerFunction() {**

**console.log(outerVariable);**

**}**

**return innerFunction;**

**}**

**const myClosure = outerFunction();**

**myClosure(); // Output: "I'm outside!"**

**In this example:**

**• innerFunction is defined within outerFunction.**

**• outerFunction returns innerFunction, which retains access to outerVariable even after outerFunction has completed.**

**• When myClosure is called, it still has access to outerVariable because of the closure.**

**2. Using Closures to Create Private Variables**

**Closures are often used to create private variables. By defining variables within a function scope and returning inner functions that access those variables, we can keep data private.**

**For example:**

**function counter() {**

**let count = 0;**

**return function() {**

**count += 1;**

**return count;**

**};**

**}**

**const increment = counter();**

**console.log(increment()); // Output: 1**

**console.log(increment()); // Output: 2**

**console.log(increment()); // Output: 3**

**In this case:**

**• The count variable is private to the counter function.**

**• The inner function (closure) has access to count and increments it each time increment is called.**

**• No external code can directly access or modify count, as it’s encapsulated within the closure.**

**3. Closures with Parameters**

**Closures can also capture parameters passed to the outer function, making them customizable.**

**For example:**

**function createMultiplier(multiplier) {**

**return function(number) {**

**return number \* multiplier;**

**};**

**}**

**const double = createMultiplier(2);**

**const triple = createMultiplier(3);**

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

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

**In this example:**

**• createMultiplier takes multiplier as a parameter.**

**• The inner function (closure) retains access to multiplier, even after createMultiplier has completed.**

**• double and triple are functions with customized behavior based on the multiplier they captured.**

**4. Closures in Asynchronous Functions**

**Closures are also essential in asynchronous code, allowing functions to “remember” variables even after asynchronous operations.**

**function delayedLogger() {**

**for (let i = 1; i <= 3; i++) {**

**setTimeout(function() {**

**console.log(i); // Outputs: 1, 2, 3**

**}, i \* 1000);**

**}**

**}**

**delayedLogger();**

**In this example:**

**• Each setTimeout callback is a closure that captures the current value of i.**

**• Using let instead of var ensures that each iteration of the loop has its own block scope, so each setTimeout remembers a different value of i.**

**If var was used instead, each setTimeout callback would refer to the same variable, causing all callbacks to print the same value.**