*Wadood Alam*

# Antra Assignment 2

1. Why are closures useful in JavaScript? Give an example use case.

**Ans:** Closures in JavaScript mean that a function contains access to the variables in the function's containing scope despite the execution of the outer function. Simply, it means that the inner function(inner scope) accesses variables from an outer function(outer scope) even after the outer function has been executed. Closure functions are particularly useful in cases where data encapsulation is needed i.e. create a private variable that is only accessible in its declared scope and not anywhere else. Another area where closures are useful is for stateful functions where a state can be updated or retrieved over a function's lifecycle.
An example use-case for closures can be:

```
function createCounter() {
        Let count = 0;
        Return function () {
                count ++;
                Return count;
        };
};
Const counter = createCounter();
console.log(counter());         // output is 1
console.log(counter());         // output is 2
```

The createCounter function essentially returns another function('inner' function). The inner function forms a closure around the *count* variable in the outer function. Every time the *counter* is called, it performs the execution as desired(incrementing the count in this case) but ensures that the *count* variable is kept private and not accessible through the outside of the outer function directly(encapsulation applied). However, in the context of the inner function, *count* will still be accessible to the inner function despite the outer function having finished executing.

2. When should you choose to use "let" or "const"?

**Ans:** When choosing either let or const, you should first assess where in the code you want to use them. The scope of both these declaration types is restricted to the 'box' they are declared in. For example, within an 'If' condition, if you declare a const or let you cannot access them outside of that if condition. If you do not want a global variable having a global scope, you should use let or const. Another important thing to take into consideration is that whenever you want to restrict the declaration, you should use let or const as both do not allow redeclaration of themselves within the same scope. Also, both of them are not hoisted.
However, when choosing between let or char, you need to consider if there will be a reassignment of value or not. In the case of const, you cannot reassign the value, for example

*const MAX_USERS = 100;*
MAX_USERS = 200; // This will throw an error

But in the case of let, you can update the value of the declared let. For example:

*let counter = 0;*
*counter = 10; // Reassigning the value of 'counter'*

3. Give an example of a common mistake related to hoisting and explain how to fix it

**Ans:** A common mistake related to hoisting, according to my understanding, is accessing the variable assuming that hoisting initializes the var as well, but it does not do that. Hoisting is when the declaration of var is moved at the top of the code, but not initialization. Before initialization(but after declaration/hoisting), if the variable is accessed, it will have an undefined value. For example:

*console.log(myVar); // Outputs: undefined*
*var myVar = 42; // Declaration is hoisted, but initialization stays here*

To avoid this confusion and mistake, ensure that you declare the var at the top and initialize it either at the top or when first accessing the var in the code. The fix would be

*var myVar = 42; // Declaration and initialization at the top*
*console.log(myVar); // then access myVar: Outputs: 42*

4. What will the outcome of each console.log() be after the function calls? Why?

```
const arr = [1, 2];
function foo1(arg) {
  arg.push(3);
}
foo1(arr);
console.log(arr);
```

OUTPUT IS [1,2,3].
Since arg uses the byReference concept, it points to the original *arr* and hence pushes the value of 3 in the array.

```
function foo2(arg) {
  arg = [1, 2, 3, 4];
}
foo2(arr);
console.log(arr);
```

OUTPUT IS [1,2,3].
Arg references this new array [1,2,3.4], hence, the original *arr* remains as is because it is a different reference than arg.

```
function foo3(arg) {
  let b = arg;
  b.push(3);
}
```

```
foo3(arr);
console.log(arr);
```

OUTPUT IS [1,2,3,3].

Now, b is assigned as a reference to arg, which is assigned as a reference to the original *arr*. Hence, any changes made to b will mean changes to the original *arr*. Thus. b.push(3) will result in appending the original *arr* with 3.

```
function foo4(arg) {
  let b = arg;
  b = [1, 2, 3, 4];
}
foo4(arr);
console.log(arr);
```

OUTPUT IS [1,2,3,3].

In the first line within the function, b is referenced to arg and obviously arg is referenced to the original *arr*. However, in the second line i.e *b = [1,2,3,4]*, b now refers to a new array(reference), hence, the original *arr* will still be the same as before resulting in [1,2,3,4].