JavaScript Closures

A **closure** is a function along with its **lexical environment** bundled together.

In JavaScript, a closure allows an **inner function** to access the variables of its **outer function**, even after the outer function has executed and returned.

A closure is created every time a function is defined within another function and retains access to the outer function's scope.

Closure Example 1

```
function x() {
    var z = 10;
    function y() {
        console.log(z);
    }
    y();
}
x();
```

- Here, function `y()` has access to variable `z` from its outer function `x()`.
- Even after `x()` has finished executing, `y()` still has access to `z`.

Closure Example 2

```
function outer() {
    var z = 10;
    function inner() {
        console.log(z);
    }
    return inner; // Closure is returned (inner function + lexical environment)
}

var x = outer(); // Outer function returns inner function with its scope
x(); // Inner function is called and prints 10
```

- When `outer()` is invoked, it returns the `inner()` function.
- The variable `z` is still accessible inside `inner()`, demonstrating closure behavior.

Closures and Loops

Issue with `var` in Loops

```
for (var i = 1; i <= 3; i++) {
    setTimeout(function() {
        console.log(i);
    }, 1000);
}
```

Why does this print `4` three times?

- `var i` is **global**, so all iterations share the same `i`.
- By the time `setTimeout` executes, `i` is already `4`.

Fixing the Issue Using Closures

Solution 1: IIFE (Immediately Invoked Function Expression)

```
for (var i = 1; i <= 3; i++) {
    (function(i) {
        setTimeout(function() {
            console.log(i);
        }, 1000);
    })(i);
}
```

Solution 2: Using a Function to Capture `i`

```
for (var i = 1; i <= 3; i++) {
    function a(i) {
        setTimeout(function() {
            console.log(i);
        }, 1000);
    }
    a(i);
}
```

Solution 3: Using `let` Instead of `var`

```
for (let i = 1; i <= 3; i++) { // `let` creates a new scope for each iteration
    setTimeout(function() {
        console.log(i);
    }, 1000);
}
```

- Each iteration has its own scope due to `let`, preserving the correct value of `i`.

Summary

- A **closure** is created when a function is defined inside another function, allowing access to the **outer function's** variables even after execution.

- **Common use cases of closures** include **data encapsulation**, **function currying**, and **event handlers**.
- **Closures help fix issues with asynchronous code**, such as loops using `var`, by preserving variable values at different execution points.
- **Using `let` instead of `var` in loops automatically creates block-scoped variables, solving common closure-related issues.**

For a deeper dive into closures, check out this JavaScript Closures Video.