JavaScript Higher-Order Functions and Closures

In JavaScript, higher-order functions and closures are key concepts in functional programming. Here's a breakdown of each with examples.

Higher-Order Functions

A higher-order function is a function that either takes another function as an argument, returns a function, or both. Higher-order functions allow for more flexible, reusable, and functional code.

Two common examples of higher-order functions in JavaScript are setTimeout and setInterval.

1. setTimeout:
   - setTimeout is a function that executes a given function after a specified delay (in milliseconds).

   Example:
   ```javascript
   setTimeout(() => {
       console.log("Hello after 2 seconds");
   }, 2000); // 2000 ms = 2 seconds
   ```

In this example, the setTimeout function takes another function as an argument (the () => { console.log("Hello after 2 seconds"); }), making it a higher-order function. It will log the message to the console after a delay of 2 seconds.

2. setInterval:

- setInterval repeatedly executes a given function at specified intervals until it's stopped.

Example:

```javascript
let counter = 0;
const intervalId = setInterval(() => {
    console.log(`Count: ${counter}`);
    counter++;
    if (counter > 5) {
        clearInterval(intervalId); // Stops the interval after 5 counts
    }
}, 1000); // 1000 ms = 1 second
```

Here, setInterval executes the provided function every second, logging the count to the console. It stops after 5 counts using clearInterval.

Closures

A closure is a feature in JavaScript where an inner function has access to the outer (enclosing) function's variables, even after the outer function has finished execution. Closures give functions private variables and are a powerful way to create encapsulated code.

Example of a closure:

```javascript
function outerFunction() {
    let count = 0;

    function innerFunction() {
        count++;
        console.log(`Count: ${count}`);
    }

    return innerFunction;
}

const counter = outerFunction(); // outerFunction returns innerFunction with access to count
counter(); // Count: 1
counter(); // Count: 2
counter(); // Count: 3
```

In this example:

- outerFunction defines a variable count and returns innerFunction.
- innerFunction has access to count because of closure, even though outerFunction has completed its execution.
- Each time counter() is called, count is incremented, and the updated value is logged. This demonstrates a persistent state with a closure.

Closures are widely used for data encapsulation, to create private variables, and in scenarios like event handling, asynchronous code, and functional programming.