JAVASCRIPT ADVANCED HOF'S , CALLBACKS, AND CLOSURES

Understanding Higher-Order Functions (HOFs)

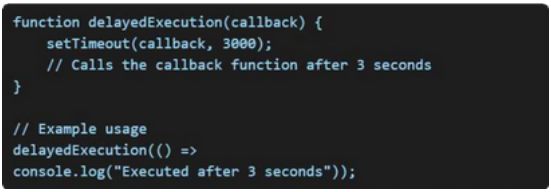
A Higher-Order Function (HOF) is a function that either takes another function as an argument or returns a function. These are widely used in JavaScript to enhance code reusability and maintainability.

1. Delayed Execution Using Callback (HOF + Callback)

Concept:

A function can accept another function as an argument (callback) and execute it after a delay using setTimeout.

Implementation



```
function delayedExecution(callback) {
    setTimeout(callback, 3000);
    // Calls the callback function after 3 seconds
}

// Example usage
delayedExecution(() =>
console.log("Executed after 3 seconds"));
```

Explanation:

- delayedExecution accepts a function callback.
- setTimeout is used to delay the execution of the callback for 3 seconds.
- When called, it logs a message to the console after 3 seconds.

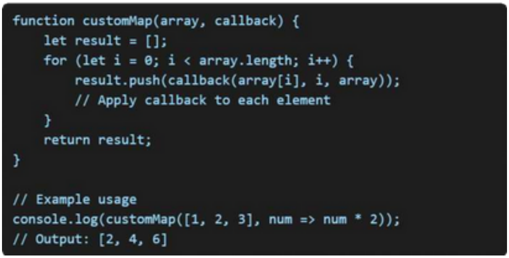
JAVASCRIPT ADVANCED HOF'S , CALLBACKS, AND CLOSURES

2. Implementing Custom Function (HOF)

Concept:

The .map() method is used to transform an array by applying a callback function to each element. We can create our own version of .map().

Implementation



```
function customMap(array, callback) {
    let result = [];
    for (let i = 0; i < array.length; i++) {
        result.push(callback(array[i], i, array));
        // Apply callback to each element
    }
    return result;
}

// Example usage
console.log(customMap([1, 2, 3], num => num * 2));
// Output: [2, 4, 6]
```

Explanation:

- customMap takes an array and a callback function.
- Iterates over the array, applies the callback function to each element, and stores the result.
- Works similarly to Array.prototype.map but is implemented manually.

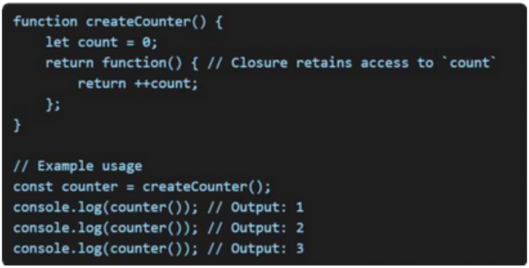
JAVASCRIPT ADVANCED HOF'S , CALLBACKS, AND CLOSURES

3. Closures: Creating a Counter Function

Concept:

A closure is a function that retains access to variables from its outer scope even after the outer function has finished execution.

Implementation



```
function createCounter() {
    let count = 0;
    return function() { // Closure retains access to `count`
        return ++count;
    };
}

// Example usage
const counter = createCounter();
console.log(counter()); // Output: 1
console.log(counter()); // Output: 2
console.log(counter()); // Output: 3
```

Explanation:

- createCounter defines a variable count and returns a function.
- The inner function forms a closure, keeping count in memory.
- Each time the inner function is called, count is incremented and returned.

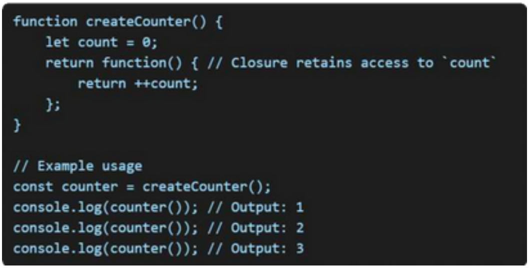
JAVASCRIPT ADVANCED HOF'S , CALLBACKS, AND CLOSURES

4. Limiting Function Calls (Closure + HOF)

Concept:

A function should only be executed a limited number of times. This can be achieved using closures.

Implementation



```
function createCounter() {
    let count = 0;
    return function() { // Closure retains access to `count`
        return ++count;
    };
}

// Example usage
const counter = createCounter();
console.log(counter()); // Output: 1
console.log(counter()); // Output: 2
console.log(counter()); // Output: 3
```

Explanation:

- limit takes a function fn and a limit value.
- It tracks the number of times the function has been called.
- Once the limit is reached, further calls do nothing

JAVASCRIPT ADVANCED HOF'S , CALLBACKS, AND CLOSURES

Conclusion

- Higher-Order Functions (HOFs) enable cleaner and reusable code by accepting functions as arguments.
- Callbacks allow asynchronous behavior and function execution control.
- Closures help retain variable states and create private data.
- These concepts are crucial in modern JavaScript development, especially in functional programming and asynchronous operations.

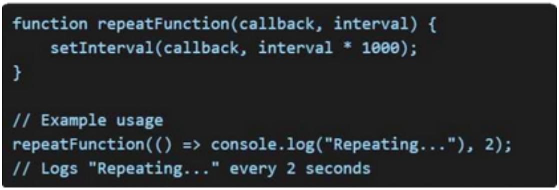
JAVASCRIPT ADVANCED HOF'S , CALLBACKS, AND CLOSURES

1. Repeating a Function at Intervals (Using Callbacks)

Concept:

A callback function is a function passed as an argument to another function. Using setInterval, we can execute a callback function repeatedly at specified intervals.

Implementation



```
function repeatFunction(callback, interval) {
    setInterval(callback, interval * 1000);
}

// Example usage
repeatFunction(() => console.log("Repeating..."), 2);
// Logs "Repeating..." every 2 seconds
```

Explanation:

- The function repeatFunction takes two parameters:
  - callback: The function to execute
  - interval: The time in seconds between executions
- setInterval is used to call callback repeatedly after every interval seconds.
- In the example, the function logs "Repeating..." every 2 seconds.

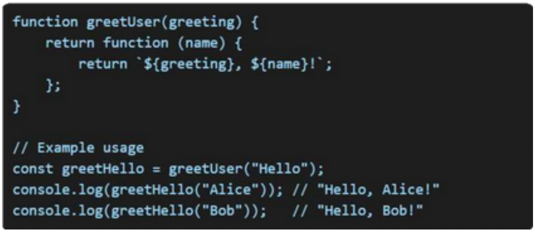
JAVASCRIPT ADVANCED HOF'S , CALLBACKS, AND CLOSURES

2. Creating a Function with a Preset Greeting (Using Closures)

Concept:

A closure allows a function to "remember" variables from its outer scope even after the outer function has finished executing.

Implementation



```
function greetUser(greeting) {
    return function (name) {
        return `${greeting}, ${name}!`;
    };
}

// Example usage
const greetHello = greetUser("Hello");
console.log(greetHello("Alice")); // "Hello, Alice!"
console.log(greetHello("Bob"));   // "Hello, Bob!"
```

Explanation:

- greetUser is a higher-order function that returns another function.
- The returned function remembers the greeting value from greetUser (closure property).
- When greetHello("Alice") is called, it uses the stored greeting ("Hello") and returns "Hello, Alice!".
- This technique is useful for creating pre-configured functions.

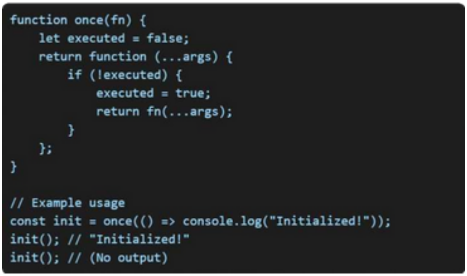
JAVASCRIPT ADVANCED HOF'S , CALLBACKS, AND CLOSURES

3. Executing a Function Only Once (Using HOFs + Closures)

Concept:

A function should only be executed once, no matter how many times it is called.

Implementation



```
function once(fn) {
    let executed = false;
    return function (...args) {
        if (!executed) {
            executed = true;
            return fn(...args);
        }
    };
}

// Example usage
const init = once(() => console.log("Initialized!"));
init(); // "Initialized!"
init(); // (No output)
```

Explanation:

- The once function wraps another function (fn) and ensures it only executes once.
- The variable executed keeps track of whether the function has already been called.
- The first call executes fn, but all subsequent calls do nothing.
- Useful for initialization functions.

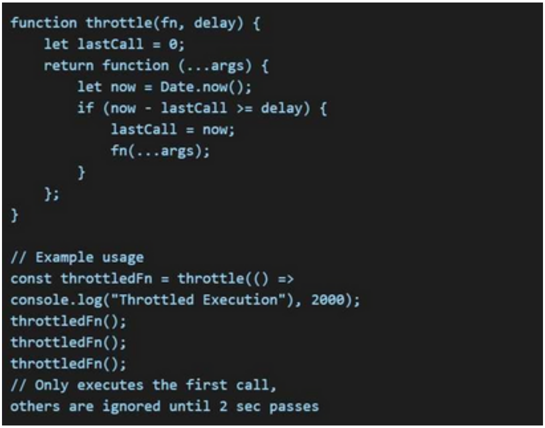
JAVASCRIPT ADVANCED HOF’S , CALLBACKS, AND CLOSURES

4. Throttling a Function (Using HOFs + Closures)

Concept:

Throttling ensures that a function is not executed more than once within a specified time period. This is useful in event listeners to improve performance.

Implementation



```
function throttle(fn, delay) {
    let lastCall = 0;
    return function (...args) {
        let now = Date.now();
        if (now - lastCall >= delay) {
            lastCall = now;
            fn(...args);
        }
    };
}

// Example usage
const throttledFn = throttle(() =>
console.log("Throttled Execution"), 2000);
throttledFn();
throttledFn();
throttledFn();
// Only executes the first call,
others are ignored until 2 sec passes
```

JAVASCRIPT ADVANCED HOF’S , CALLBACKS, AND CLOSURES

Explanation:

- throttle ensures that fn runs only once in every delay milliseconds.
- The lastCall variable stores the last execution time.
- If delay hasn't passed, additional calls are ignored.
- Useful in scenarios like scrolling events or resizing windows.