codeastu

A CIKLUM COMPANY

Javascript Essentials

Closures

Closures

- A closure is a function that has access to variables in its outer (enclosing) scope even after the outer function has returned
- Closures are created every time a function is created, at function creation time
- They allow for data privacy and function factories
- One of the most powerful and important concepts in JavaScript

```
function outerFunction(x) {
    // This is the outer function's scope
    return function innerFunction(y) {
        // This is the inner function's scope
        return x + y; // x is accessible here due to closure
    };
}

const addFive = outerFunction(5);
console.log(addFive(3)); // 8
```

1. Basic Closure Concept

- A closure gives you access to an outer function's scope from an inner function
- The inner function "closes over" the outer function's variables
- This happens even after the outer function has finished executing

```
function createCounter() {
    let count = 0; // This variable is "closed over"

    return function() {
        count++; // Can access and modify count
        return count;
    };
}

const counter1 = createCounter();
const counter2 = createCounter();

console.log(counter1()); // 1
console.log(counter1()); // 2
console.log(counter2()); // 1 (separate closure)
console.log(counter1()); // 3
```

Notes

Each call to `createCounter()` creates a new closure with its own `count` variable

2. Closure with Parameters

- Closures can capture parameters from the outer function
- The parameters become part of the closure's scope
- Useful for creating specialized functions

```
function createMultiplier(multiplier) {
    return function(number) {
        return number * multiplier;
    };
}

const double = createMultiplier(2);
const triple = createMultiplier(3);

console.log(double(5)); // 10
console.log(triple(5)); // 15
console.log(double(10)); // 20
```

References

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures

3. Data Privacy with Closures

- Closures provide a way to create private variables
- Variables in the outer function are not accessible from outside
- Only the inner function can access and modify these variables

```
function createBankAccount(initialBalance) {
    let balance = initialBalance; // Private variable

    return {
        deposit: function(amount) {
            balance += amount;
            return balance;
        },
        withdraw: function(amount) {
            if (amount <= balance) {
                balance -= amount;
                return balance;
            } else {
                return "Insufficient funds";
            }
        },
        getBalance: function() {
            return balance;
        }
    };
}

const account = createBankAccount(1000);
console.log(account.getBalance()); // 1000
console.log(account.deposit(500)); // 1500
console.log(account.withdraw(200)); // 1300
// console.log(balance); // ReferenceError: balance is not defined
```

4. Module Pattern with Closures

- Closures enable the module pattern for creating encapsulated code
- Provides a way to create public and private methods
- Helps organize code and prevent global namespace pollution

```
const Calculator = (function() {
    // Private variables
    let result = 0;
    let history = [];

    // Private methods
    function addToHistory(operation, value) {
        history.push(`${operation}: ${value}`);
    }

    // Public API
    return {
        add: function(num) {
            result += num;
            addToHistory('add', num);
            return this;
        },
        subtract: function(num) {
            result -= num;
            addToHistory('subtract', num);
            return this;
        },
        multiply: function(num) {
            result *= num;
            addToHistory('multiply', num);
            return this;
        },
        getResult: function() {
            return result;
        },
        getHistory: function() {
            return history.slice(); // Return copy of history
        },
        reset: function() {
            result = 0;
            history = [];
            return this;
        }
    };
})();

console.log(Calculator.add(5).multiply(2).getResult()); // 10
console.log(Calculator.getHistory()); // ['add: 5', 'multiply: 2']
```

5. Closures in Loops - The Classic Problem

- A common mistake when using closures in loops
- All closures created in the loop share the same variable reference
- Solutions: Use `let`, `bind()`, or IIFE (Immediately Invoked Function Expression)

```javascript
// Problem: All functions reference the same i
function problemExample() {
    const functions = [];
    for (var i = 0; i < 3; i++) {
        functions.push(function() {
            console.log(i); // All will log 3
        });
    }
    return functions;
}

// Solution 1: Use let instead of var
function solution1() {
    const functions = [];
    for (let i = 0; i < 3; i++) {
        functions.push(function() {
            console.log(i); // Will log 0, 1, 2
        });
    }
    return functions;
}

// Solution 2: Use IIFE
function solution2() {
    const functions = [];
    for (var i = 0; i < 3; i++) {
        functions.push((function(index) {
            return function() {
                console.log(index); // Will log 0, 1, 2
            };
        })(i));
    }
    return functions;
}
```

6. Closures with Event Handlers

- Closures are commonly used in event handlers
- They allow event handlers to access variables from their enclosing scope
- Useful for maintaining state between events

```
function setupButtons() {
    const buttons = document.querySelectorAll('.counter-btn');

    buttons.forEach((button, index) => {
        let count = 0; // Each button gets its own count

        button.addEventListener('click', function() {
            count++;
            this.textContent = `Clicked ${count} times`;
        });
    });
}

// HTML example:
// <button class="counter-btn">Click me</button>
// <button class="counter-btn">Click me too</button>
```

Notes

Each button maintains its own separate count variable due to closures

7. Closures with setTimeout and setInterval

- Closures capture the current value of variables when the function is created
- Useful for maintaining state in asynchronous operations
- Be careful about variable references in loops

```
function delayedGreeting(name) {
    return function() {
        console.log(`Hello, ${name}!`);
    };
}

const greetAlice = delayedGreeting("Alice");
const greetBob = delayedGreeting("Bob");

setTimeout(greetAlice, 1000); // "Hello, Alice!" after 1 second
setTimeout(greetBob, 2000);   // "Hello, Bob!" after 2 seconds

// Example with multiple timeouts
function createCountdown(start) {
    let count = start;

    return function() {
        if (count > 0) {
            console.log(count);
            count--;
        } else {
            console.log("Time's up!");
        }
    };
}

const countdown = createCountdown(3);
const interval = setInterval(countdown, 1000);
```

8. Function Factories with Closures

- Closures enable the creation of function factories
- Each factory function can create specialized functions
- Useful for creating reusable, configurable functions

```
function createValidator(rules) {
    return function(data) {
        const errors = [];

        for (const field in rules) {
            const rule = rules[field];
            const value = data[field];

            if (rule.required && !value) {
                errors.push(`${field} is required`);
            }

            if (rule.minLength && value && value.length < rule.minLength) {
                errors.push(`${field} must be at least ${rule.minLength} characters`);
            }

            if (rule.pattern && value && !rule.pattern.test(value)) {
                errors.push(`${field} format is invalid`);
            }
        }

        return {
            isValid: errors.length === 0,
            errors: errors
        };
    };
}

// Create specific validators
const userValidator = createValidator({
    name: { required: true, minLength: 2 },
    email: { required: true, pattern: /^[^\s@]+@[^\s@]+\.[^\s@]+$/ },
    age: { required: true }
});

const result = userValidator({
    name: "John",
    email: "john@example.com",
    age: 25
});

console.log(result); // { isValid: true, errors: [] }
```

9. Closures for Memoization

- Closures can be used to implement memoization (caching function results)
- Improves performance by avoiding repeated calculations
- The cache is private to the function

```
function createMemoizedFunction(fn) {
    const cache = {}; // Private cache

    return function(...args) {
        const key = JSON.stringify(args);

        if (cache[key]) {
            console.log('Cache hit!');
            return cache[key];
        }

        console.log('Computing...');
        const result = fn.apply(this, args);
        cache[key] = result;
        return result;
    };
}

// Example: Memoized Fibonacci
const fibonacci = createMemoizedFunction(function(n) {
    if (n <= 1) return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
});

console.log(fibonacci(10)); // Computing... 55
console.log(fibonacci(10)); // Cache hit! 55
console.log(fibonacci(5));  // Computing... 5
console.log(fibonacci(5));  // Cache hit! 5
```

10. Closures with Currying

- Closures enable currying (transforming a function with multiple arguments into a sequence of functions)
- Each function returns another function until all arguments are provided
- Useful for creating specialized functions

```
function curry(fn) {
    return function curried(...args) {
        if (args.length >= fn.length) {
            return fn.apply(this, args);
        } else {
            return function(...nextArgs) {
                return curried.apply(this, args.concat(nextArgs));
            };
        }
    };
}

// Example: Curried add function
const add = curry((a, b, c) => a + b + c);

console.log(add(1)(2)(3));     // 6
console.log(add(1, 2)(3));     // 6
console.log(add(1)(2, 3));     // 6
console.log(add(1, 2, 3));     // 6

// Practical example: API calls
const apiCall = curry((baseUrl, endpoint, data) => {
    return fetch(`${baseUrl}${endpoint}`, {
        method: 'POST',
        body: JSON.stringify(data)
    });
});

const myApiCall = apiCall('https://api.example.com');
const userEndpoint = myApiCall('/users');
const createUser = userEndpoint({ name: 'John', email: 'john@example.com' });
```

11. Closures and Memory Management

- Closures keep references to their outer scope variables
- This can lead to memory leaks if not managed properly
- Be aware of what variables are being captured

```
// Potential memory leak
function createLeakyFunction() {
    const largeData = new Array(1000000).fill('data'); // Large array

    return function() {
        console.log('Function called');
        // Even though we don't use largeData, it's still in memory
    };
}

// Better approach - only capture what you need
function createEfficientFunction() {
    const largeData = new Array(1000000).fill('data');
    const processedData = largeData.map(item => item.toUpperCase()); // Process data

    return function() {
        console.log('Function called');
        // Only processedData is captured, largeData can be garbage collected
        return processedData.length;
    };
}

// Clean up closures when done
function createCleanupFunction() {
    let data = { value: 42 };

    const fn = function() {
        return data.value;
    };

    // Provide cleanup method
    fn.cleanup = function() {
        data = null;
    };

    return fn;
}
```

12. Closures in Modern JavaScript

- Closures work seamlessly with modern JavaScript features
- Arrow functions also create closures
- Useful with async/await and Promises

```javascript
// Closures with async/await
function createAsyncCounter() {
    let count = 0;

    return {
        async increment() {
            await new Promise(resolve => setTimeout(resolve, 100));
            count++;
            return count;
        },
        async decrement() {
            await new Promise(resolve => setTimeout(resolve, 100));
            count--;
            return count;
        },
        getCount() {
            return count;
        }
    };
}

// Usage
async function example() {
    const counter = createAsyncCounter();

    console.log(await counter.increment()); // 1
    console.log(await counter.increment()); // 2
    console.log(await counter.decrement()); // 1
}

// Closures with Promises
function createPromiseFactory() {
    let requestId = 0;

    return function(url) {
        const currentId = ++requestId;

        return fetch(url)
            .then(response => {
                console.log(`Request ${currentId} completed`);
                return response.json();
            })
            .catch(error => {
                console.log(`Request ${currentId} failed:`, error);
                throw error;
            });
    };
}
```

13. Common Closure Patterns

- **Module Pattern**: Encapsulate code and create public/private APIs
- **Factory Pattern**: Create objects with private state
- **Observer Pattern**: Maintain lists of observers with private state
- **Singleton Pattern**: Ensure only one instance exists

```
// Observer Pattern with Closures
function createEventEmitter() {
    const events = {}; // Private event registry

    return {
        on(event, callback) {
            if (!events[event]) {
                events[event] = [];
            }
            events[event].push(callback);
        },

        emit(event, data) {
            if (events[event]) {
                events[event].forEach(callback => callback(data));
            }
        },

        off(event, callback) {
            if (events[event]) {
                events[event] = events[event].filter(cb => cb !== callback);
            }
        }
    };
}

// Usage
const emitter = createEventEmitter();

emitter.on('userLogin', (user) => {
    console.log(`User ${user.name} logged in`);
});

emitter.emit('userLogin', { name: 'Alice', id: 1 });
```

14. Closure Best Practices

- **Understand scope**: Know what variables are being captured
- **Avoid memory leaks**: Don't capture unnecessary large objects
- **Use meaningful names**: Make closure purposes clear
- **Consider performance**: Closures have a small performance overhead
- **Test thoroughly**: Closures can make debugging more complex

```
// Good: Clear purpose and minimal scope
function createIdGenerator(prefix = 'id') {
    let counter = 0;
    return () => `${prefix}_${++counter}`;
}

// Good: Proper cleanup
function createResourceManager() {
    const resources = new Set();

    const manager = {
        add(resource) {
            resources.add(resource);
            return resource;
        },
        remove(resource) {
            resources.delete(resource);
        },
        cleanup() {
            resources.clear();
        }
    };

    return manager;
}

// Avoid: Capturing unnecessary data
function badExample() {
    const largeObject = { /* lots of data */ };
    const smallValue = largeObject.importantValue;

    return function() {
        return smallValue; // Only need smallValue, not largeObject
    };
}
```

References

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures
- https://eloquentjavascript.net/03_functions.html#h_hOd2VLinKD