Understanding Closures in JavaScript

Closures in JavaScript is a fundamental concept to master as you progress in your JavaScript journey. Let`s delve into what closures are, how they work, and examine them in depth with practical code examples.

Lexical Scope

Before we embark on understanding closures, it is crucial to have a good grasp of the concept of scope, particularly lexical scope. Lexical scope governs how variable names are resolved in nested functions. In essence, if we have a child function nested within a parent function, the child function has access to the scope of the parent function and the global scope.

Consider the following JavaScript code:

```
let x = 1;

const parentFunction = () => {
    let myValue = 2;
    console.log(x); // Outputs: 1
    console.log(myValue); // Outputs: 2
}

parentFunction();
console.log(myValue); // Outputs: Reference Error
```

In the code above, the `parentFunction` is a child of the global scope. Thus, it has access to the global variable `x` and the locally defined `myValue`. However, if we attempt to log the `myValue` from the global scope, we get a reference error because it does not have access to variables defined in the local scope of `parentFunction`.

Lexical Scope and Closure: The Relationship

While lexical scope is often mistaken for closure, it`s important to note that lexical scope is only an essential part of closure, not the entire concept.

In our next code example, we extend our understanding of lexical scope by creating a nested child function within the `parentFunction`.

```
let x = 1;

const parentFunction = () => {
    let myValue = 2;
    console.log(x); // Outputs: 1
    console.log(myValue); // Outputs: 2

    const childFunction = () => {
        x += 5;
        myValue += 1;
        console.log(x); // Outputs: 6
        console.log(myValue); // Outputs: 3
    }

    childFunction();
}

parentFunction();
```

In the above example, `childFunction` has access to both the parent scope (`parentFunction`) and the global scope. It can manipulate the variables `x` and `myValue` due to this access. This example demonstrates the power of lexical scoping, but not closure per se.

Closure: The Real Deal

A brief definition of a closure, as stated on the W3Schools website, is **"a function having access to the parent scope, even after the parent function has closed."** This definition underscores the vital characteristic of closures: **they are functions that retain access to the parent`s scope even after the parent function has finished execution**.

We can understand closures better with the same code example:

```
let x = 1;

const parentFunction = () => {
    let myValue = 2;
    console.log(x); // Outputs: 1
    console.log(myValue); // Outputs: 2

    const childFunction = () => {
        x += 5;
        myValue += 1;
        console.log(x);
        console.log(myValue);
    }

    return childFunction;
}

const result = parentFunction();
// Outputs: 1, 2; `result` is now a reference to `childFunction`

result();
// Outputs: 6, 3; `childFunction` has access to the `myValue` and `x`
even after `parentFunction` has closed

result();
// Outputs: 11, 4; `childFunction` maintains the changes to `x` and
`myValue` across calls
```

In this example, the `parentFunction` returns the `childFunction`. When we call `parentFunction`, it executes and returns a reference to `childFunction`, which we store in `result`.

The magic of closure happens when we call `result()`. Even though `parentFunction` has already finished executing, `childFunction` still has access to `myValue` and `x`. We can increment these variables each time we call `result()`, demonstrating the power of closure.

It is important to note that `myValue` becomes a private variable, only accessible within `childFunction`. If we try to access `myValue` from the global scope, we will encounter a reference error.

Practical Closure Examples

Let`s explore some additional practical examples of closures.

Example 1: Counters

Now let`s consider the following function:

```
const incrementByFive = () => {
  // We created a private variable
  let privateValue = 0;

  // Here we create closure, so each time that the this returned function will run, it will increment by 5.
  const addFive = () => {
    // Here we assign a dynamic outcome of the mathematical action
    return privateValue = privateValue + 5;
  };

  return addFive;
};

const increment = incrementByFive();

console.log(increment()); // 5
console.log(increment()); // 10
console.log(increment()); // 15
console.log(increment()); // 20
console.log(increment()); // 25
```

In this example, we are receiving the `addFive` function, that was returned by the `incrementByFive` function, and now `increment` is function that has a private value, that will always increment by 5.

Let's look at another counter example:

```
const privateCounter = (() => {
  let count = 0;

  return {
    increment: function() {
      // But here we return NOT AN ASSIGNMENT - WE RETURN A RESULT OF THE
ACTION OF 'COUNT += 1'
      return count += 1;
    },
  };
})();

console.log(privateCounter.increment()); // Output: 1
console.log(privateCounter.increment()); // Output: 2
console.log(privateCounter.increment()); // Output: 3
console.log(privateCounter.increment()); // Output: 4
```

The code above establishes a counter with a private value (`count`) that is not directly accessible outside the function. `privateCounter` is assigned to the object returned from the Immediately Invoked Function Expression (IIFE), that has an `increment` method, which increases `count` by 1 each time it's called and then returns the new count.

Despite the fact that the surrounding function having finished execution, `increment` can still interact with `count` due to the closure created.

Example 2: 'addTo' Function

In the following example, Inside the `addTo` function, we initialize an inner function called `add` , that receives an argument called `inner` . The inner function, `add` will now return the result of `outer` plus `inner` .

The `add` function takes the `outer` variable and it adds it to `inner` variable and then returns it. We are returning the result of `outer` plus `inner` , not an assignment. In the end we just returning the `add` function. Notice that we are NOT calling it.

```
const addTo = (outer) => {
  const add = (inner) => {
    // Here we return NOT AN ASSIGNMENT - WE RETURN A RESULT OF THE
ACTION 'OUTER + INNER'
    return outer + inner;
  };

  return add;
};
```

The value of running `addTo(3)` will be the function `add` that we returned from the function `addTo`, without invoking it, and now it will have a **closure with the value of 3**.

```
const addThree = addTo(3);

console.log(addThree(10)); // 13
```

Here, we called the function we initialized `addThree` because now, with the closure, it has the variable `outer` with the value of `3` **even after the parent function has closed**. Now this function will **always** return the number we pass to it plus `3`, **and this is closure**.

Example 3: Creating a function factory

A factory function is a function in JavaScript that returns a new object or function when called, effectively "manufacturing" new instances as required. This pattern is useful for creating objects or functions with similar characteristics or behaviors, but with some degree of customization.

The following `powerOf` function, is a factory function that creates and returns new functions that calculate the nth power of a given number. Each function it creates is tailored with a specific exponent, `n`, due to the lexical scoping of JavaScript. This allows to create a suite of similar functions (like `square`, `cube`, etc.) with ease.

```
// This is our factory function, powerOf. It takes one argument 'n',
which represents the power.
function powerOf(n) {
  // The factory function returns a new function.
  // This function takes one argument 'x', which is the base of the
exponentiation operation.
  return function (x) {
    // The returned function calculates 'x' to the power of 'n' using the
built-in JavaScript function Math.pow().
    // Then it returns this result.
    return Math.pow(x, n);
  };
}


const square = powerOf(2);
console.log(square(5)); // Output: 25, because 5^2 = 25

const powerOfThree = powerOf(3);
console.log(powerOfThree(2)); // Output: 8, because 2^3 = 8

const powerOfFour = powerOf(4);
console.log(powerOfFour(2)); // Output: 16, because 2^4 = 16
```

Example 4: Credits Game

```
const game = (() => {
    let credits = 10;

    return () => {
        if (credits > 0) {
            credits -= 1;
            console.log(`You played the game. Remaining credits:
${credits}`);
        } else {
            console.log("Sorry, you have no credits left.");
        }
    }
})();

game(); // Outputs: You played the game. Remaining credits: 9
game(); // Outputs: You played the game. Remaining credits: 8
game(); // Outputs: You played the game. Remaining credits: 7
// ... continues until credits are exhausted
```

This example demonstrates a game where each play costs 1 credit. The credit balance is maintained privately within the game function, and it is only accessible through the returned function. Each time we call `game()`, it decrements the credits and logs the remaining balance. Once the credits reach zero, we are informed that no credits are left.

This is an excellent example of how closures can encapsulate and maintain private state, leading to cleaner and safer code by preventing unauthorized access or modifications to specific variables.

Real-World Examples of the use of closures

Let's take a look at a few real-world examples of where closures might come in handy.

1. Memoization

Memoization is a programming technique used to optimize computer programs by storing the results of expensive function calls and reusing them when the same inputs occur again. This technique uses closure in JavaScript:

```
// The memoize function is a higher-order function that takes a function
(fn) as an argument.
function memoize(fn) {
  // It then defines a "cache" object that will be used to store the
results of previous function calls.
  const cache = {};

  // The memoize function returns a new function. This inner function has
access to the "cache" object thanks to closure.
  return function (...args) {
    // It creates a unique "key" for every unique set of arguments that
the function is called with by stringifying the arguments.
    const key = JSON.stringify(args);
    // If this function has been called before with these exact
arguments, then the result will be in the cache.
    if (key in cache) {
      // In this case, we just return the cached result.
      return cache[key];
    } else {
      // If not, we call the original function with the provided
arguments and store the result in our cache.
      let val = fn(...args);
      cache[key] = val;
      // Finally, we return the result.
      return val;
    }
  };
}


// An expensive function (just for demonstration)
function factorial(n) {
  if (n === 0) {
    return 1;
  }
  return n * factorial(n - 1);
}

const memoFactorial = memoize(factorial);

console.log(memoFactorial(5)); // 120
console.log(memoFactorial(5)); // 120 (from cache)
```

This `memoize` function utilizes a closure to persist data (the `cache` object) across multiple calls. Even after `memoize` function has returned the inner function, the inner function can still access `cache` due to the closure. This `cache` stores the results of previous computations to avoid re-computation and enhance performance.

2. Module Pattern

In JavaScript, we can use closures to create private variables or functions, which is often referred to as the Module Pattern:

```
const bankAccountModule = (() => {
  let balance = 0; // private variable

  const getBalance = () => balance;
  const deposit = (amount) => { balance += amount; };
  const withdraw = (amount) => { balance -= amount; };

  return { getBalance, deposit, withdraw };
})();

bankAccountModule.deposit(100);
bankAccountModule.withdraw(20);
console.log(bankAccountModule.getBalance()); // 80
```

Here, `balance` is a private variable that cannot be modified directly from outside the `bankAccountModule`. The only way to interact with `balance` is through the functions `getBalance`, `deposit`, and `withdraw`, which have access to `balance` due to closure.

Conclusion

Understanding closures in JavaScript is crucial as they offer great power and flexibility when dealing with JavaScript functions. From preserving the state and making variables private to performing memoization and creating functional factories, closures are an essential feature of the JavaScript language that each developer should master.

This article explained closures and their relationship with lexical scope, provided practical examples of closure usage, and showcased real-world scenarios where closures play a significant role.

While closures might seem daunting at first, with practice and by studying examples, you'll find them easier to understand and use in your coding. Remember that a closure gives a function access to the variables from an outer function scope, even after that function has finished executing. This unique feature opens up many opportunities for efficient and effective coding practices in JavaScript.

Keep practicing and exploring this concept, and soon, you'll find yourself using closures effortlessly in your daily coding tasks, leveraging their power to write clean, efficient, and secure JavaScript code.