Calculating Closures

What is a *closure*? This question is one of the *most frequent interview questions*where JavaScript is involved. If you answer this question quickly and knowledgeably you'll look like a great candidate. We know you want to know it all so let's dive right in!

The official definition of a closure from MDN is, "A closure is the combination of a function and the lexical environment within which that function was declared." The practicality of how a *closure*is used it simple: a *closure*is when an inner function uses, or changes, variables in an outer function. Closures in JavaScript are incredibly important in terms of the creativity, flexibility and security of your code.

When you finish this reading you should be able to implement a closure and explain how that closure effects scope.

Closures and scope

Let's look at an example of a simple closure below:

```
function climbTree(treeType) {
  let treeString = "You climbed a ";

  function sayClimbTree() {
    // this inner function has access to the variables in the outer scope
    // in which is was defined - including any defined parameters
    return treeString + treeType;
  }

  return sayClimbTree();
}

// We assign the result to a variable
const sayFunction = climbTree("Pine");

// So we can call it, and indeed the variables have been saved in the closure
// and the sayFunction prints out their values.
console.log(sayFunction); // You climbed a Pine
```

In the above snippet the `sayClimbTree`function captures and uses the`treeString`and `treeType`variables within its own inner scope.

Let's go over some basic closure rules:

1. Closures have access to any variables within its own, as well as any outer function's, scope when they are declared. This is where the *lexical environment*comes in - the *lexical environment*consists of any variables available within the scope in which the closure was declared (which are the local inner scope, outer function's scope, and global scope).
2. A closure will keep reference to all the variables when it was defined **even if the outer function has returned**.

Notice above that even though the above `climbTree`had run its `return`statement the inner function of `sayClimbTree`**still has access**to the variables(`treeString`and `treeType`) from the outer scope where it was declared. So, even after an outer function has returned, an inner function will still have access to the outer function's variables.

Let's look at another example of a closure:

```
function makeAdder(x) {
  return function(y) {
    return x + y;
  };
}

const add5 = makeAdder(5);

console.log(add5(2)); // prints 7
```

In the above example the function the anonymous function within the `makeAdder` function **closes over** the `x` variable and utilizes it within the inner anonymous function. This allows us to do some pretty cool stuff like creating the `add5` function above. Closures are your friend ♡.

Applications of closures

Let's take a look at some of the common and practical applications of closures in JavaScript.

Private State

Information hiding is incredibly important in the world of software engineering. JavaScript as a language does not have a way of declaring a function as exclusively private, as can be done in other programming languages. We can however, use *closures* to create private state within a function.

The following code illustrates how to use *closures* to define functions that can emulate private functions and variables:

```
function createCounter() {
  let count = 0;

  return function() {
    count++;
    return count;
  };
}

let counter = createCounter();
console.log(counter()); // => 1
console.log(counter()); // => 2

//we cannot reach the count variable!
counter.count; // undefined
let counter2 = createCounter();
console.log(counter2()); // => 1
```

In the above code we are storing the anonymous inner function inside the `createCounter` function onto the variable `counter`. The `counter` variable is now a *closure*. The `counter` variable **closes over** the inner `count` value inside `createCounter` even after `createCounter` has returned.

By **closing over** (or **capturing**) the `count` variable, each function that is return from `createCounter` has a **private**, mutable state that cannot be accessed externally. There is no way any outside function beside the closure itself can access the `count` state.

Passing Arguments Implicitly

We can use closures to pass down arguments to helper functions without explicitly passing them into that helper function.

```
function isPalindrome(string) {
  function reverse() {
    return string
      .split("")
      .reverse()
      .join("");
  }

  return string === reverse();
}
```

What you learned

How to implement a closure and explain how that closure effects scope.