Review of Scope

Before talking about closures, we need to review scope. Scope is a topic we covered in a lot of detail in the Learn Modern JavaScript: Getting Started course. So we simply want to review it here.

Definition

Scope can be defined as a set of rules that determine where identifiers (variables, functions etc) are accessible within your code.

JavaScript uses function-based scope. This means that the scope of identifiers is the function in which they are defined. So a variable defined in a function named *test* is only accessible by code that resides inside function *test*.

If a variable or function is defined outside of a function, it is part of the global scope. Globally scoped identifiers are accessible by all code that exists in your program. Using global scope for any identifiers that you need access to throughout your program is a common practice for beginning JavaScript variables, but the challenge is it muddies the global space and can lead to problems. As such globally scoped identifiers should be avoided.

Practical Application

To expand on this, lets take a look at some sample code:

```
var num1 = 1,
    num2 = 5;

function test() {
    var num3 = 10;
    console.log(num1, num2, num3, num4);
    function test2() {
        var num4 = 15;
        console.log(num1, num2, num3, num4);
    }
    test2();
}

test();
console.log(num1, num2, num3, num4);
```

This code contains three console.log statements. However, the only one that will execute without an error is the second console.log statement that is inside the test2 function. It is the only console.log statement that has access to all those variables. It finds variable num4 within its own scope; the test2 function scope. It finds num3 in its parent's scope; the test function scope. And it can access num1 and num2 from the global scope.

When an identifier is encountered (be it variable or function), the JavaScript engine looks first in the enclosing functions scope for that identifier. If it cannot find it, it continues looking up the

scope chain until it comes to the global scope. If it is not found in the global scope, an error is generated.

For the first console.log statement, it cannot access num4 because it is not in its scope. The scope for function *test* does not include num4. The scope of function test includes test2, but not anything that is defined inside that function. When it doesn't find num4 inside its own scope, it moves up the scope chain. Its parent scope happens to be the global scope. When it doesn't find it there, it generates an error.

The third console.log statement doesn't have access to num3 or num4 because they are scoped inside other functions. The global scope has access to the test function, that is why it can invoke it, but not anything defined inside that function. Therefore, we could not invoke test2 from the global scope.

Rewriting this code without errors would look like this:

```
var num1 = 1,
    num2 = 5;

function test() {
    var num3 = 10;
    console.log(num1, num2, num3);
    function test2() {
        var num4 = 15;
        console.log(num1, num2, num3, num4);
    }
    test2();
}

test();
console.log(num1, num2);
```

If you would like to further review the concept of scope in JavaScript, you can view the following YouTube video that is part of my channel.

https://youtu.be/SBjf9-WpLac

Block Scope in JavaScript

One exception to functional scope is when variables are declared with the keyword let. let creates block scope. A block is defined by a set of curly braces. Lets take a look at another example:

```
function test() {
    let num1 = 1;
    if (num1 === 1) {
        let num2 = 10;
        console.log(num1, num2);
    }
    console.log(num1, num2);
}
```

```
test();
```

The second console.log statement will generate an error. It tries to access num2, but num2 is out of scope. The scope for num2 is the block created by the if statement. Since the second console.log statement is outside that scope, it cannot access it and generates an error.

The scope for num1 is the block created by function test.

The let keyword is a part of the ES6 standard. Using let with a for loop is ideal, because in most cases the variables defined as a part of a for loop is only used as a part of the block that is created with the curly braces used to define the loop.