Lesson Introduction

Closures can be behind some of the surprising behaviors in JavaScript that make people a little bit leery about it.The thing to remember about closures, is that closures created in the same scope all share the same data. And the upshot of that is that closures will reflect the current value of the data that they share. Not necessarily the value that might appear to be appropriate at the time of creation.

Creating Variables

Let me show you what I mean.

```
var counter;
var functions = [];
var values = [];
for(counter = 0; counter < 5; counter++){
    values[counter] = counter;
    functions[counter] = function(){
        return counter;
    }
}
console.log(values[0]); // 0
console.log(values[2]); // 2
console.log(values[4]); // 4
console.log(functions[0]()); // 5
console.log(functions[2]()); // 5
```

Even though `values[counter]` is being set right before `functions[counter]` and, theoretically, they should have the same value for `counter`, each one of these functions is actually returning the final value that comes at the end of running through the entire loop. The reason for that is that all of these functions share common data. Because of that, they're returning the current value of the data that they have, not the value that it was at at the time that they were created.

So the trick when defining functions in a loop is, first of all, don't define functions in a loop. But if you're going to and you're using ECMAScript 5 or earlier, remember that loops cannot have their own scope. So what you want to do is use a constructor function.

Constructor functions allow you to create a new independent scope for each closure that you're creating.So starting again with our counter, our functions, and our values arrays.

```
var counter;
var functions = [];
var values = [];
var makeReturner = function() {
    return function() {
        return value;
    }
};

for(counter = 0; counter < 5; counter++){
    values[counter] = counter;
    functions[counter] = makeReturner(counter);
}

console.log(functions[0]()); // 0
console.log(functions[2]()); // 2
```

So now everything works as expected.

Let for Loops in ECMAScript 6

Either avoid creating functions in a loop o ruse a constructor function in order to make sure that the values are not trapped in a closure that would have them all share the same value. Happily with ECMAScript 6 and later we have a new option. ECMAScript 6 uses the keyword `let` to create a temporary scope inside of `for` loops.

```
var functions = [];
var values = [];
for(let counter = 0; counter < 5; counter++){
    values[counter] = counter;
    functions[counter] = function(){
        return counter;
    }
}
console.log(values[0]); // 0
console.log(values[2]); // 2
console.log(values[4]); // 4
console.log(functions[0]()); // 0
console.log(functions[2]()); // 2
```

In this case we didn't predefine our `counter` variable because we're going to use the keyword `let`.