Closure

1. This is a topic on which I have noted quite a bit of confusion. Even at the most basic level, I have heard closure defined in many different ways, even as several different parts of speech (i.e., as a noun, a verb, an adjective). I believe that much of the confusion stems from a misplaced apprehension that closure is a difficult and mysterious process that can be understood only with a mighty effort. The good news is that it is not so difficult to understand, although it can produce results that are counterintuitive until one has gotten a good handle on it.
2. To illustrate the operation of closure in Javascript, I will set out a simple coding exercise. The goal of our code will be to get a countdown along these lines:

```
5 and counting!
4 and counting!
3 and counting!
2 and counting!
1 and counting!
Bazoom!
```

3. Obviously, anyone who has gotten past his or her first lesson or two in Javascript will think of using a for-loop to get this output, perhaps as follows:

```
for (var i = 0; i < 5; i++) {
    var counter = 5;
    console.log (counter + " and counting!");
    counter--;
}
console.log("Bazoom!");
```

4. Which, of course, leads to the following result:

```
5 and counting!
5 and counting!
5 and counting!
5 and counting!
5 and counting!
Bazoom!
```

because the counter variable is inside our for-loop. Placing it outside the for-loop will

allow the counter to avoid being reset to 5 each time the loop runs, so code that works might look something like:

```
var counter = 5;
for (var i = 0; i < 5; i++) {
    console.log (counter + " and counting!");
    counter--;
}
console.log("Bazoom!");
```

producing the following output:

```
5 and counting!
4 and counting!
3 and counting!
2 and counting!
1 and counting!
Bazoom!
```

5. Okay, that is a solution. However, taking a more sophisticated approach to the problem, we can remove the repeated functionality and compartmentalize it in a function (of course, in such a simple example, one might decide "why bother", but this is something we will want to do in a more typical case). Doing so, we might end up with:

```
var counter = 5;
function countDown() {
    console.log(counter + " and counting!");
    counter--;
}

for (var i = 0; i < 5; i++) {
    countDown();
}
console.log("Bazoom!");
```

6. In the above example, the counter variable must be declared and set outside the countDown() function, or else it would get reset each time, just as in the simple looping example with which we began. That gives rise to a problem of scope - if this is part of a large program, with several programmers working on its various components over a period of years, then there is a pretty good chance that somebody else may try using the name "counter" for his own variable, in which case there would be a conflict between

these multiple uses of a global variable.

7. This problem is solved by a **closure**. A **closure** is simply a function, together with the variables that are in its environment. So, consider the following, which gets the counter variable out of the global scope and into that of the wrapper() function (you may also note that we change the countDown() function to return a value, rather than log it to the console, but that is beside the point):

```
function wrapper() {
    var counter = 5;
    function countDown() {
        result = counter + " and counting!");
        counter--;
        return (res);
    }
    return countDown;
}
for (var i = 0; i < 5; i++) {
    console.log(wrapper());
}
console.log("Bazoom");
```

8. Now, this does not work, either. What logs to the console is:

```
 [Function: countDown]
 [Function: countDown]
 [Function: countDown]
 [Function: countDown]
 [Function: countDown]
 Bazoom!
```

9. So, as we should expect if we look at the return of the wrapper() function, we are getting five instances of the actual countDown function itself - not implementations of the function; rather, wrapper() is simply passing the function itself to the console. This should be a familiar concept: that in Javascript functions are first-class objects, and therefore can be passed around just like integers or arrays, etc. In contrast, if the return value of the wrapper() function were *countDown()*, then countDown() would be evaluated each time wrapper() was called in the for loop, which would leave us with a counter that resets to 5 each time around).

Okay, Here is the Important Stuff!!!

1. Let's assume that in the above example we then assign the value of the wrapper() function to a variable "x". In other words :

```
var x = wrapper();
```

2. We know that the return value of wrapper() is countDown. Let's examine a few possible calls:

```
console.log(countDown());
```

The above would simply generate a "Reference Error", because countDown, being inside the wrapper() function, is not within our current scope. Instead, we might try:

```
console.log(wrapper()());  // results in "5 and counting"
```

This seems like progress. So, what if we now construct our program as follows:

```
function wrapper() {
    var counter = 5;
    function countDown() {
        result = counter + " and counting!");
        counter--;
        return (result);
    }
    return countDown
}
for (var i = 0; i < 5; i++) {
    console.log(wrapper()());
}
console.log("Bazoom");
```

What we end up with is:

```
5 and counting!
5 and counting!
5 and counting!
5 and counting!
5 and counting!
Bazoom!
```

This should not be a shock. Each time it is called in the for-loop, wrapper() returns countDown which, being followed by parentheses, gets evaluated anew, resulting in a fresh setting of counter to 5.

Hang On, Here is the Magic Sleight-of-Hand

1. Following this, we might expect a similar result from the following, being the exact copy of the code above, except that we assign, to a variable x, the value "wrapper()". So every time the for loop runs, we run *console.log(x())*; instead of *console.log(wrapper()())*;

```
function wrapper() {
    var counter = 5;
    function countDown() {
        result = counter + " and counting!");
        counter--;
        return (result);
    }
    return countDown
}

var x = wrapper();

for (var i = 0; i < 5; i++) {
    console.log(x());
}
console.log("Bazoom");
```

However, the resulting output to the console differs:

```
5 and counting!
4 and counting!
3 and counting!
2 and counting!
1 and counting!
Bazoom!
```

Bazoom! for sure!!! - this is the output we are looking for, but why did it happen, when the counter got reset each loop in the previous example? **The answer lies in the fact that, behind the scenes, when a function is assigned, it is accompanied by the values of the variables in its scope, as they exist *at the time the function is assigned***. So, in the first example, "wrapper()" kept evaluating to countDown, and getting a nice, fresh set of scope variables (including counter as 5) each time. In the latter example, x was

assigned the function countDown, together with the accompanying variables, but did not receive new sets by assignment again. After being assigned the value of countDown, when x() was called, it was evaluated, counter was decremented, and then next time x() was called, counter was decremented again, and so forth.

Detour

1. This is not necessary to consider immediately, while getting the other points set in your mind, but reflect upon what happens when we make an assignment, "x = y". There are two possible interpretations, when we remember that a variable is a piece of memory somewhere in the computer, in which a value is held. Think of the memory as a sheet of paper onto which a number is written.
    a. If I have the paper, with a 5 written on it, and you wanted access to the information, I could copy the sheet and give you your own copy. If I erased my 5 and wrote a 6 on it, yours would still say 5, and vice versa.
    b. Alternatively, I could give you a key to a safe deposit box where I keep the paper. You can go see it anytime, and if either you or I scratches out the 5 and writes a 6, then both of us will now have a 6 in our paper memory.

    Depending on the situation, either thing could happen in Javascript. Compare the following two snippets:

```
var y = "tail";

var x = y;

y = "tale";

console.log(x) => results in "tail" (i.e., x and y are independent)
```

    with

```
var y = [1, 2, 3, 4];

var x = y;

y[0] = "Bazoom";

console.log(x) => results in [Bazoom, 2, 3, 4] (i.e., x and y are linked)
```

    For present purposes the point is this: **When a function is assigned to a variable, a new copy is assigned to the variable, which is independent of any other copies.**

Keep this in mind below.

Back to Where We Were

1. Note that what counts is when countDown is **assigned**, which creates a new set of scope variables. So, for example, if we assign countDown to a new variable countDown1, and then assign countDown to another variable countDown2, we have two separate and independant countdowns! Examine the following:

```
function wrapper() {
    var counter = 5;
    function countDown() {
        result = counter + " and counting!");
        counter--;
        return (result);
    }
    return countDown
}

var countDown1 = wrapper();
var countDown2 = wrapper();

console.log(countDown1());  => 5 . . .
console.log(countDown1());  => 4 . . .
console.log(countDown2());  => 5 . . .
console.log(countDown1());  => 3 . . .
console.log(countDown1());  => 2 . . .
console.log(countDown2());  => 4 . . .
```

2. Finally, note that with use of a closure, i.e., the countDown function and its scope variables or referencing environment, we have completely removed our counter from the global variable scope, so that even if 100 other coders declare variables named "counter", it won't be our problem because we will be inside our function scope.