Functions as Closures

Relevant Links

- Flanagan's book, section 8.6

Notes

- Javascript, like most modern programming languages, employs what is known as **lexical scope**:

  > In *lexical scope* the local variables that a function has access to are determined by where the function was *defined*, rather than where it was *used*.

  Here's a code example:

```
var c = 2;
var v = 10;
function storeMe(v) {
   var c = 5;
   return function printStuff() {
      console.log("In f's call, v is:", v);
      console.log("In f's call, c is:", c);
   }
}
var f = storeMe(20);
f();
console.log("Out here, v is:", v);
console.log("Out here, c is:", c);
```

  Let us see what is going on:

  - First of all, the two variables `c` and `v` are defined, with values 2 and 10.
  - Then a function `storeMe` is defined, and later on is called with value 20.
  - When that call is executed, a new scope is created, on which `v` has the passed value 20, and after the first line is executed a variable `c` is defined with value 5.
  - We then return a function. This function will need a `v` and a `c` when it is called.
  - These are determined by what values they have *where* the function was defined, namely inside the execution of `storeMe`.

- So when `storeMe(20)` returns a function, and we store it at the variable named `f`, we in fact store more than just a function. We store the "environment", the "scope", in which that function was defined.

- This is called a "function closure", or just "closure" for short. It is the basis for some extremely interesting patterns that we will study in the days to come.

- The most standard example of this idea is a counter function:

```
function makeCounter() {
   var c = 0;
   return function count() {
      c += 1;
      return c;
   }
}
var c1 = makeCounter();
var c2 = makeCounter();
c1();
c1();
c2();
```

  Call c1 and c2 a couple of times and notice their behavior, before we discuss it.

  - Let's go through what goes on when makeCounter is called. First of all, a variable c is created.
  - Then a function count is returned. That function will have access to all the variables available where it was defined, in particular to the variable c.
  - When that function count is called, it first increments that variable c. It then returns the new value. Every time it is called, it will increment once more.
  - If makeCounter is called a second time, to create c2, then a new different scope is created, in which a variable c is declared, completely different from the one used in c1.
  - In fact, noone else has access to that variable c belonging to the function c1. The counter function c1 can count (if you pardon the pun) on the fact that noone can mess with its counter variable c. We have effectively created a "private variable".

- As a final and more extended example, here is an implementation of a stack. Of course we could use an array instead, but arrays have many more methods. We want to provide a system, so that users should only be able to do the following, and nothing else:

  - Create a new empty stack
  - Push a new element at the top of the stack
  - Pop and return the element at the top of the stack
  - query whether the stack is empty

  Here is a way to implement this:

```
function makeStack() {
    var arr = [];
    return {
        push: function(el) { arr.push(el); return null; },
        pop: function() {
            if (arr.length === 0) {
                throw new Error("Attempt to pop from empty stack");
            } else {
                return arr.pop();
```

```
            }
        },
        isEmpty: function() { return arr.length === 0; }
    };
}
var s1 = makeStack();
var s2 = makeStack();
s1.push(1); s1.push(2); s1.push(3);
while (!s1.isEmpty()) { s2.push(s1.pop()); }
```

When the function makeStack is called, it creates a new array empty, arr, to hold the values. But it keeps that array secret. Instead, it returns an object with three properties, whose values implement the three operations we require of a stack.

This is important. We only expose to the user the functions they are supposed to call, and hide everything else. This way we can me sure our structure is used in a predictable way.