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 d = 3;
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);
      console.log("In f's call, d is:", d);
   }
}
var f = storeMe(20);
f();
console.log("Out here, v is:", v);
console.log("Out here, c is:", c);
console.log("Out here, d is:", d);
```

  Let us see what is going on:

  - First of all, the three variables `c`, `d` and `v` are defined, with values 2, 3 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. The `d` that was defined outside the function is still visible.
  - We then return a function. This function will need a `v`, a `c` and a `d` 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.

  A **function closure** is a function together with the scope/environment in which the function was *defined*.

- The most standard example of this idea is a counter function: javascript function makeCounter()

  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".
  - Another important consequence is the following:

    Variables local to a function can survive for a long time past the function's execution, if they are part of the scope/environment of the function closures created within the function and returned with the return value.

- **Practice**:

  1. Write a version of makeCounter that takes an argument a, and uses that as a starting point for the increment. Make it so that if no argument is provided, it defaults to starting at 1.
  2. Write a version of makeCounter that takes up to two arguments, a and b. If b is not provided, it should behave like the previous example. If b (and a) is provided, it should start incrementing from a, but whenever the current count is b or more it should reset to a.
  3. In this problem the goal is to be able to keep track of how many counters we created. Write a function makeCounterMaker, which takes no input, and will return the makeCounter function.
     - It should have a variable that counts how many counters have been created so far.
     - Each time a new counter is created by calling the returned makeCounter function, this variable should be incremented.
     - Each created counter should also store in it its index/number, i.e. the first counter that was created would have index 1, the second should have index 2 etc.

  - When a counter is called, it should console log the message: "Counter #..., now at count ..." where the first number is the counter's index, and the second is the counter's current number.
  - Yes there is a 3-level nesting of functions: The makeCounterMaker function returns a function makeCounter which in turn when called makes a new counter which is itself a function that we call and increment.

```
// Example run
var maker = makeCounterMaker();
var c1 = maker(3);   // Counter #1, starting at 3
var c2 = maker();    // Counter #2, starting at 1
c1();        // Prints: "Counter #1, now at count 3"  and returns 3
c2();        // Prints: "Counter #2, now at count 1"  and returns 1
c1();        // Prints: "Counter #1, now at count 4"  and returns 4
c2();        // Prints: "Counter #2, now at count 2"  and returns 2
```

- 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 empty array, 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 be sure our structure is used in a predictable way.

The other important element is that noone from the outside world has direct access to the array arr. The array was created within the function makeStack, so the functions that were defined within makeStack can see it, but when makeStack returns this array is no longer visible (it was local to the function).

Practice

Implement a simple "single linked-list" structure, as follows:

- A makeList structure should be called when you want to create a new linked list. It should create a local head variable, and it should return an object consisting of the methods described below.
- Each new "node" will consist of an object with two properties: value, and next. next would be the next node (or null if we are at the end).
- You should implement the following methods:
  - prepend(value) adds a new node at the beginning of the list.
  - append(value) adds a new node at the end of the list.
  - first () returns the value at the beginning of the list.
  - last () returns the value at the end of the list.
  - length() returns the length of the list.
  - popFront() removes the node from the front of the list, and returns its value.
  - popEnd() removes the node at the end of the list, and returns its value.
  - toArray() returns a (possibly empty) array of the values in the list.