Duplicate Methods With Closures

One thing to remember about closures is that they occupy a certain amount of space in memory. Creating functions inside of other functions can actually lead to duplication in the memory. That can eventually slow down performance.

```
var actionTracker = function(choice) {
  var tracking = {};
  return {
    setAction: function(action) {
      if (action) {
        tracking[choice] = action;
      }
    },
    getAction: function() {
      return tracking[choice];
    }
  };
};

var redTracker = actionTracker("red");
var blueTracker = actionTracker("blue");
console.log(redTracker.getAction);
//function() { return tracking[choice]; }
console.log(blueTracker.getAction);
//function() { return tracking[choice]; }
```

Each of these is taking up space in memory and because each one has two methods, those methods are also taking up space.

Closures Not Freeing Memory

Another issue that comes up with closures is that they don't automatically free their memory. JavaScript has a garbage collection routine that will automatically delete any unused memory once all references it are gone. But the outer function scope is never going to be cleaned from memory as long as it's still being used by some existing closures.

```
var actionTracker = function(choice) {
  var tracking = {};
  return {
    setAction: function(action) {
      if (action) {
        tracking[choice] = action;
      }
    },
    getAction: function() {
      return tracking[choice];
    }
  };
};

var redTracker = actionTracker("red");
redTracker.setAction("click");
console.log(redTracker.getAction()); //"click"
redtracker = null;
```

The “click” is being stored in a closure that’s inside of `actionTracker` and that means that it is not going to be able to be released from memory, as long as `redTracker` still exists. So in your program if there comes a point, when you know that you’re done using an element inside of a closure, you can set it to `null`.

In this case we can do `redTracker = null;`. Running that doesn’t appear to do anything, but the result is that the next time that the JavaScript garbage collection loop comes around, it’ll recognize that memory location is no longer being used and it’ll be able to clean it up. That’ll avoid memory leaks in your programs and help your programs run more efficiently.