Functions as a means of creating scope

In this section we discuss how functions are used to create a new scope in which to hide information that we do not want to share with the world.

Relevant Links

- Flanagan's book, section 8.5

Notes

An integral part of programming is deciding what to share with whom and what to protect from others. A good example of this was our counter function that we saw earlier. We will look at a variant of that system here, around the idea of a "timer".

A "timer" has the following components

- A method to fire the timer.
- A method to observe the timer. This method is given a function to call when the timer "fires", and adds it to a list of such previously given functions. This way lots of parties can "subscribe" to be called when the timer fires.
- A count of how many times the timer has fired.

For now we will manually be firing the timers, but later on we will add a way to fire them at regular intervals.

Here is how the timer code skeleton might look like:

```
function makeTimer() {
    var times = 0;

    // Array holding functions waiting for the timer to fire
    var observers = [];

    function observe(f) {
        // ... Add f to the observers ...
    }
    function fire() {
        // Increment times
        // Notify all observers
    }
    function notify() {
        // private method that does the calling of the observers
    }
    // We only return the methods we want to expose:
    return {
        observe: observe,
        fire: fire
    };
}
```

So, within the function makeTimer, we define some local variables and some local functions. None of these methods are seen by anyone outside the function. But we can choose to share some of them by including them in the returned object. In this case, we keep notify private, and return the two methods that are part of the interface. All three methods have access to each other, as well as the two local variables. Here is how a full implementation might look like. Notice how fire calls notify.

```
function makeTimer() {
    var times = 0;
    var observers = [];

    function observe(f) {
        observers.push(f);
    }
    function fire() {
        times += 1;
        notify();
    }
    function notify() {
        for (var i = 0; i < observers.length; i += 1) {
            observers[i](times);
        }
    }
    return {
        observe: observe,
        fire: fire
    };
}
```

So far so good, a perfectly fine implementation. But let us take it to the next step:

We want a way to manage a single main list of timers. We should be able to:

- Add a new timer to the list, using makeTimer and storing the result.
- Have a method to fire all timers.
- We would also like to take the notify method out of makeTimer. Maybe we can have a single notify, that takes as input the observers list and the times value, and calls those observers with that value.

So let us think about this for a minute. we want to create a new object, let's call it timers, that has basically just two methods, one for a new timer and one to fire all timers. But we also need to store a lot of intermediate stuff, like the whole makeTimer function above, the list of timers, the shared notify function, and so on. These are all private. In order to protect them, we use a pattern called **immediate function invocation**. In this instance it looks like this:

```
var timers = (function() {
    // We put here all the private stuff
    ...
    return {
        // We return the stuff we want people to access
        ...
    };
}());     // And immediately invoke this function.
```

The idea is this:

- We use an anonymous function, and immediately after its declaration ends at the closing brace we call it.
- Any local variables and functions defined inside this anonymous function are private to the function and not visible from the outside.
- Whatever we return from the anonymous function can be captured as the return value of the immediate invocation, and forms the public part of our object.

Let us look at the full implementation:

```
var timers = (function() {
    var allTimers = [];

    // private method
    function notify(observers, times) {
        for (var i = 0; i < observers.length; i += 1) {
            observers[i](times);
        }
    }

    // private method
    function makeTimer() {
        var times = 0;
        var observers = [];

        function observe(f) {
            observers.push(f);
        }
        function fire() {
            times += 1;
            notify(observers, times);
        }
        return {
            observe: observe,
            fire: fire
        };
    }

    function addTimer() {
        var newTimer = makeTimer();
        allTimers.push(newTimer);
        return newTimer;
    }

    function fireAll() {
        for (var i = 0; i < allTimers.length; i += 1) {
            allTimers[i].fire();
        }
    }

    // We expose the two methods we want to share
    return {
        make: addTimer,
        fireAll: fireAll
```

```
    };
}());
```

Here is a sample run, make sure you understand what is going on.

```
var t1 = timers.make();
var t2 = timers.make();
function log(i) { console.log(i); }
t1.observe(log);
t2.observe(log);
t1.observe(log);
t1.fire();
t1.fire();
t1.fire();
t2.fire();
timers.fireAll();
```

There is a lot going on in the above code, so take some time digesting it all.

Later on we will discuss modules and make some of these ideas more precise.

Practice

Create a similar object (via immediate function invocation) called "population" that is meant to manage a list of all our people.

1. It should contain in its private information an allPeople variable, which will be an object/dictionary whose keys will be people's names, and whose values are going to be the person objects that the function newPerson returns.
2. Your population object should provide a "newPerson(name)" function that adds a new "person", with age 0. This "person" will contain two private variables, the name and age. It should export a getName() method and a getAge() method. It should only do so if a person with that name does not exist yet. It should return the newly created person, or else return the existing person.
3. The objects created by newPerson should also have two more methods: incrAge which increments the age by 1 (and returns the new age) and retire which removes the person from the list of people.
4. Your population object should also provide a getPerson(name) function that attempts to recover the person with a given name, and returns null if there is no such person.
5. Your population object should also provide a incrAll function that increments the ages of all persons by 1 (ideally by calling incrAge on all of them).
6. Your population object should also provide a printAll function that prints one at a time the name plus age of each person. Bonus points for making it so the ages all line up (i.e. the correct number of spaces is added to the names, to make the ages line up).