JavaScript - Defining Methods for Classes :: Efficiency vs Scope

Abstract:

The JavaScript language specification allows for different ways of defining Classes and its methods, each having its own pros and cons. This document captures the observations gathered as part of an exercise to find an *efficient approach* in defining methods (operations) for a Class. The required access scope of a method (private, privileged, public etc.) also has a role in deciding on a specific approach. However, in this analysis the importance is given for the efficiency over the access scope of method, considering that JavaScript language in itself does not provide scope specification unlike other scripting/programming languages. The efficiency is measured primarily in terms of the memory consumption upholding the basic essence of JavaScript.

Approach:

The following approaches were considered for the analysis:

1. Defining instance Methods within the Class Constructor
2. Defining prototype level Methods within the Class Constructor
3. Defining prototype level Methods outside the Class Constructor [**Recommended Approach**]
4. Defining Class and Methods within an IIFE (Immediately Invoked Function Expression) to scope its members inside a Closure

Plot (/Subject):

A Class named '*MyClass*' having two methods '*myAction()*' and '*getActionCount()*' is to be defined. The '*MyClass*' has a instance variable ('actionCount') to track the number actions invoked on that instance. The class wishes (not mandatory) to keep the instance variable scope as *private* as far as possible. The analysis is to find the balance between memory efficiency and the fulfillment of the access scope of variables.

Defining instance Methods within the Class Constructor

In this approach the methods are defined within the Class's constructor using the 'this' operator, as seen below -

```
function MyClass() {
            var actionCount = 0;

            this.myAction = function () {
                actionCount++;
            };

            this.getActionCount = function () {
                return actionCount;
            };
        }
```

Complete source:

Expand source

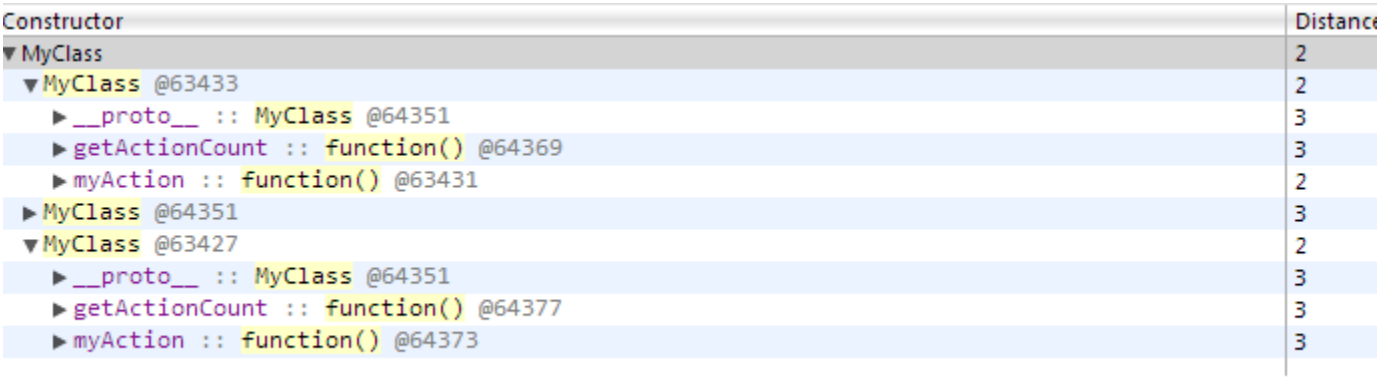
```
<html>
<head>
    <script>
        var globalMyClassMyAction;

        function MyClass() {
            var actionCount = 0;

            this.myAction = function () {
                actionCount++;
            };

            this.getActionCount = function () {
                return actionCount;
            };

            if (globalMyClassMyAction == undefined) {
                globalMyClassMyAction = this.myAction;
            }
        }

        var obj = new MyClass();
        obj.myAction();

        var obj2 = new MyClass();
        obj2.myAction();

        window.console.log("---------------------------------------");
        window.console.log("obj.myAction == obj2.myAction");

        if (obj.myAction == obj2.myAction)
            window.console.log(" --- myAction is SAME instance");
        else
            window.console.log(" --- myAction is DIFF instance");

        window.console.log("---------------------------------------");
        window.console.log("globalMyClassMyAction == obj2.myAction");

        if (globalMyClassMyAction == obj2.myAction)
            window.console.log(" --- myAction is SAME instance");
        else
            window.console.log(" --- myAction is DIFF instance");

        window.console.log("---------------------------------------");
    </script>
</head>
<body>
 <p>Defining instance Methods within the Class Constructor</p>
</body>
</html>
```

Analysis

This approach wraps all the method definition within the constructor projecting an illusion of encapsulation to the developer. The approach also

scopes the instance variable '*actionCount*' within the constructor and hence is private to the class, accessible only to the instance methods by the blessings of *Closure*. However, we need to note that the constructor of '*MyClass*' will be executed every time a new instance of the class is created by means of the '*new*' operator. That is, each instance of '*MyClass*' will get a new copy of the method object, making the approach **memory inefficient** (as well as performance inefficient, as the method instantiation and assignment is done during every construction).

The following diagram, **a snapshot of the JavaScript Heap from Webkit's Inspector** (Inspector > Profile > Take Heap Snapshot), shows different instances of the method definitions when 2 instances of *'MyClass*' is created -



The '*MyClass*' has a hashcode of **@64351**, which is referenced by 2 instances (as __proto__). However, the method definitions for each such instances are different. The '*getActionCount()'* method for the 1st instance is **@64369** and the 2nd instance is **@64377**, showing two different representation in the memory. This indicates a duplication of method definitions in memory, created each time a new instance of the '*MyClass*' is instantiated.

For this example of 2 instance, the memory overhead (or performance overhead) may not be significant (36 bytes per method), but if we create many instances and if the method definitions are large enough to hold more logic/scope-variables, the memory usage (and performance overhead) could grow significantly high.

Console output of the complete source code:

```
--------------------------------------
obj.myAction == obj2.myAction
 --- myAction is DIFF instance
--------------------------------------
globalMyClassMyAction == obj2.myAction
 --- myAction is DIFF instance
--------------------------------------
```

In the subsequent section we will notice that how the same method instance will be referenced by both class instances.

Defining prototype level Methods within the Class Constructor

In this approach the methods are defined within the Class's constructor using the '*prototype*' object of the class, as seen below -

```
function MyClass() {
            var actionCount = 0;

            MyClass.prototype.myAction = function () {
                actionCount++;
            };

            MyClass.prototype.getActionCount = function () {
                return actionCount;
            };
        }
```

Complete source:

Expand source

```
<html>
<head>
    <title>Defining prototype level Methods within the Class Constructor</title>
    <script>
        var globalMyClassMyAction;

        function MyClass() {
            var actionCount = 0;

            MyClass.prototype.myAction = function () {
                actionCount++;
            };

            MyClass.prototype.getActionCount = function () {
                return actionCount;
            };

            if (globalMyClassMyAction == undefined)
                globalMyClassMyAction = MyClass.prototype.myAction;
        }

        var obj = new MyClass();
        obj.myAction();
        var obj2 = new MyClass();
        obj2.myAction();

        window.console.log("---------------------------------------");
        window.console.log("obj.myAction == obj2.myAction");

        if (obj.myAction == obj2.myAction)
            window.console.log(" --- myAction is SAME instance");
        else
            window.console.log(" --- myAction is DIFF instance");

        window.console.log("---------------------------------------");
        window.console.log("globalMyClassMyAction == obj2.myAction");

        if (globalMyClassMyAction == obj2.myAction)
            window.console.log(" --- myAction is SAME instance");
        else
            window.console.log(" --- myAction is DIFF instance");

        window.console.log("---------------------------------------");
    </script>
</head>
<body>
    <p>Defining prototype level Methods within the Class Constructor</p>
    <p>prototype method inside constructor will also allocate memory for the methods
in each instance, but the memory allocated for the
       previous instance will be garbage collected and it will point to the new
memory. Please see the console for the
       results.  Memory efficiency is low if we put the prototype inside the
constructor when compared to putting it outside the
       constructor.</p>
</body>
</html>
```

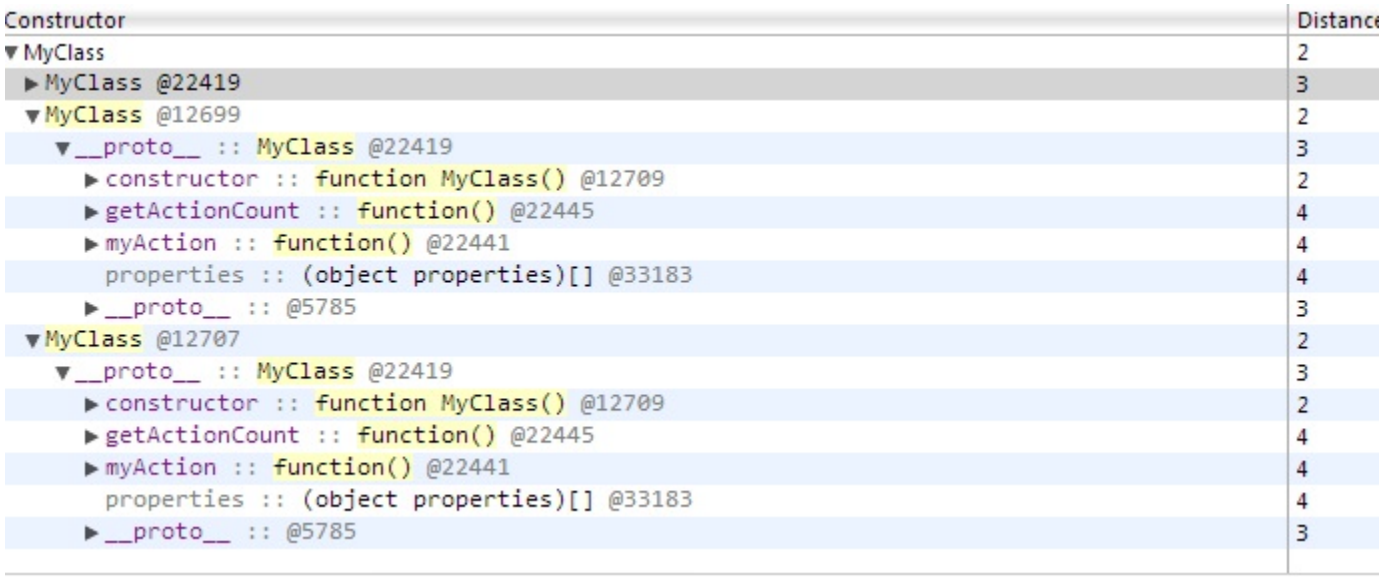
Analysis

This approach also gives an illusion of encapsulation to the developer by means of defining all the function definition within the constructor. Apart from that it also shares many other commonality with the earlier approach (1), including the support for the intended scope of variable '*actionCount*' within the constructor by means of *Closure*. The key difference being that the *prototype* of the class is used for assigning the method definition. The approach ensures that only ONE single copy of the method object existed at any given point in time. But that is not the same as saying - single method definition.

As the constructor of '*MyClass*' gets executed every time a new instance of the class gets created, new method object is allocated to be assigned as an instance method, overwriting the previous method definition/object. The main difference in comparison to the earlier approach (1) is that only the last method object (created during the last *MyClass* instance creation) is maintained and the previous method objects are de-referenced for garbage collection.

This approach is **as inefficient as the earlier approach** (1) from memory and performance perspective, as there is unnecessary allocation/de-allocation triggered in the system.

The following diagram, a snapshot of the JavaScript Heap, shows **the same instances of the method definitions** being used when 2 instances of *'MyClass*' is created -



In the above snapshot, the '*getActionCount()'* method for the 1st instance is **@22445** and the 2nd instance is also **@22445**, indicating that only one method instance is available in the memory.

Now we cannot rely only on the heap snapshot, so a quick change in the source code to remember the first instance's method allocation for later comparison with the second instance, clarifies the fact that the system has newly allocated for **@22445** when the second instance of '*MyClass*' was created.

The below output captured by executing the above *complete source code* helps in inspecting the correctness of the observation. *The variable* '*globalMyClassMyAction*' is holding the reference of the first allocation for '*getActionCount*' when the *Obj* was created -

Console output of the complete source code:

```
--------------------------------------
obj.myAction == obj2.myAction
 --- myAction is SAME instance
--------------------------------------
globalMyClassMyAction == obj2.myAction
 --- myAction is DIFF instance
--------------------------------------
```

Defining prototype level Methods outside the Class Constructor

In this approach the methods are defined outside the Class's constructor using the '*prototype*' object of the class, as seen below -

```
function MyClass() {
            this.actionCount = 0;
        }

        MyClass.prototype.myAction = function () {
            this.actionCount++;
        };

        MyClass.prototype.getActionCount = function () {
            return this.actionCount;
        };
```

Complete source:

Expand source

```
<html>
<head>
    <title>Defining prototype level Methods outside the Class Constructor</title>
    <script>
        var globalMyClassMyAction;

        function MyClass() {
            this.actionCount = 0;
        }

        MyClass.prototype.myAction = function () {
            this.actionCount++;
        };

        MyClass.prototype.getActionCount = function () {
            return this.actionCount;
        };

        if (globalMyClassMyAction == undefined)
            globalMyClassMyAction = MyClass.prototype.myAction;

        var obj = new MyClass();
        obj.myAction();
        var obj2 = new MyClass();
        obj2.myAction();

        window.console.log("---------------------------------------");
        window.console.log("obj.myAction == obj2.myAction");

        if (obj.myAction == obj2.myAction)
            window.console.log(" --- myAction is SAME instance");
        else
            window.console.log(" --- myAction is DIFF instance");

        window.console.log("---------------------------------------");
        window.console.log("globalMyClassMyAction == obj2.myAction");

        if (globalMyClassMyAction == obj2.myAction)
            window.console.log(" --- myAction is SAME instance");
        else
            window.console.log(" --- myAction is DIFF instance");

        window.console.log("---------------------------------------");
    </script>
</head>
<body>
    <p>Defining prototype level Methods outside the Class Constructor</p>
    <p>Prototype method defined outside the constructor will not allocate memory for
methods in each instance. Please see the console
    for the results. And this is best compared to any other methods in terms of memory
management. The drawback here is that
       we needs to declare everything global.</p>
</body>
</html>
```

Analysis

In this approach the methods are defined outside the constructor and is defined as part of the prototype object of the '*MyClass*'. If the methods are defined as named functions and then assigned to the prototype, those functions are invokable by just calling without the class and that will have undesired affect/result. In the above test an anonymous function definition is used to associate method to the prototype, which avoid the above concern.

The key difference in this approach is that the method definitions will be shared among all the instances of *MyClass*, as the definitions are part of the class's prototype object. Hence, this approach proves to be more memory efficient (and performance efficient too as method assignments, allocation are not repeated). The only limitation is that all the methods are Public from an access level perspective. However, that is a tradeoff for efficiency on JavaScript platform that inherently does not have support for access levels from OOPs perspective.

This is the approach recommended for defining EPG/Framework Classes and methods. The notion of private properties and methods can be brought in by means of an agreed naming convention, preferably names that start with an "_" character (which is mostly used in popular public domain JavaScript frameworks).

The following diagram, a snapshot of the JavaScript Heap, shows **the same instances of the method definitions** being used when 2 instances of '*MyClass*' is created -

In the snapshot, the '*getActionCount()*' method for the 1st instance is **@54399** and the 2nd instance is also **@54399**, indicating that only one method instance is available in the memory as expected. It can observed that the methods are now under the **__proto__** object of the *MyClass* in stance, and instance is pointing to the same **__proto__** (i.e. **@54397).**

Console output of the complete source code:

```
-------------------------------------
obj.myAction == obj2.myAction
 --- myAction is SAME instance
-------------------------------------
globalMyClassMyAction == obj2.myAction
 --- myAction is SAME instance
-------------------------------------
```

Defining Class and Methods within an IIFE (Immediately Invoked Function Expression) to scope its members inside a Closure

In this approach the methods are defined outside the Class's constructor using the '*prototype*' object of the class, but enclosed in a IIFE for attempting to bring private/privilege scope for function/property, as seen below -

```
var MyClass = (function() {

            var privilegedSharedVar = 'foo';

            function privilegedSharedFunction(data) {
                /* Has access to privateSharedVar
                   may also access publicSharedVar via MyObj.prototype as
                   a function (not as method of an instance)
                */
                privilegedSharedVar = data;
            }
            function MyClass() {
                this.actionCount = 0;
            }

            MyClass.prototype.updatePrivilegedVar = function (data) {
                privilegedSharedFunction(data);
            };

            MyClass.prototype.myAction = function () {
                this.actionCount++;
            };

            MyClass.prototype.getActionCount = function () {
                return this.actionCount;
            };

            MyClass.prototype.getPrivilegedVar = function () {
                return privilegedSharedVar;
            };

            return MyClass;
        })();
```

Complete source:

```
<html>
<head>
    <title>Defining Class and Methods within an IIFE (Immediately Invoked Function
Expression) to scope its members inside a Closure.</title>
    <script>
        var globalMyClassMyAction;

        var MyClass = (function() {

            var privilegedSharedVar = 'foo';

            function privilegedSharedFunction(data) {
                privilegedSharedVar = data;
            }

            function MyClass() {
```

› Expand source

```
        this.actionCount = 0;
    }

    MyClass.prototype.updatePrivilegedVar = function (data) {
        privilegedSharedFunction(data);
    };

    MyClass.prototype.myAction = function () {
        this.actionCount++;
    };

    MyClass.prototype.getActionCount = function () {
        return this.actionCount;
    };

    MyClass.prototype.getPrivilegedVar = function () {
        return privilegedSharedVar;
    };

    if (globalMyClassMyAction == undefined)
        globalMyClassMyAction = MyClass.prototype.myAction;

    return MyClass;
})();

var obj = new MyClass();
obj.myAction();

var obj2 = new MyClass();
obj2.myAction();

window.console.log("---------------------------------------");
window.console.log("obj.myAction == obj2.myAction");
if (obj.myAction == obj2.myAction)
    window.console.log(" --- myAction is SAME instance");
else
    window.console.log(" --- myAction is DIFF instance");
window.console.log("---------------------------------------");

window.console.log("globalMyClassMyAction == obj2.myAction");
if (globalMyClassMyAction == obj2.myAction)
    window.console.log(" --- myAction is SAME instance");
else
    window.console.log(" --- myAction is DIFF instance");
window.console.log("---------------------------------------");

window.console.log("----------------------");
window.console.log("obj.getPrivilegedVar() -> " + obj.getPrivilegedVar());
window.console.log("obj2.getPrivilegedVar() -> " + obj2.getPrivilegedVar());
window.console.log("----------------------");

window.console.log("obj.updatePrivilegedVar('hello')");
obj.updatePrivilegedVar('hello');

window.console.log("----------------------");
window.console.log("obj.getPrivilegedVar() -> " + obj.getPrivilegedVar());
window.console.log("obj2.getPrivilegedVar() -> " + obj2.getPrivilegedVar());
window.console.log("----------------------");
```

```
        window.console.log("----------------------------------------");
        /* Error (cant access) ->
                obj.privilegedSharedVar;
                obj.privilegedSharedFunction();
        */
    </script>
</head>
<body>
    <p>Defining Class and Methods within an IIFE (Immediately Invoked Function
Expression) to scope its members inside a Closure.</p>
    <p>As the private variables and methods are within the Closure of the IIFE, the
scope of the variable/methods are protected.  However, the private variables withing
the Closure scope would be shared among all the instances of the Class, making those
variables not an Instance member.  The approach fails to provide "private" scoped
```

```
instance state management, and hence does not serve the purpose.</p>
</body>
</html>
```

Analysis

In this approach '*MyClass*' definition is enclosed inside a self-invoked anonymous function (also known as IIFE - Immediately Invoked Function Expression) as an attempt to create a private/privileged scope for some variables/methods. As the privileged variable and method are within the Closure of the IIFE, the scope of the variable/methods are protected with exclusive access only within the 'MyClass' definition (functions within the IIFE). However, the private/privileged variable within the Closure scope would be shared among all the instances of the '*MyClass*', making those variables not an Instance member. The approach fails to provide "private" scoped instance state management, and hence does not serve the purpose. Also note that any privileged methods that is scoped within the closure does not have access to any instance variable of the class, hence would remain as an utility functions that is useable only to '*MyClass'* methods.

The following diagram, a snapshot of the JavaScript Heap, shows **the same instances of the public method definitions, and the Closure scope/context shared by the instances** when 2 instances of *'MyClass*' is created -

In the snapshot, the first *MyClass* (**@40331**) is the Class definition and the other two *MyClass* are the instances (obj, obj1). It can be observed that the class definition holds a context (**@40321**) with reference to closure object, and function/variable (**@40323, @15043** respectively) that were defined within the IFFE. Since the *MyClass* definition is within a closure, the same instance of *privilegedSharedFunction()* and *privilegedSharedVar* is referred by all the instances of MyClass, hence the variables cannot hold an instance state and plays a **Static role** within *MyClass*.

Console output of the complete source code:

```
-------------------------------------
obj.myAction == obj2.myAction
 --- myAction is SAME instance
-------------------------------------
globalMyClassMyAction == obj2.myAction
 --- myAction is SAME instance
-------------------------------------
---------------------
obj.getPrivilegedVar() -> foo
obj2.getPrivilegedVar() -> foo
---------------------
obj.updatePrivilegedVar('hello')
---------------------
obj.getPrivilegedVar() -> hello
obj2.getPrivilegedVar() -> hello
---------------------
-------------------------------------
```

Conclusion:

The above analysis highlights the efficiency aspects of JavaScript environment when it comes to defining Classes and its member scope. It is very evident that introducing *private scope* for members using various options that are not inherently recommended by the language leaves us with efficiency problems. Based on the above study it is advisable to focus on *efficiency* over the introduction of *private/privilege scope*.

It is recommended to follow the prototype approach for method definitions (Defining prototype level Methods outside the Class Constructor) and indicate the private nature of the members (methods, properties) by following widely adopted naming conventions (E.g. names starting with "_").