JavaScript Design Patterns

BANU PRAKASH C

BANUPRAKASHC@YAHOO.CO.IN

What is a pattern?

A pattern is a reusable solution that can be applied to commonly occurring problems in software design - in our case - in writing JavaScript-powered applications.

Another way of looking at patterns are as templates for how you solve problems - ones which can be used in quite a few different situations.

Design patterns benefits

Patterns are proven solutions

Patterns can be easily reused

Patterns can be expressive

Patterns can provide generalized solutions which are documented in a fashion that doesn't require them to be tied to a specific problem

Certain patterns can actually decrease the overall file-size footprint of your code by avoiding repetition

Patterns add to a developers vocabulary, which makes communication faster

Patterns that are frequently used can be improved over time by harnessing the collective experiences other developers using those patterns contribute back to the design pattern community

Categories Of Design Pattern

Creational Design Patterns

Creational design patterns focus on handling object creation mechanisms where objects are created in a manner suitable for the situation you are working in.

Some of the patterns that fall under this category are: Constructor, Factory, Abstract, Prototype, Singleton and Builder.

Structural Design Patterns

Structural patterns are concerned with object composition and typically identify simple ways to realize relationships between different objects. They help ensure that when one part of a system changes, the entire structure of the system doesn't need to do the same. They also assist in recasting parts of the system which don't fit a particular purpose into those that do. Patterns that fall under this category include: Decorator, Facade, Flyweight, Adapter and Proxy.

Behavioral Design Patterns

Behavioral patterns focus on improving or streamlining the communication between disparate objects in a system.

Some behavioral patterns include: Iterator, Mediator, Observer and Visitor

Creational Pattern

```
//In JavaScript, the three common ways to create new objects
//Each of the following options will create a new empty object
var firstObj = {};
var secondObj = Object.create(null);
var thirdObj = new Object();
//There are then four ways in which keys and values can then be assigned to an object:
// ECMAScript 3 compatible approaches
// 1. Dot syntax
firstObj.name = "Banu Prakash"; // write property
var authorName = firstObj.name; // access property
// 2. Square bracket syntax
secondObj['name'] = "Rahul Prakash"; // write property
var empName =  secondObj['name']; // access property

// ECMAScript 5 only compatible approaches
Object.defineProperties(thirdObj, {"name":
                                   { value:"Smith", writable:true,enumerable:true, configurable:true},
                                   "age":
                                   {value:34, writable:true,enumerable:true, configurable:true}});
```

The Constructor Pattern

Constructors are used to create specific type of objects. In JavaScript, constructor functions are generally considered a reasonable way to implement instances.

By simply prefixing a call to a constructor function with the keyword '**new**', you can tell JavaScript you would like function to behave like a constructor and instantiate a new object.

```
//constructor pattern
function Employee(name, age) {
    this.name = name; //the keyword 'this' references the new object that's being created
    this.age = age;
    this.getName = function() {
        return name;
    }
    this.getAge = function() {
        return age;
    }
}
var firstEmp = new Employee("Smith" , 45);
var secEmp = new Employee("Diana", 22);
console.log(firstEmp.getName() + ", " + firstEmp.getAge());
```

Constructors With Prototypes

Functions in JavaScript have a property called a prototype.

When you call a JavaScript constructor to create an object, all the properties of the constructor's prototype are then made available to the new object.

```
//constructor pattern with prototype
function Employee(name, age) {
    this.name = name; //the keyword 'this' references the new object that's being created
    this.age = age;
}
Employee.prototype.getName = function() {
        return this.name;
    }
Employee.prototype.getAge = function() {
        return this.age;
    }

var firstEmp = new Employee("Smith" , 45);
var secEmp = new Employee("Diana", 22);

console.log(firstEmp.getName() + ", " + firstEmp.getAge());
```

Constructors With Prototypes contd..

Another approach of using prototype

```
//constructor pattern with prototype
function Employee(name, age) {
    this.name = name;
    this.age = age;
}
Employee.prototype = {
    getName : function() {
        return this.name;
    },
    getAge : function() {
        return this.age;
    }
}
```

The Singleton Pattern

The singleton pattern is thus known because traditionally, it restricts instantiation of a class to a single object.

In its simplest form, a singleton in JS can be an object literal grouped together with its related methods and properties as follows:

```
var singleton = {
    firstProperty : "Value for first property",
    secondProperty : "Value for second property",
    display : function() {
        console.log(this.firstProperty + ", " + this.secondProperty);
    }
}
singleton.display();
```

The Singleton Pattern

```
var SingletonPattern = (function() {
    function Singleton(options) {
        options = options || {}; // set options to the options supplied or an empty object if none provided.
        this.name = options.name || "Singleton Point";
        this.pointX = options.pointX || 2; //set the value of x
        this.pointY = options.pointY || 4; // set the value of y
    }
    var instance;
    var _static = {
        // This is a method for getting an singleton instance
        getInstance : function(options) {
            if (instance === undefined) {
                instance = new Singleton(options);
            }
            return instance;
        }
    };
    return _static;
})();

var singleObj = SingletonPattern.getInstance({pointY:5,name:"Testing"});
console.log(singleObj.pointX + ", " + singleObj.pointY +", " + singleObj.name);
```

Immediately-invoked Function Expressions (IIFE)s

An IIFE is effectively an unnamed function which is immediately invoked after it's been defined.

In JavaScript, because both variables and functions explicitly defined within such a context may only be accessed inside of it, function invocation provides an easy means to achieving privacy.

```
// an (anonymous) immediately-invoked function expression
(function(){ /*...*/})();
```

The Module Pattern

In JavaScript, the module pattern is used to *emulate* the concept of classes in

such a way that we're able to include both public/private methods and variables inside

a single object, thus shielding particular parts from the global scope.

This gives us a clean solution for shielding logic doing the heavy lifting while only

exposing an interface you wish other parts of your application to use.

The pattern is quite similar to an immediately-invoked functional expression (IIFE) except that an object is returned rather than a function.

The Module Pattern example

The module itself is completely self-contained in a global variable called shoppingCart.

The cart array in the module is kept private and so other parts of your application are unable to directly read it. It only exists with the module's closure and so the only methods able to access it are those with access to its scope (addItem(), getItems() , etc).

```
var shoppingCart = (function() {
    var cart = [];
    return {
        addItem : function(item) {
            cart.push(item);
        },
        getItems : function() {
            return cart;
        },
        getItemCount : function() {
            return cart.length;
        }
    };
})();
```

```
shoppingCart.addItem({
    product : "Dell",
    price : 54456.66
});
shoppingCart.addItem({
    product : "HP",
    price : 74643.99
});
var items = shoppingCart.getItems();
for (var i = 0; i < shoppingCart.getItemCount(); i++) {
    console.log(items[i].product + ":" + items[i].price);
}
```

The Module Pattern contd..

Advantages

- Provides cleaner approach for developers coming from an object-oriented background.
- Provides encapsulation from a JavaScript perspective.

Disadvantages

- You access both public and private members differently, when you wish to change visibility, you actually have to make changes to each place the member was used.
- You also can't access private members in methods that are added to the object at a later point.

The Revealing Module Pattern

Define all of your functions and variables in the private scope and return an anonymous object at the end of the module along with pointers to both the private variables and functions you wished to reveal as public.

```
var shoppingCart = (function() {
    var cart = [];
    function addItem(item) {
        cart.push(item);
    }
    function getItems() {
        return cart;
    }
    function length() {
        return cart.length;
    }
    return {
        add : addItem,
        get : getItems,
        getItemCount : length
    };
})();
```

Namespace Pattern

In many programming languages, name spacing is a technique employed to avoid **collisions** with other objects or variables in the global namespace.

They're also extremely useful for helping organize blocks of functionality in your application into easily manageable groups that can be uniquely identified

nsn.utilities.sort.reverseSort([5,3,5,78,2]);

nsn.utilities.math.max(34,23);

```
var nsn = {
    utilities:{
        math: {
            max: function max(a,b) {
                    return a>b ? a : b;
                }
            },
         sort: {
             reverseSort: function sort(data) {
                 data.sort(function(a,b) {
                                return b-a;
                            });
                 return data;
             }
         }
        }
};
```

Inject new behaviour into the namespace

inject new behaviour into the nsn.utilities.math namespace

```
// inject new behaviour into the nsn.utilities.math namespace
(function(obj) {
        obj.min = function min(a,b) {
            return a<b ? a : b;
        }
    })(nsn.utilities.math);
```

Observer pattern

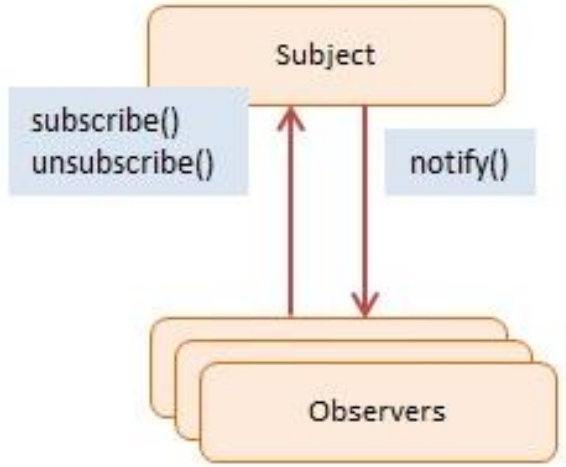
The Observer pattern offers a subscription model in which objects subscribe to an event and get notified when the event occurs.

This pattern is the cornerstone of event driven programming, including JavaScript.

The Observer pattern facilitates good object-oriented design and promotes loose coupling.

Subject: maintains a list of observers,
facilitates adding or removing observers

Observer: provides a update interface for objects that need
to be notified of a Subject's changes of state



Observer example

```
var pubsub = {};
(function(q) {
    var topics = {};
    var subUid = -1;

    q.subscribe = function(topic, func) {
        if (!topics[topic]) {
            topics[topic] = [];
        }
        var token = (++subUid).toString();
        topics[topic].push({
            token : token,
            func : func
        });
        return token;
    };

    q.publish = function(topic, args) {
        if (!topics[topic]) {
            return false;
        }
        var subscribers = topics[topic];
        var len = subscribers ? subscribers.length : 0;
        while (len--) {
            subscribers[len].func(topic, args);
        }
        return this;
    };

}(pubsub));
```

Observer example contd..

```
var testHandler = function(topics, data) {
    console.log(topics + ": " + data);
};

// Subscribers basically "subscribe" (or listen)
// And once they've been "notified" their callback functions are invoked
var testSubscription = pubsub.subscribe('example1', testHandler);

// Publishers are in charge of "publishing" notifications about events
pubsub.publish('example1', 'hello world!');
pubsub.publish('example1', [ 'test', 'a', 'b', 'c' ]);
pubsub.publish('example1', [ {
    'color' : 'blue'
}, {
    'text' : 'hello'
} ]);
```

Observable pattern

using jQuery

```
(function($) {
    var movieList = [];
    // subscribers
    $.subscribe("rating", function(e, movieTitle, userRating) {
        if (movieTitle.length) {
            movieList.push({
                movie : movieTitle,
                rating : userRating
            });
            $("#ratings").html("");
            $.each(movieList, function(index, movie) {
                $("#ratings").append(
                        "<li><strong>" + movie.movie
                                + "</strong> was rated " + movie.rating
                                + "/5</li>");
            });
        }
    });

    $('#add').on('click', function() {
        var strMovie = $("#movie_seen").val();
        var strRating = $("#movie_rating").val();
        $.publish('rating', [ strMovie, strRating ]);// publishers
    });
})(jQuery);
```