Chapter 65: Creational Design Patterns

Design patterns are a good way to keep your **code readable** and DRY. DRY stands for **don't repeat yourself**. Below you could find more examples about the most important design patterns.

Section 65.1: Factory Functions

A factory function is simply a function that returns an object.

Factory functions do not require the use of the `new` keyword, but can still be used to initialize an object, like a constructor.

Often, factory functions are used as API wrappers, like in the cases of jQuery and moment.js, so users do not need to use `new`.

The following is the simplest form of factory function; taking arguments and using them to craft a new object with the object literal:

```
function cowFactory(name) {
    return {
        name: name,
        talk: function () {
            console.log('Moo, my name is ' + this.name);
        },
    };
}

var daisy = cowFactory('Daisy');  // create a cow named Daisy
daisy.talk();  // "Moo, my name is Daisy"
```

It is easy to define private properties and methods in a factory, by including them outside of the returned object. This keeps your implementation details encapsulated, so you can only expose the public interface to your object.

```
function cowFactory(name) {
    function formalName() {
        return name + ' the cow';
    }

    return {
        talk: function () {
            console.log('Moo, my name is ' + formalName());
        },
    };
}

var daisy = cowFactory('Daisy');
daisy.talk();  // "Moo, my name is Daisy the cow"
daisy.formalName();  // ERROR: daisy.formalName is not a function
```

The last line will give an error because the function `formalName` is closed inside the `cowFactory` function. This is a closure.

Factories are also a great way of applying functional programming practices in JavaScript, because they are functions.

Section 65.2: Factory with Composition

'Prefer composition over inheritance' is an important and popular programming principle, used to assign behaviors to objects, as opposed to inheriting many often unneeded behaviors.

Behaviour factories

```
var speaker = function (state) {
    var noise = state.noise || 'grunt';

    return {
        speak: function () {
            console.log(state.name + ' says ' + noise);
        }
    };
};

var mover = function (state) {
    return {
        moveSlowly: function () {
            console.log(state.name + ' is moving slowly');
        },
        moveQuickly: function () {
            console.log(state.name + ' is moving quickly');
        }
    };
};
```

Object factories

Version ≥ 6

```
var person = function (name, age) {
    var state = {
        name: name,
        age: age,
        noise: 'Hello'
    };

    return Object.assign(     // Merge our 'behaviour' objects
        {},
        speaker(state),
        mover(state)
    );
};

var rabbit = function (name, colour) {
    var state = {
        name: name,
        colour: colour
    };

    return Object.assign(
        {},
        mover(state)
    );
};
```

Usage

```
var fred = person('Fred', 42);
```

```
fred.speak();        // outputs: Fred says Hello
fred.moveSlowly();   // outputs: Fred is moving slowly

var snowy = rabbit('Snowy', 'white');
snowy.moveSlowly();  // outputs: Snowy is moving slowly
snowy.moveQuickly(); // outputs: Snowy is moving quickly
snowy.speak();       // ERROR: snowy.speak is not a function
```

Section 65.3: Module and Revealing Module Patterns

Module Pattern

The Module pattern is a creational and structural design pattern which provides a way of encapsulating private members while producing a public API. This is accomplished by creating an IIFE which allows us to define variables only available in its scope (through closure) while returning an object which contains the public API.

This gives us a clean solution for hiding the main logic and only exposing an interface we wish other parts of our application to use.

```
var Module = (function(/* pass initialization data if necessary */) {
  // Private data is stored within the closure
  var privateData = 1;

  // Because the function is immediately invoked,
  // the return value becomes the public API
  var api = {
    getPrivateData: function() {
      return privateData;
    },

    getDoublePrivateData: function() {
      return api.getPrivateData() * 2;
    }
  };
  return api;
})(/* pass initialization data if necessary */);
```

Revealing Module Pattern

The Revealing Module pattern is a variant in the Module pattern. The key differences are that all members (private and public) are defined within the closure, the return value is an object literal containing no function definitions, and all references to member data are done through direct references rather than through the returned object.

```
var Module = (function(/* pass initialization data if necessary */) {
  // Private data is stored just like before
  var privateData = 1;

  // All functions must be declared outside of the returned object
  var getPrivateData = function() {
    return privateData;
  };

  var getDoublePrivateData = function() {
    // Refer directly to enclosed members rather than through the returned object
    return getPrivateData() * 2;
  };

  // Return an object literal with no function definitions
  return {
    getPrivateData: getPrivateData,
```

```
    getDoublePrivateData: getDoublePrivateData
  };
})(/* pass initialization data if necessary */);
```

Revealing Prototype Pattern

This variation of the revealing pattern is used to separate the constructor to the methods. This pattern allow us to use the javascript language like a objected oriented language:

```
//Namespace setting
var NavigationNs = NavigationNs || {};

// This is used as a class constructor
NavigationNs.active = function(current, length) {
    this.current = current;
    this.length = length;
}

// The prototype is used to separate the construct and the methods
NavigationNs.active.prototype = function() {
    // It is an example of a public method because is revealed in the return statement
    var setCurrent = function() {
        //Here the variables current and length are used as private class properties
        for (var i = 0; i < this.length; i++) {
                $(this.current).addClass('active');
        }
    }
    return { setCurrent: setCurrent };
}();

// Example of parameterless constructor
NavigationNs.pagination = function() {}

NavigationNs.pagination.prototype = function() {
// It is a example of an private method because is not revealed in the return statement
    var reload = function(data) {
        // do something
    },
    // It the only public method, because it the only function referenced in the return statement
     getPage = function(link) {
        var a = $(link);

        var options = {url: a.attr('href'), type: 'get'}
        $.ajax(options).done(function(data) {
           // after the ajax call is done, it calls private method
           reload(data);
        });

        return false;
    }
    return {getPage : getPage}
}();
```

This code above should be in a separated file .js to be referenced in any page that is needed. It can be used like this:

```
var menuActive = new NavigationNs.active('ul.sidebar-menu li', 5);
menuActive.setCurrent();
```

Section 65.4: Prototype Pattern

The prototype pattern focuses on creating an object that can be used as a blueprint for other objects through prototypal inheritance. This pattern is inherently easy to work with in JavaScript because of the native support for prototypal inheritance in JS which means we don't need to spend time or effort imitating this topology.

Creating methods on the prototype

```
function Welcome(name) {
  this.name = name;
}
Welcome.prototype.sayHello = function() {
  return 'Hello, ' + this.name + '!';
}

var welcome = new Welcome('John');

welcome.sayHello();
// => Hello, John!
```

Prototypal Inheritance

Inheriting from a 'parent object' is relatively easy via the following pattern

```
ChildObject.prototype = Object.create(ParentObject.prototype);
ChildObject.prototype.constructor = ChildObject;
```

Where `ParentObject` is the object you wish to inherit the prototyped functions from, and `ChildObject` is the new Object you wish to put them on.

If the parent object has values it initializes in its constructor you need to call the parents constructor when initializing the child.

You do that using the following pattern in the `ChildObject` constructor.

```
function ChildObject(value) {
    ParentObject.call(this, value);
}
```

A complete example where the above is implemented

```
function RoomService(name, order) {
  // this.name will be set and made available on the scope of this function
  Welcome.call(this, name);
  this.order = order;
}

// Inherit 'sayHello()' methods from 'Welcome' prototype
RoomService.prototype = Object.create(Welcome.prototype);

// By default prototype object has 'constructor' property.
// But as we created new object without this property  -  we have to set it manually,
// otherwise 'constructor' property will point to 'Welcome' class
RoomService.prototype.constructor = RoomService;

RoomService.prototype.announceDelivery = function() {
  return 'Your ' + this.order + ' has arrived!';
}
```

```
RoomService.prototype.deliverOrder = function() {
  return this.sayHello() + ' ' + this.announceDelivery();
}

var delivery = new RoomService('John', 'pizza');

delivery.sayHello();
// => Hello, John!,

delivery.announceDelivery();
// Your pizza has arrived!

delivery.deliverOrder();
// => Hello, John! Your pizza has arrived!
```

Section 65.5: Singleton Pattern

The Singleton pattern is a design pattern that restricts the instantiation of a class to one object. After the first object is created, it will return the reference to the same one whenever called for an object.

```
var Singleton = (function () {
        // instance stores a reference to the Singleton
        var instance;

        function createInstance() {
            // private variables and methods
            var _privateVariable = 'I am a private variable';
            function _privateMethod() {
                console.log('I am a private method');
            }

            return {
                // public methods and variables
                publicMethod: function() {
                    console.log('I am a public method');
                },
                publicVariable: 'I am a public variable'
            };
        }

        return {
            // Get the Singleton instance if it exists
            // or create one if doesn't
            getInstance: function () {
                if (!instance) {
                    instance = createInstance();
                }
                return instance;
            }
        };
    })();
```

Usage:

```
// there is no existing instance of Singleton, so it will create one
var instance1 = Singleton.getInstance();
// there is an instance of Singleton, so it will return the reference to this one
var instance2 = Singleton.getInstance();
console.log(instance1 === instance2); // true
```

Section 65.6: Abstract Factory Pattern

The Abstract Factory Pattern is a creational design pattern that can be used to define specific instances or classes without having to specify the exact object that is being created.

```
function Car() { this.name = "Car"; this.wheels = 4; }
function Truck() { this.name = "Truck"; this.wheels = 6; }
function Bike() { this.name = "Bike"; this.wheels = 2; }

const vehicleFactory = {
    createVehicle: function (type) {
        switch (type.toLowerCase()) {
            case "car":
                return new Car();
            case "truck":
                return new Truck();
            case "bike":
                return new Bike();
            default:
                return null;
        }
    }
};

const car = vehicleFactory.createVehicle("Car"); // Car { name: "Car", wheels: 4 }
const truck = vehicleFactory.createVehicle("Truck"); // Truck { name: "Truck", wheels: 6 }
const bike = vehicleFactory.createVehicle("Bike"); // Bike { name: "Bike", wheels: 2 }
const unknown = vehicleFactory.createVehicle("Boat"); // null ( Vehicle not known )
```