Advanced JavaScript for Web Sites and Web Applications

Design Patterns

What are Design Patterns?

- A pattern is a reusable solution that can be applied to commonly occurring problems in software design
  - in our case, while writing JavaScript web applications.
- Another way of looking at patterns are as *code templates* that can be adapted to solve problems in a variety of situations

The Module pattern

- One of the most common patterns used in JavaScript is the *Module pattern*
- It is an extension of the *object literal* pattern we looked at earlier in the course, but attempts to emulate object-oriented language features that are not present in JavaScript natively.
  - Primarily, the *visibility* of object properties and methods

Visibility - why it is a problem

- Consider this object literal application:

```
var VatCalc = {
    rate: 20,
    addVat: function (num) {
        var vat = (this.rate / 100) * num;
        return num + vat;
    }
}
```

Visibility - why it is a problem

- We can use its properties and methods throughout our script, which is useful:

```
var price = VatCalc.addVat(250);
var txt = 'VAT is ' + VatCalc.rate + '%';
```

Visibility - why it is a problem

- But it is also possible for code outside the object to change it's properties and methods:

```
// Code somewhere in script changes "rate" value
VatCalc.rate = 10;

// Other code is unaware of change...
// ... so results are wrong!
var price = VatCalc.addVat(250);
var txt = 'VAT is ' + VatCalc.rate + '%';
```

Visibility - why it is a problem

- To ensure that our applications are robust, we tend to want to prevent global-scope code from changing certain elements of the app
  - In this case, the VAT rate stored in the object
- Many of the commonly used *design patterns* attempt to address this issue.
- One such pattern is the *module pattern*...

```
// Introducing The Module Pattern
var VatCalculator = (function () {

    var rate = 20, calcVat, obj;

    calcVat = function(nett) {
        return (rate / 100) * nett;
    };

    obj = {
        addVat: function(num) {
            return num + calcVat(num);
        }
    };
    return obj;
})();
```

The Module pattern - explained

- The module is contained in an IIFE (has its own scope)
- The *return value* of the IIFE is captured and stored in a global variable:
  - `VatCalculator`
- Any variables defined inside `VatCalculator` are *private* and cannot be accessed from outside of the module:
  - `rate` and `calcVat`

The Module pattern - explained

- The IIFE *returns* an object (`obj`).
- The returned object contains the methods and properties you want to make available to the *outside*:
  - `addVat()`
- The methods contained within the returned object CAN access private methods and properties:
  - E.g. `addVat()` can call `calcVat()` and access `rate`

The Module pattern - using it

- After the IIFE has completed, the `VatCalculator` variable will be a reference to the *object that the function returned*, not the function itself.
- Code outside the module accesses the *public* functionality via the returned object:

```
var price = VatCalculator.addVat(250);
```

- But, the `rate` property and the `calcVat` method can not be accessed from outside.

The Module pattern - partial visibility

- In the previous example, `rate` is completely hidden from the global scope code.
  - It can not be changed (good)
  - It can not be read (not always good).

The Module pattern - partial visibility

- One solution to this common problem is to create a public *getter* method that can retrieve the internal `rate` value.
- Remember, methods defined in the returned object CAN access the *private* properties and methods.

```
// Partial visibility
var VatCalculator = (function () {
    var rate = 20, calcVat, obj;
    // Private methods omitted for clarity...
    obj = {
        addVat: function (num) { /* code */ },
        getRate: function () {
            // This method can access "rate"
            return rate;
        }
    };
    return obj;
})();
var price = VatCalculator.addVat(250);
// Code outside must use "getRate"
var vat_rate = VatCalculator.getRate();
```

The Revealing Module pattern

- Another common pattern which is derived from the module pattern is the *revealing module pattern*.
- The main difference between them is readability and manageability of the code.

The Revealing Module pattern

- In the *revealing module pattern*, all methods and properties are defined within the body of the immediately invoked function.
- The returned object contains *references* to the methods and properties you want to make public
  - Unlike the *module pattern*, no functions are defined in the returned object
- Example:

```
var VatCalculatorR = (function () {

    var rate = 20, calcVat, addVat, getRate, obj;

    // Method bodies omitted for clarity...
    calcVat = function (nett) { /* code */ };
    addVat = function (num) { /* code */ };
    getRate = function () { /* code */ };

    obj = {
        addVat: addVat,
        getRate: getRate
    };
    return obj;
})();
```

The Revealing Module pattern - using it

- From outside the module, there is little difference in how we interact with a regular module and a revealing module.
- Again, we simply use the method/properties of the returned object:

```
var price = VatCalculatorR.addVat(250);
var vat_rate = VatCalculatorR.getRate();
```

The Revealing Module pattern - with aliases

- The method names used in the returned object do not have to be the same as the methods which they are references to
- Providing *aliases* for the module methods can make it easier for *outside* code to interact with the module
- But code outside the module MUST use the aliases, not the function names they refer to

```
// Using aliases for public methods
var VatCalculatorR = (function () {
    var rate = 20, calcVat, addVat, getRate, obj;

    // Method bodies omitted for clarity...
    calcVat = function (nett) { /* code */ };
    addVat = function (num) { /* code */ };
    getRate = function () { /* code */ };

    // Aliases for "public" methods
    obj = {
        add: addVat,
        get: getRate
    };
    return obj;
})();
```

Using the aliases

- Code outside the module still uses the methods/properties of the returned object

```
// Call the "add" and "get" methods
var price = VatCalculatorR.add(250);
var vat_rate = VatCalculatorR.get();

// But this won't work:
var price2 = VatCalculatorR.addVat(250);
```

The Revealing Module pattern - benefits

- The main benefits of using this pattern are code manageability and readability:
  - All of the methods are defined in one *place*
  - Methods can easily be made public/private by removing or adding one line to the returned object definition
  - Short *aliases* can be provided for the outside code to use, helping to create a *fluent interface*

Exercise 1

- Download the *Session 4 exercises* document from Moodle and do *Exercise 1*

Module config objects

- As we saw before, you can use an *object* as a function argument
- We can also use this technique with the revealing module pattern
- By combining it with a configuration object stored within the module, we can create flexible, reusable programs

```
// Module config objects
var myModule = (function() {
    var config, setConfig, obj;
    config = {
        state: true,
        message: "Hello"
    };
    setConfig = function (settings) {
        config.state = settings.state;
        config.message = settings.message;
    };
    obj = {
        updateConfig: setConfig
    };
    return obj;
})();
```

Module config objects

- Because `setConfig` is *public*, code outside can change the settings:

```
var mySettings = {
    state: false,
    message: "Goodbye"
};
myModule.updateConfig(mySettings);
```

- But, it can only do this via `updateConfig()`, so we can validate/check values before storing and using them

Modules and `this`

- We can use the `this` keyword within our modules, but it will refer to something completely different, depending on the context it occurs in!

Modules and this

```
var message = 'I am Global';
var MyMod = (function () {
    var message, text, getMesssage;
    message = 'I am the Module';
    getMessage = function() {
        return this.message;
    };
    console.log(getMessage()); // Displays???
    return {
        message: 'I am the Object',
        getMessage: getMessage
    };
})();
console.log(MyMod.getMessage()); //Displays???
```

Modules and `this`

- The first console.log will display: *I am Global*
  - Method is executed in the context of the immediately invoked function, so `this` refers to the window
- The second console.log will display: *I am the Object*
  - Method is executed in the context of the returned object, and in objects, `this` refers to the object in which it is used

Exercise 2

- Now do *Exercise 2*

Search arrays with `indexOf`

- You can search for an element in an array with the `indexOf` method of the Array object:

```
array.indexOf(element)
```

- It returns the *index* of the first occurrence of the element passed to it, or `-1` if the element isn't found in the array.
  - Note, `indexOf` performs a strict comparison when searching the array!

`indexOf` example

```
// An array of strings
var beatles = ["John", "Paul", "George", "Ringo"];

// Search array for names with indexOf
beatles.indexOf("John"); // returns 0
beatles.indexOf("George"); // returns 2
beatles.indexOf("Mick"); // returns -1
beatles.indexOf("george"); // returns -1
```

Exercise 3

- Now do *Exercise 3*