ADVANCED JAVASCRIPT FOR WEB SITES AND WEB APPLICATIONS

Session 4

Patterns

WHAT ARE PATTERNS?

> *A pattern is a reusable solution that can be applied to commonly occurring problems in software design - in our case - in writing JavaScript web applications.*
>
> *Another way of looking at patterns are as templates for how we solve problems - ones which can be used in quite a few different situations.*

Patterns are proven solutions: They provide solid approaches to solving issues in software development using proven techniques that reflect the experience and insights the developers that helped define them bring to the pattern.

Patterns can be easily reused: A pattern usually reflects an out of the box solution that can be adapted to suit our own needs. This feature makes them quite robust.

Patterns can be expressive: When we look at a pattern there's generally a set structure and vocabulary to the solution presented that can help express rather large solutions quite elegantly.

Addy Osmani - Learning JavaScript Design Patterns

MODULE PATTERN

```
var myModule = (function () {

  var

    myPrivateProperty = "someValue",

    myPrivateMethod = function () {
      console.log("JavaScript is great!");
    };

  return {

    myPublicProperty: "someOtherValue",

    myPublicMethod: function () {
      myPrivateProperty; // call private property here
      myPrivateMethod(); // call private method here
    }

  }

})();
```

```
myModule.myPublicProperty(); // someOtherValue

myModule.myPublicMethod(); // run that method
```

```
myModule.myPrivateProperty; // error!

myModule.myPrivateMethod(); // error!
```

The module is contained in a global variable ("myModule" here).

It uses the single var pattern.

Any variable of function inside this variable is private and cannot be accessed outside of this module.

It returns an object (through the `return` keyword), that you use to expose the methods and properties you need.

- less danger of polluting the global namespace and conflicting with other modules
- namespace all our functions and variables (as properties and method of the module)
- private and public methods and properties of the module (don't need to expose everything)

REVEALING MODULE PATTERN

```
var myModule = (function () {

  var

    myPrivateProperty = "someValue",

    variableToExpose = "someOtherValue",

    myPrivateMethod = function () {
      console.log("JavaScript is great!");
    },

    functionToExpose = function () {
      myPrivateProperty; // call private property here
      myPrivateMethod(); // call private method here
    };

  return {
    print: variableToExpose,
    run: functionToExpose
  };

})();
```

```
myModule.print;

myModule.run();
```

Remember that you can still pass arguments to the functions you use and return in the pattern!

```
var myModule = (function () {

  var

    doubleMyNumber = function (x) {
      return x * 2;
    };

  return {
    double: doubleMyNumber
  };

})();

myModule.double(2);
```

Exercise 1

Using the module pattern, build a simple application that will have multiple functions. One will add two arguments that are passed, one will multiple two arguments passed, one will divide two arguments.

The pattern should return the different functions that you have created (they become public methods) of the main application.

```
var MATHAPP = (function () {

  var
    privateAddFunction = function (a, b) {
      return a + b;
    },
    privateMultiplyFunction = function (x, y) {
      return x*y;
    },
    privateDivideFunction = function (q, r) {
      return q/r;
    };

  return {
    add: privateAddFunction,
    multiply: privateMultiplyFunction,
    divide: privateDivideFunction
  };

}());
```

Config object

As we saw before, you can use an object as a configuration item.

In the module pattern, it can look like this:

```
var myModule = (function () {

  var

    config = {
      initialState: "true",
      initialValue: "value"
    },

    overrideConfig = function (state, value) {
      config.initialState = state;
      config.initialValue = value;
    };

  return {
    updateConfig: overrideConfig,
    readConfig: config
  };

})();
```

```
myModule.updateConfig(false, "new value");

console.log(myModule.readConfig);
```

Using *`this`* with the pattern

We can use the `this` keyword with this pattern if needed.

```
var myApp = (function () {

  var name = "Gilles",

  displayName = function () {
    return this.name
  };

  return {
    name: name,
    show: displayName
  };

})();

myApp.show();
```

Exercise 2

Re implement the exercise we saw last week (using `this` to create chainable methods) using the module pattern.

Previously we were setting the initial value to 0. Add an extra function that lets you set that value to anything. If it isn't explicitly set, the default will 0. Use an object to do that!

This was the solution for last week's exercise:

```
var myObj = {
  total: 0,
  add: function (x) {
    this.total = this.total + x;
    return  this
  },
  sub: function (x) {
    this.total = this.total - x;
    return  this
  },
  increment: function () {
    this.total = this.total + 1;
    return this
  },
  decrement: function () {
    this.total = this.total - 1;
    return this
  },
  result: function () {
    console.log(this.total);
    return this
  }
};
```

```
var APP = (function () {

  var total = {
    start: 0
  },
  setInitialValue = function (x) {
    this.total = x;
    return this;
  },
  privateAdd = function (x) {
    this.total = this.total + x;
    return this
  },
  privateSubstract = function (x) {
    this.total = this.total - x;
    return this
  },
  privateIncrement = function () {
    this.total = this.total + 1;
    return this
  },

[...]
[...]

  privateDecrement = function () {
    this.total = this.total - 1;
    return this
  },
  displayResult = function () {
    console.log(this.total.start);
    return this
  };

  return {
    total: total.start,
    start: setInitialValue,
    add: privateAdd,
    sub: privateSubstract,
    addOne: privateIncrement,
    subOne: privateDecrement,
    recall: displayResult
  };

})();
```

`indexOf`: how to find an element in an Array

You can search for an element in an array, using `arr.indexOf(element)`.

This returns the index of the first occurance of element, or -1 if the element isn't found.

```
var myArray = ["Gilles", "Aris", "Larry", "Rik"];

myArray.indexOf("Gilles") // returns 0
myArray.indexOf("Larry") // returns 2
myArray.indexOf("John") // returns -1
```

Check the support!!!

`indexOf()` uses strict equality!!

Exercise 3

Build a simple shopping list application that lets you:

- add an item to the shopping list
- remove an item from the list
- return the list itself
- count the number of items in the list

The list can be an array, the remove and add methods need to be chainable. The count and list return do not have to be.

We'll want to make sure an item doesn't already exist in the list before we add it!!

```
var myShoppingList = (function () {

  var shoppingList = [],

  countItemsInList = function () {
    console.log(shoppingList.length);
  },

  addItemToList = function (item) {
    var itemPosition = shoppingList.indexOf(item);
    if (itemPosition === -1) {
      shoppingList.push(item);
    }
    return this
  },

  removeItemFromList = function (item) {
    var itemPosition = shoppingList.indexOf(item);
    if (itemPosition > -1) {
        shoppingList.splice(itemPosition, 1);
    }
    return this
  };

  return {
    list: shoppingList,
    count: countItemsInList,
    add: addItemToList,
    remove: removeItemFromList
  };

})();

console.log(myShoppingList.add("apple").add("pears").add("apple").add("ap
console.log(myShoppingList.list);
console.log(myShoppingList.count());
```