ADVANCED JAVASCRIPT FOR WEB SITES AND WEB APPLICATIONS

Session 6

1. Pattern variations
2. Extending modules
3. Example

1. `Init` function

In a variation of the revealing module pattern, if you need to run some code, and don't need to expose methods externally, you can use an initialising function of the global object.

```
var APP = (function () {

  var privateVar,

    privateFunction = function () {
      // do something
    },

    // in the init function, add everything
    // that needs to happen when the app starts
    init = function () {
      privateFunction(); // call private function
      // do other stuff here
    };

  return {
    init: init
  };

}());

APP.init();
```

This is useful if you need your application to run and do things, but don't need to return properties and method through the object.

If you need to *trigger* the app.

2. Extend existing modules

When using the module pattern, you are not limited to storing everything under the same object or even the same file.

You can break down your code into the core application and different modules and organise it in different files.

extending the module directly

Here is our initial module:

```
var module = (function () {

  var publicVar,

    publicMethod = function () {
      // do something
    };

  return {
    exposeVar: publicVar,
    exposeMethod: publicMethod
  };

}());
```

We can extend the properties and methods of the module created:

```
module.newProperty = "some value";

module.newMethod = function () {
  // extend module by adding
  // a new method
};
```

And we can extend improve the previous code slightly:

```
var module = (function (originalModule) {

    // do some private stuff here
    var
      anotherPrivateVar = "",
      privateMethodAgain = function () {};

    // extend the original module here
    // using private stuff
    originalModule.anotherMethod = function () {
      anotherPrivateVar;
      privateMethodAgain();
    };

  return module

}(module));
```

In this last pattern, we:

- import the module as an argument of the immediately invoked function
- extend the imported module

We get all the benefits of a local scope.

extending the module by adding a sub module

Here is our initial module:

```
var module = (function () {

  var publicVar,

    publicMethod = function () {
      // do something
    };

  return {
    exposeVar: publicVar,
    exposeMethod: publicMethod
  };

}());
```

Add a new property to the module, that uses the module pattern itself and is an object:

```
module.scopedModule = (function () {

  var privateVar,

    privateMethod = function () {
      // do something
    };

  return {
    newModuleVar: privateVar,
    newModuleMethod: privateMethod
  };

}());
```

We can still access the module's initial methods and properties:

```
module.exposeVar;

module.exposeMethod();
```

And the properties and methods of my sub module:

```
module.scopedModule.newModuleVar;

module.scopedModule.privateMethod();
```

extending the module through new dependable modules

Here is our initial module:

```
var module = (function () {

  var publicVar,

    publicMethod = function () {
      // do something
    };

  return {
    exposeVar: publicVar,
    exposeMethod: publicMethod
  };

}());
```

We can create a new module that will depend on our initial one:

```
var newModule = (function (originalModule) {

    var
      anotherPrivateVar = "",

      privateMethodAgain = function () {

        // use the original module here:
        originalModule.exposeVar;
        originalModule.exposeMethod();

      };

  return {
    newModuleProp: anotherPrivateVar,
    newModuleMethod: privateMethodAgain
  }

}(module));
```

In this last pattern, we:

- create a new global variable for a new module
- import the module as an argument of the immediately invoked function
- create properties and method for our new module, but that depend on the original
- return a new object

In both of these patterns, the extension could be stored in different files so each corresponds to a specific functionality of the application.

coreModule.js

```
var module = (function () {

  var publicVar,

    publicMethod = function () {
      // do something
    };

  return {
    exposeVar: publicVar,
    exposeMethod: publicMethod
  };

}());
```

extension.js

```
var newModule = (function (originalModule) {

    var
      anotherPrivateVar = "",

      privateMethodAgain = function () {

        // use the original module here:
        originalModule.exposeVar;
        originalModule.exposeMethod();

      };

  return {
    newModuleProp: anotherPrivateVar,
    newModuleMethod: privateMethodAgain
  }

}(module));
```

As long as you load both file in the correct order!

How to use this?

Let's reuse the math application we built previously.

This was our code (roughly):

```
var MATH = (function () {

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

  return {
    total: total.start,
    set: setInitialValue,
    add: privateAdd,
    sub: privateSubtract
  };
```

```
})();
```

This gives us a simple way of adding or subtracting numbers and returning the result.

We now want to build a simple calculator using this code and extending it.

Have a look at the interface in `workshop6.html`

How do we start?

In `math.js`, we have our first module that does simple addition, subtraction, returns total and rests total.

In `calculator.js`, we'll add another module that manages the interface and events, and calls `MATH` module.

You can use the method you prefer to build the calcualtor, extend `MATH` directly, add a new scoped object to it or create a new module that depends on `MATH`.

The second module's structure is fairly simple: we can have an event handler registered in an `init` function.

The `init` function is then triggered and we don't need to expose anything else.

The event handler function for the `init` function can check which button was just clicked, and do something based on that.

Use event delegation!!!

Hints:

1. we'll need to set a flag to know which "add" or "subtract" button was used. When the users clicks on the result button, use this flag to add or subtract.

2. remember that MATH expects number variables, and that `.value` on an input element returns a string. You can parse a string as a number using `parseInt('number')`.

Event Handler function:

1. if the button was one of the digits:
Append the new digit to the view.

2. if the button was clear:
Clear the view and set the total of MATH to zero.

3. if the button was add:
Set flag to "add", use the `set` method of MATH to change the value of `total`, clear the view.

4. if the button was subtract:
Set flag to "subtract", use the `set` method of MATH to change the value of `total`, clear the view.

5. if the button was result:
Check if the flag is set to "add" or "subtract", use the `add` or `sub` method of MATH to add to the `total`, update the view with the result.

Solution

```
MATH.calc = (function (MATH) {

  var

    calcState = null,
    calculator = document.querySelectorAll(".calc")[0],
    view = document.querySelectorAll(".calc-view .form-control")[0],

    whichButton = function (e) {

      var
        buttonClicked = e.target,
        buttonType = buttonClicked.getAttribute("data-value");

      if (buttonClicked.classList.contains("button-digit")) {
        // append more digits to the view
        var existingData = view.value;
        view.value = existingData + buttonType;
        return
      }
      else if (buttonClicked.classList.contains("button-clear")) {
        // reset the view and total to 0
        view.value = "";
        MATH.total = 0;
        return
      }
```

```
    else if (buttonClicked.classList.contains("button-sub")) {
      // when the add button is clicked
      // set flag to "sub", set the total to the value of the view
      calcState = "sub";
      MATH.set(parseInt(view.value));
      view.value = "";
      return
    }
    else if (buttonClicked.classList.contains("button-result")) {
      // when result button is used
      // check the calc state flag and add or remove to the total
      // update view
      if (calcState === "add") {
        MATH.add(parseInt(view.value));
      }
      else if (calcState === "sub") {
        MATH.sub(parseInt(view.value));
      }
      view.value = MATH.total;
      return
    }
  },
```

```
    init = function () {
      calculator.addEventListener("click", whichButton);
    };

  return {
    init: init
  }

})(MATH);

MATH.calc.init();
```