Advanced JavaScript for Web Sites and Web Applications

Pattern variations, Global imports, Extending modules

The revealing module `init()` method

- In a variation of the revealing module pattern...
- ... if you need to run some code when the module is first initialised, you can use an *init* method

The revealing module `init()` method

- The *init* method can take care of essential tasks, such as:
  - Setting the module properties
  - Attaching event handlers
  - Creating event objects
  - Etc.
- You will normally return a pointer to the *init* method in the module's *returned object*
  - i.e. it is a *public* method

Example `init()` method

```
var APP = (function () {
    var el, report, init;
    report = function (){
        console.log(el);
    };
    init = function () {
        // Set properties, attach handlers, Etc.
        var id = 'content';
        el = document.getElementById(id);
        report();
    };
    return { init: init };
}());
```

Example `init()` method

- To initialise the module, simply call the `init` method:

```
APP.init();
```

`init()` method usage

- In this example, `init` was the only method returned from the module.
- This is common practice when you need your application to run and do a specific thing, but don’t need to return properties and methods via the object.

`init()` method usage

- The `init` method can also be used in a more conventional way, where other methods/properties are also returned from the module...

`init()` and friends

```
var APP = (function () {
    var el,report,init;
    report = function (){
        console.log(el);
    };
    init = function (){
        var id = 'content';
        el = document.getElementById(id);
    };
    return {
        init: init,
        run: report
    };
}());
```

Order of execution

- But in this scenario, care must be taken to always call the `init` method, before calling `run`:

```
APP.init();
APP.run();
```

- Alternatively, perform a check within `run` to see if `init` has been called
  - e.g. with a *flag* variable

Monitoring module *state*

```
var APP = (function () {
    var el, report, init, isReady = false;
    report = function (){
        // Check if "isReady" is true/false
        // true: proceed
        // false: run "init()"
    };
    init = function (){
        // Do stuff & set "isReady" to true
        isReady = true;
    };
    return { init: init, run: report };
}());
```

`init()` with arguments

- We can, and often do, pass arguments to the `init()` method
- Passing arguments to an `init()` method is particularly useful if our module needs to be configured before use...

`init()` with arguments

```
var APP = (function (){
    var el, report, init;
    report = function(){ /* do stuff */ };
    init = function (id){
        el = document.getElementById(id);
    };
    return {
        init: init,
        run: report
    };
}());
APP.init('content');
APP.run();
```

`init()` with a config object

- We can use the `init` method to extend the *configuration object pattern* we looked at earlier on the course, making it easier for outside code to work with our module

`init()` with a config object

```
var myModule = (function (){
        var config = {
            initialState: "true",
            initialValue: "value"
        },
        init = function (settings) {
            for(var prop in settings){
                config[prop] = settings[prop];
            }
        };
    return {
        init: init
    };
})();
```

Pattern variations - init() method

- When using the module, we can pass custom settings:

```
var customConfig = {
    initialState:"false",
    initialValue:"Hello"
};
myModule.init(customConfig);
```

- Or, we can use the default settings:

```
myModule.init();
```

Global imports

- We can import variables from the *global scope* to our module, by passing them as arguments to the immediately invoked function (IIFE).
- Like any function argument, we decide the *name* it will use within our function
  - i.e. We can create aliases for global variables

Global imports - example

```
// Argument passed to IIFE is stored in "d"
var myModule = (function (d) {
    var report = function () {
        // Use "document" aka: d
        var id = 'content';
        var el = d.getElementById(id);
        console.log(el);
    };
    return {
        run: report
    };
}(document)); // "document" is passed as arg
myModule.run();
```

Global imports

- In this example, we specify that the IIFE will accept an argument, who's value we store in `d`:

```
var myModule = (function (d) ...
```

- We are also passing the *document object* to the IIFE:

```
}(document));
```

- So, *document* is available within the module as `d`

Global imports - multiple arguments

- We can pass as many arguments as we like to the IIFE...

```
var randomModule = (function (){
    var randomMethod = function () {};
    return { run: randomMethod };
}());
var coreModule = (function (w, d, rm){
    var report = function (){
        // Using the imports
        el = d.getElementById('content');
        hash = w.location.hash;
        rm.run();
    };
    return {
        run: report
    };
}(window, document, randomModule));
coreModule.run();
```

Global imports

- As we see in the previous example, the variables we import can be native (document, window) or custom (randomModule)
- Note, it is not necessary for us to pass global elements in to our module in order to use them.
  - They are global, so we can access them anyway!
- The main benefit to importing them, is the aliases we can define for them within the module…
  - … Less typing for us!

Exercise 1

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

Extending existing modules

- When using the module pattern, you are not limited to storing everything in the same object or even the same file.
  - You can divide your code into a *core application* and a collection of *modules* that work with it
  - You can define your modules in separate files.
- There are several approaches we can take to achieve this kind of separation...

Extending existing modules: Option 1

- **Option 1:** *Extend the module directly*
- In JavaScript, we can dynamically add methods and properties to existing objects.
  - Warning: We can also override existing properties and methods!

Dynamically extending an object

```
var myObject = {
    name:'Joe',
    work: function () {
        console.log('Going to work!');
    }
}
// Extend the object:
myObject.surname = 'Bloggs';
myObject.rest = function () {
    console.log('Going to sleep!');
};
// Use the object's old & new props
myObject.work();
myObject.rest();
```

Extending existing modules: Option 1

- Consider that the revealing module returns an *object*…
  - Once it has been created, we can dynamically extend this object, adding new methods and properties to it
- *Result:* we have the functionality provided by the module object plus the methods we have added dynamically!

Consider this core module:

```
var coreModule = (function () {
    var message,report;
    message = 'Hello';
    report = function () {
        // do something
        console.log('I am an original method');
    };
    return {
        text:message,
        run:report
    };
}());
```

Extending the core module's object

```
// "coreModule" is reference to module object

// Add a new property
coreModule.newProperty = "some value";
// Add a new method
coreModule.newMethod = function () {
    console.log('I am a new method');
};
// Use core method
coreModule.run();
// Use new stuff
coreModule.newMethod();
console.log(coreModule.newProperty);
```

Option 1 - alternative

- Alternatively, we can do the direct extending from within another module.
- To do this:
  - we create a new module, which has the same name as the *core* module
  - We pass the core module's returned object to the new module as an argument
  - The new module will add to core object and return it.
- *Result:* One module object that holds members from both modules

Extending coreModule:

```
// New module, with same name as "core" module
// Passed "coreModule" obj is aliased as core
coreModule = (function (core){
    var extMethod = function(){
        console.log('I am a new method');
    };
    // Directly extend the original module
    core.newProperty = 'some value';
    core.newMethod = function (){
        extMethod();
    };
    return core;
}(coreModule));
```

Extending `coreModule`:

```
// Now "coreModule" contains old & new members
coreModule.run();
coreModule.newMethod();
console.log(coreModule.newProperty);
```

Extending existing modules: Option 1

- In this last pattern, we:
  - import the original module as an argument of the new module's IIFE
  - extend the imported module object
  - return the extended module object
  - get all the benefits of a local scope.

Tight coupling alert!

- BUT... The extension is useless without the original `coreModule` and errors may occur!
- Solution: Amend the argument passed to the IIFE so that an empty object is passed when `coreModule` is undefined.
- `coreModule||{}` : coreModule OR empty object

```
var coreModule = (function (core) {
    // module code
}(coreModule||{}));
```

Tight coupling solution - benefits

- Using the extension without the `coreModule` won't generate errors
- Extension could be usable as a stand-alone application
  - but won't have the original module methods and properties

Extending existing modules: Option 2

- **Option 2:** *Extend the module with a sub module*
- In this variation, the extension is stored as a nested module within the core module
- The extension also follows the revealing module pattern, and returns its own object
- We add the extension to the core module by directly extending the core module's return object

The extension (sub-module)

```
// Add extension as property of "coreModule"
// returned object will be stored as "subModule"
coreModule.subModule = (function () {
    var scopedProperty = 'some value',
        scopedMethod = function () {
            console.log('I am a new method');
        };
    return {
        newProperty: scopedProperty,
        newMethod:scopedMethod
    };
}());
```

Extending existing modules: Option 2

- We can access the methods and properties of the original module:

```
coreModule.run();
console.log(coreModule.text);
```

- And those belonging to the sub-module:

```
coreModule.subModule.newMethod();
console.log(coreModule.subModule.newProperty);
```

Extending existing modules: Option 2

- With this approach, we must take care using the `this` keyword.
- Depending on the context, it could refer to the core module, or the sub- module.
- Remember, in the context of the module's returned object, `this` refers to the object it is used in

```
var coreModule = (function (){
    var report = function (){
        console.log(this);
    };
    return{ run: report };
}());
coreModule.subModule = (function () {
    var scoped = function(){
        console.log(this);
    };
    return { newMethod: scoped };
}());
// What will the console say?
coreModule.run();
coreModule.subModule.newMethod();
```

Extending existing modules: Option 3

- **Option 3:** *Extend the module with dependent modules*
- In this variation, we:
  - define a new module and store it in a global variable
  - import the *core module* as an argument of the new module's IIFE
  - define properties and methods for the new module, but that use things stored in the core module
  - return a new, distinct object

Extending existing modules: Option 3

- This variation differs from the previous 2 options, as the result will be 2 distinct modules/objects in our application, as opposed to 1 extended module/object

The dependent module

```
var dependentModule = (function (core){
    var myProperty = 'some value',
        myMethod = function (){
            // Use core module methods/props
            console.log(core.text);
            core.run();
        };
    return {
        newProperty: myProperty,
        newMethod: myMethod
    };
}(coreModule));
```

Extending existing modules: Option 3

- The core module remains unchanged:

```
console.log(coreModule.text);
coreModule.run();
```

- The new module has no obvious link to the core module, but does use it's methods internally:

```
console.log(dependentModule.newProperty);
dependentModule.newMethod();
```

Extending existing modules: Separation

- In all of these variations, it is possible for us to store the extension code and the core module code separately.

Extending existing modules: Separation

- The core module definition is placed in one file. e.g.:
  - `core.js`
- Each extension is placed in it's own file. e.g.:
  - `extension.js`
  - `other_extension.js`
  - *etc.*

Extending existing modules: Separation

- Depending on what we need on a particular page, we simply link to the relevant files:

```
<script src="core.js"></script>
<script src="extension.js"></script>
```

- *But we must ensure they are linked in the correct order!*

Exercise 2

- Now do *Exercise 2* from the session exercises document.