

ADVANCED JAVASCRIPT FOR WEB SITES AND WEB APPLICATIONS

Session 1

1. Coding Conventions
2. Patterns
3. General code organisation

1. CODING CONVENTION

When writing JavaScript, it is important to follow coding conventions.

These rules are not set in stone but are a good way to get started in structuring your code, making it readable and reusable. This also makes it easier to work in teams with other developers.

Indentation

Intend your code, and intend it properly. This is key to writing readable code.

You can follow a simple rule: anything with curly braces should be indented.

Example:

```
function () {  
  var a = 0;  
  if (a === 0) {  
    function () {  
      for (var i = 0; i < 10; i++) {  
        //statements  
      }  
    }  
  }  
}
```

Spacing

Choose a spacing convention and follow it when writing all your scripts. If picking up someone else's code, try to follow their conventions.

Here are some conventions that you can start following for this course:

Note how many spaces separate the opening `function`, `if` or `for` statements from the keywords.

```
function (arg) {  
    //statements  
}
```

```
if (condition) {  
    //statements  
}  
else if () {  
    //statements  
}  
else {  
    //statements  
}
```

```
for (i=0; i>10; i++) {  
    //statements  
}
```

```
var myArray = ["a", "b", "c"];
```

```
var myObject = {  
  property: "abc",  
  property2: "def",  
  method: function () {  
    //do something  
  },  
  method2: function () {  
    //do something else  
  }  
}
```

Curly braces

Curly braces are not strictly required in JavaScript for `if` and `for` statements. But always include them, it improves readability and prevents program errors.

Example:

```
var a = 1;  
if (a === 1)  
  var b = 1;  
  runFunction();  
a = 0;
```

If the code is badly indented, it makes things worse:

```
var a = 1;  
if (a === 1)  
var b = 1;  
runFunction();  
a = 0;
```


User braces, even if they are not strictly required!

```
var a = 1;  
if (a === 1) {  
  var b = 1;  
  runFunction();  
  a = 0;  
}
```

Semicolons

As with curly braces, semicolons at the end of a statement are not strictly required in JavaScript.

To avoid confusion, always use them. This prevents the JavaScript engine from adding them where it think is best.

Naming

It is also a good idea to choose a naming convention when creating variable and function names.

Most commonly, developers use lower camel case.

```
myFunctionName()
```

```
var myVariable
```

You can also use uppercase for your global variables.

```
var GLOBALVAR;  
  
function wrapper () {  
    var localVar;  
}
```

Exercise 1

In the workshop folder, there is a file called exercise 1. open this with a text editor and reorganise the code.

Globals

Avoid using global variables.

With a simple program, it is easy to make sure your variable names do not conflict. But when building complex web applications, there could be dozens of different scripts running on a single page. And any clash between variable names will cause issues and bugs.

Remember that global variables are created when:

- you create a new variable without the `var` keyword
- you create a new variable with the `var` keyword but outside of any scope

```
a; //global  
b = 3; //global  
var c = 0; // global  
  
function () {  
    d = 3 // global  
    var e = 4 //non global  
}
```

Single var pattern

When declaring multiple variables in your script, use a single `var` keyword for all of them.

If you have multiple scope in your script, use a single `var` keyword for each scope.

Remember that in JavaScript, functions introduce a new scope. Variables declared inside a function are not visible outside of this function. Variables declared outside of a function, will be available within any function.

```
var a;  
var b = 3;  
var c = 0;
```

Preferred:

```
var a, b = 3, c = 0;
```

Or to make it easier to read:

```
var a,  
    b = 3,  
    c = 0;
```


Multiple var / scopes:

```
var a,  
    b = 0,  
    c = 1;  
  
function () {  
    var e = 0,  
        f = false,  
        g;  
}
```

It's also a good idea to initialise your variables with the correct type, when declaring them. If you know the type of variable you will need.

```
var a = 0, // we know we'll use this as a number  
    b = true, // we know we'll use this as a boolean  
    c = ""; // we know we'll use this as a string
```

We will have a look at function scopes and variable hosting in more details later on in the course.

Caching

Some operations in JavaScript are time consuming and it can be useful to cache the results of such operations.

For example, finding out the length of an array or finding elements in the DOM.

The best way to do this is to assign the result of the operation or the lookup to a variable and work on this variable for the rest of the script.

```
document.querySelectorAll(".myClass")
    .classList.add("anotherClass"); // add a class

document.querySelectorAll(".myClass")
    .classList.add("yetAnotherClass"); // add another class

document.querySelectorAll(".myClass")
    .insertAdjacentHTML("beforeend", "<p>some html</p>") // do something
else with the element
```

Preferred:

```
var element = document.querySelectorAll(".myClass");  
  
element.classList.add("anotherClass"); // add a class  
element.classList.add("yetAnotherClass"); // add another class  
element.insertAdjacentHTML("beforeend", "<p>some html</p>") // do  
something else with the element
```

For loop pattern

When using a `for` loop, there are a few things you can do to optimise it:

- as seen previously, if iterating over a collection or array, cache the object length
- use the single var pattern for the iterator (and length variable)

Non optimised loop:

```
for (i = 0; i < myArray.length; i++) {  
    // do something here  
}
```

Preferred:

```
var i,  
    max = myArray.length;  
  
for (i = 0; i < max; i++) {  
    // do something here  
}
```


Immediately-invoked functions

Another interesting pattern is anonymous functions that execute as soon as they are defined.

Using JavaScript's scope structure, this pattern can solve a lot of issues around global variables and scoping generally.

This is how the pattern works:

```
(function(){  
  // code  
})();
```

Any code inside the function will be run as soon as that function finishes.

Any variables and functions created inside this function will be visible only within its scope.

When to use this pattern:

- to avoid polluting the global name space
- it is common to use this pattern to wrap an entire script, to completely lock all variables declared there from the global scope

Conditional Expressions

Use `===` and `!==` over `==` and `!=`

Conditional expressions are evaluated using coercion and always follow these simple rules:

- Objects evaluate to `true`
- Undefined evaluates to `false`
- Null evaluates to `false`
- Booleans evaluate to the value of the boolean
- Numbers evaluate to `false` if `+0`, `-0`, or `NaN`, otherwise `true`
- Strings evaluate to `false` if an empty string `""`, otherwise `true`

Don't repeat yourself!

Reuse pieces of code as much as possible, avoid rewriting things over and over.

Avoid this:

```
if (element1.state === false) {  
    element1.stop();  
}  
if (element2.state === false) {  
    element2.stop();  
}  
if(element3.state === false) {  
    element3.stop();  
}
```

Prefer this:

```
var el = [element1, element2, element3],
    elLength = el.length;

for (var i = 0; i < elLength; i++) {
    if (el[i].state === false) {
        el[i].stop();
    }
}
```

GENERAL CODE ORGANISATION

When writing large and complex applications, you can make things a lot easier by organising the code using patterns. This means moving away from writing procedural code, to a more modular, object oriented way.

We'll have a look at the revealing module pattern, but there are a lot of other patterns you can use.

The pattern:

```
var APP = (function () {  
    var privateVar,  
        publicVar,  
        privateFunction = function () {  
            // do something  
        },  
        publicFunction = function () {  
            // do something else  
        };  
  
    // returns a public object  
    return {  
        returnValue: publicVar,  
        returnAction: publicFunction  
    };  
  
})();  
  
APP.returnValue;  
APP.returnAction();
```

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

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

It returns an object (through the `return` keyword), that you can use to expose methods and properties. In this example, it exposes the `publicVar` variable and `publicFunction` function through `returnValue` and `returnAction`.

It also uses the single var pattern.

It is a great way to create private and public methods that will either be visible or invisible to the outside the module.

Exercise 2

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  
        add = function (a, b) {  
            return a + b;  
        },  
        multiply = function (x, y) {  
            return x*y;  
        };  
        divide = function (q, r) {  
            return q/r;  
        }  
  
    return {  
        add: add,  
        multiply: multiply,  
        divide: divide  
    };  
  
})();
```

Exercise 3

Open workshop1.html.

On the next slide is a simple script to declare a click event to hide/show the paragraph on the page.

Rewrite this script using the module pattern.

 You can create two functions inside the app, one to change the class, one to register the event handler.

```
var link = document.getElementById("link");
var paragraph = document.getElementById("paragraph");

function hideText (event) {
    var pClass = paragraph.classList;
    if(pClass.contains("hide")) {
        pClass.remove("hide");
    }
    else {
        pClass.add("hide");
    }
    event.preventDefault();
}

link.addEventListener(
    "click",
    hideText,
    false
);
```

```
var APP = (function () {  
    var link = document.getElementById("link"),  
        paragraph = document.getElementById("paragraph"),  
        changeClass = function (event) {  
            var pClass = paragraph.classList;  
            if(pClass.contains("hide")) {  
                pClass.remove("hide");  
            }  
            else {  
                pClass.add("hide");  
            }  
            event.preventDefault();  
        },
```

```
    addhandler = function () {  
        link.addEventListener("click", changeClass, false);  
    };  
  
    return {  
        addhandler: addhandler  
    };  
}());  
  
APP.addhandler();
```

In a variation of the pattern, you can use an `init` function to *trigger* the application.

```
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();  
        };  
    // returns a public object  
    return {  
        init: init  
    };  
})();  
APP.init();
```