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 1 = 0; 1>10; 1++) {
          //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 seperate 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;
```

Use 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 chose a naming convention when creating variable and function names.

A common convention is to 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 `exercise1.js`. 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 scopes 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 3 = 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 el
```

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 somethi
```

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 >l; 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 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 !=

here is why:

Conditional expressions using == or `!=` 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`

Exercise 2

In the workshop folder, there is a file called `exercise2.js`. open this with a text editor and reorganise the code.

Keep in mind that you want to:

- avoid global
- use the single var pattern
- indent the code
- respect one naming convention
- use semi colons
- cache items when possible
- create optimized for loop
- use immidiately invoke function when possible

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 procedurial code, to a more modular, object oriented way.

There are lots of different code organisation patterns out there, each one with its specific application and use cases.

For the rest of this course, we'll have a look at the revealing module pattern. Which itself is built on the module pattern.

module pattern example:

```
var myModuleApp = {

  myProperty: "someValue",

  myConfig: {
    useCaching: true,
    language: "en"
  },

  myMethod: function () {
    //do something here
  }

};

myModuleApp.myProperty;

myModuleApp.myMethod();

myModuleApp.myConfig();
```