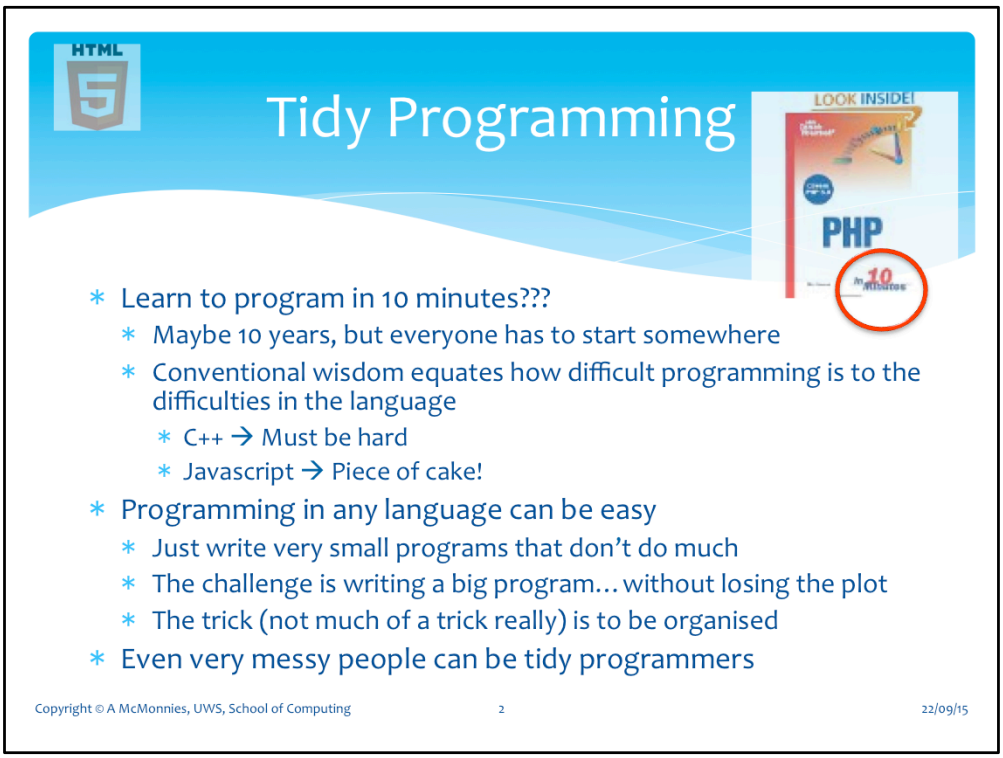

HTML
HTML5 & Javascript
Lecture 2: Basic Javascript Programming
Copyright © A McMonnies, UWS, School of Computing
1
22/09/15

Here again?



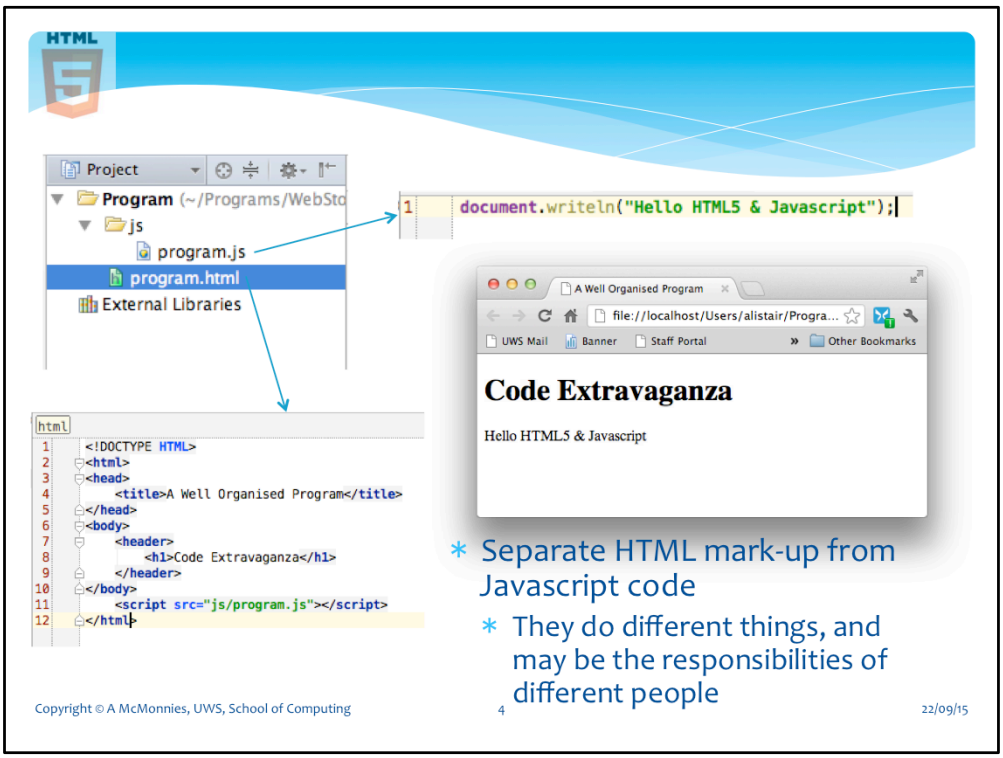
A pretty obvious message – if you want to learn to program in 10 minutes, you're missing the point. The real purpose of this module is to teach programming to a decent level of skill, and this is as hard in Javascript as it is in any other language. Don't expect miracles.

Programming is perceived as difficult because trivial programs don't do much (validating a textbox is not a major task), so blaming the language for being difficult really does miss the point. Programming properly can be learned in a few months, but can take decades to master. The best approach is to manage the class's expectations – this will take time, it will be difficult and frustrating, and quite a few of them will never get it (at least to the level where they could work as programmers). Even so, anyone in the class that does the work ought to pass (and maybe learn some useful stuff along the way).

Describing a major organizational principle, just as valid for programming as it is for building motor vehicles, office management or woodworking. Build a big thing by designing it as an assembly of components. Follow some rules (it almost doesn't matter which ones, but I'm suggesting):

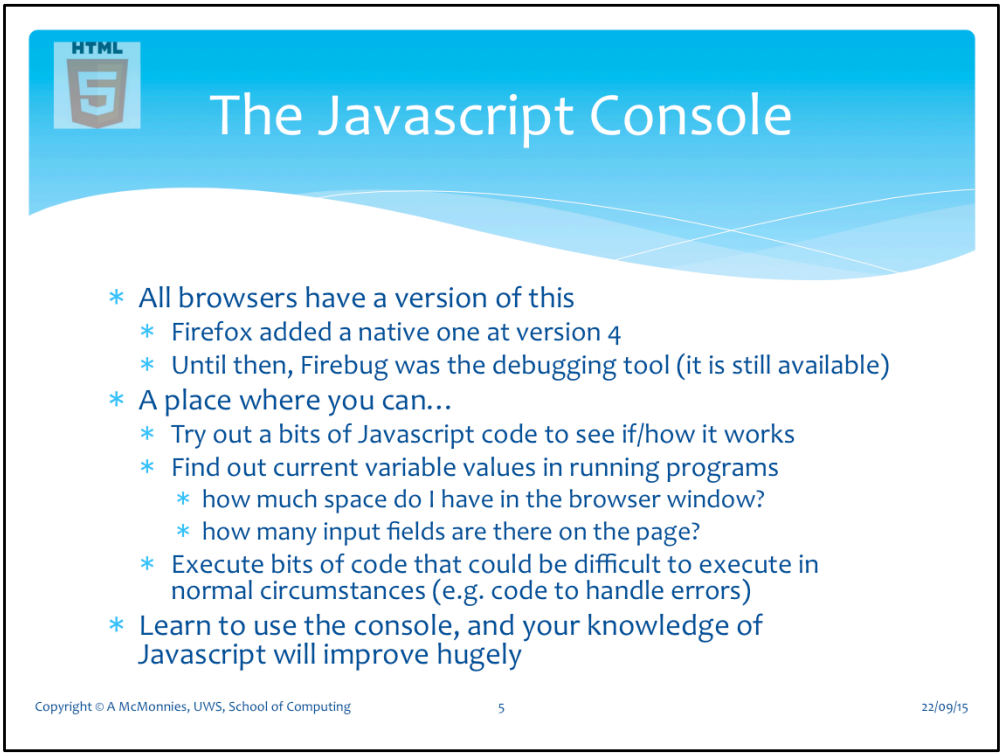
1. Start by deciding what the components are
2. Break these down into components of components, etc., bearing in mind that at each stage you need to be able to describe how the components interact
3. Use clear and sensible names for components
4. Work towards components where you can follow the entire working of each as a single thought process. Don't be scared of abstractions, but make sure that each abstraction makes sense at its own level.
5. Design interfaces (function names, variable sets) that are clear and obvious.
6. Reduce the number of bits of information passed between components where possible.
7. Document as you go. You'll never remember how you made it work this time tomorrow.
8. Use available tools.

At a more general level – read other people's code, blogs (identify the good ones), books, articles etc. When you have a task, find out how someone else has done it and re-use that approach possibly converting it into your language/idiom). Keep a



This slide is supposed to be showing a component view of an application (although the app is trivial). We've separated the major concerns into (in this case) presentation (html) and model (js). Explain that an obvious practical benefit of this is that some web developers might do HTML while others do JS, and that different tools might be used for each, so it makes sense to separate these tasks.

Explain also that every html developer should understand Javascript at some level, every JS programmer should be able to lash up a web page – this isn't car manufacture in the 1960s.



Some useful demos to do in front of the class. I'm assuming Chrome and its JS console, so if you're using a different browser, try these out before the class:

Space available in browser window:

```
window.innerWidth, window.innerHeight
```

Input fields on the page:

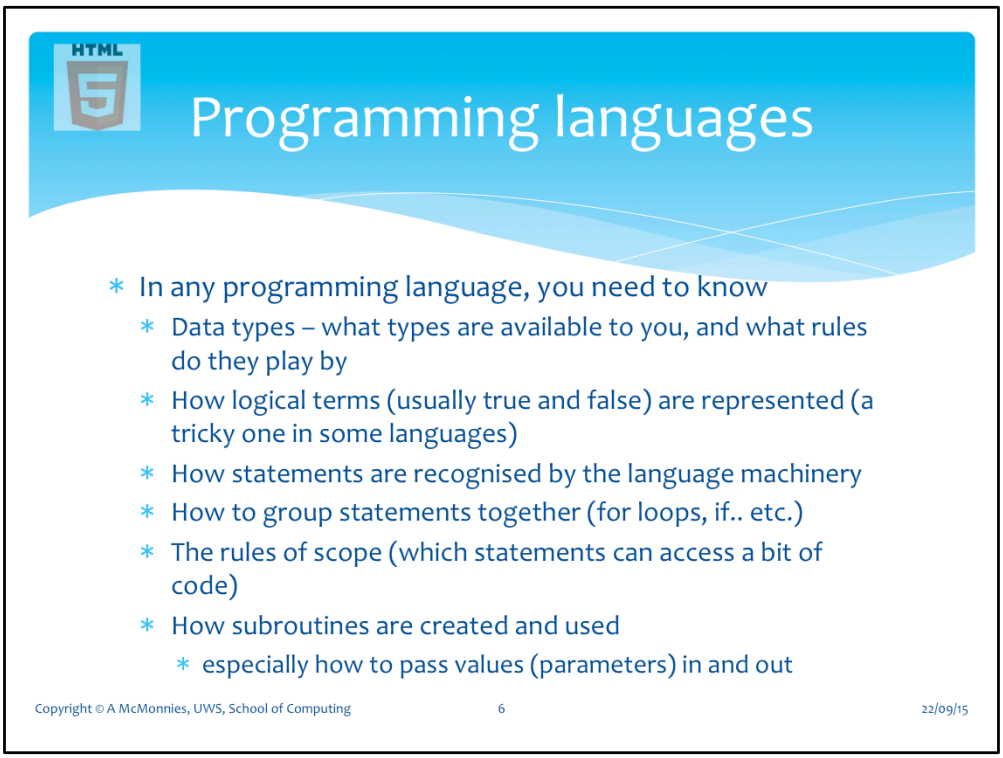
```
document.getElementsByTagName("input");
```

Add a H1 Element to the end of the page:

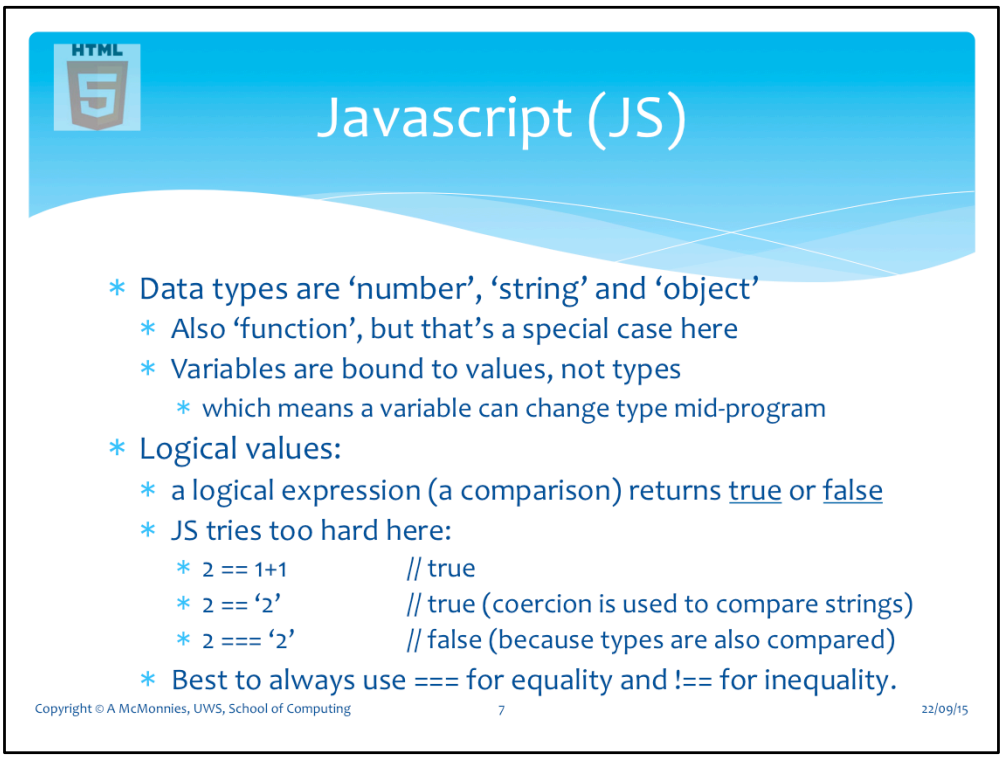
```
var h1 = document.createElement("h1");
h1.innerText = "Hello Mum!";
document.getElementsByTagName("body")[0].appendChild(h1);
```

Get the browser to lie to you about what it is:

```
navigator.appName, navigator.userAgent etc…
```



These are what I consider to be the major syntactical features within any language. Elaborate where you think it would help. For example, I'll probably briefly describe a couple of different approaches to data types (C/Java/VB/JS), Booleans in C & JS, statement delimiters in VB vs. C-like languages (explain the awful JS provisions while I'm at it), Blocks ({..} and Begin..End), Global, Local and the others that we don't get in JS, and spend a bit of time on parameters and arguments.



Some easy tests in a console window will make this stuff clear:

1. Create variables and then check using typeof:

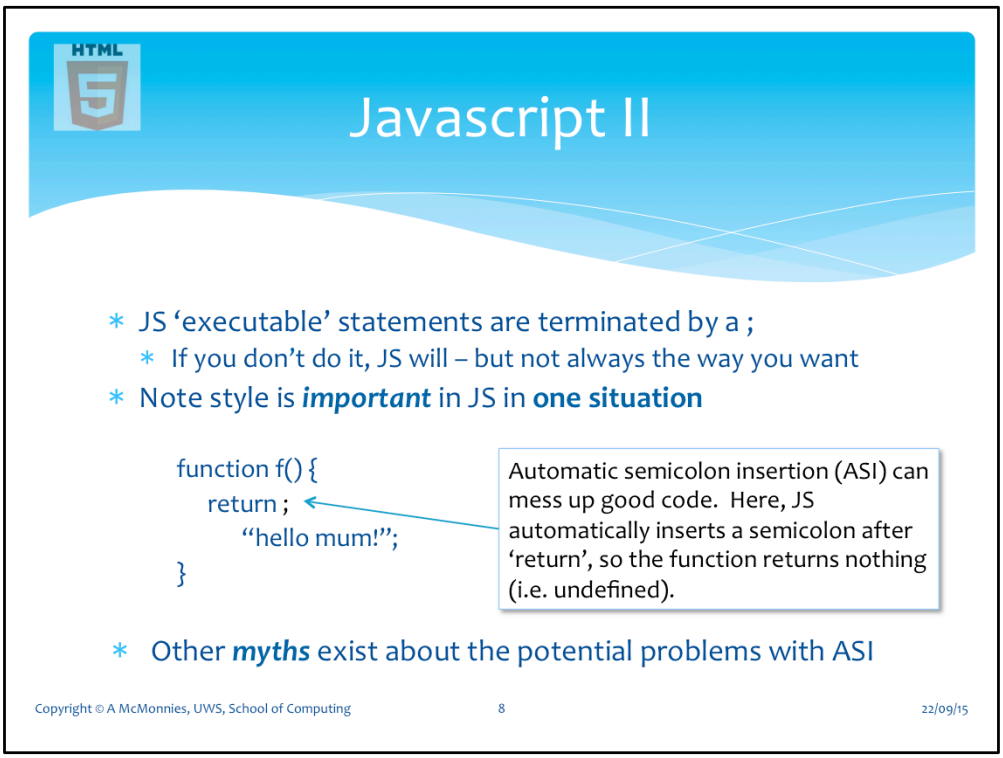
```
var v="12345";          typeof v;
v = 3;                      typeof v;
v = function f(){return 0;}     typeof v
```

2. Do some obvious comparisons:

```
3 == 3
3 == "3"
3 === "3"
3 !== 2+1
```

For terminating semicolons, this is a good demo of what goes wrong:

```
function f()
{
        return
        {
                name: "Fred",
                age: 25
        };
```

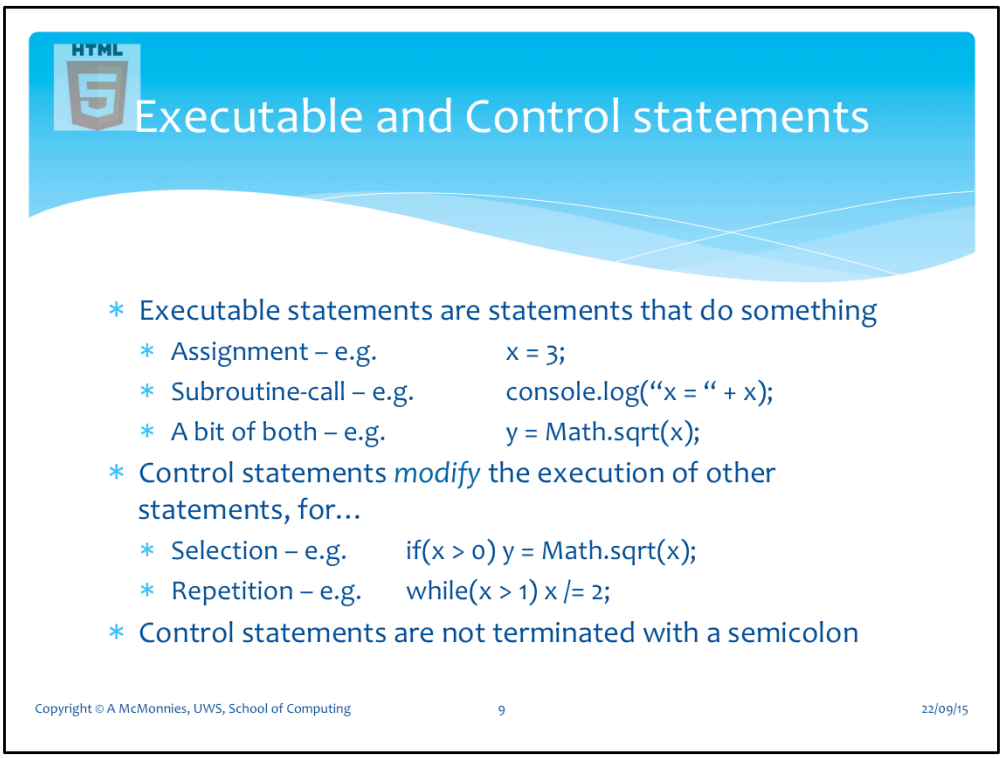


It's worth mentioning the debate about K&R style placement of braces with the alternative. i.e.

```
for(…){        vs.        for(…)
                          {
}
                          }
```

(and for if(), functions, do() etc.).

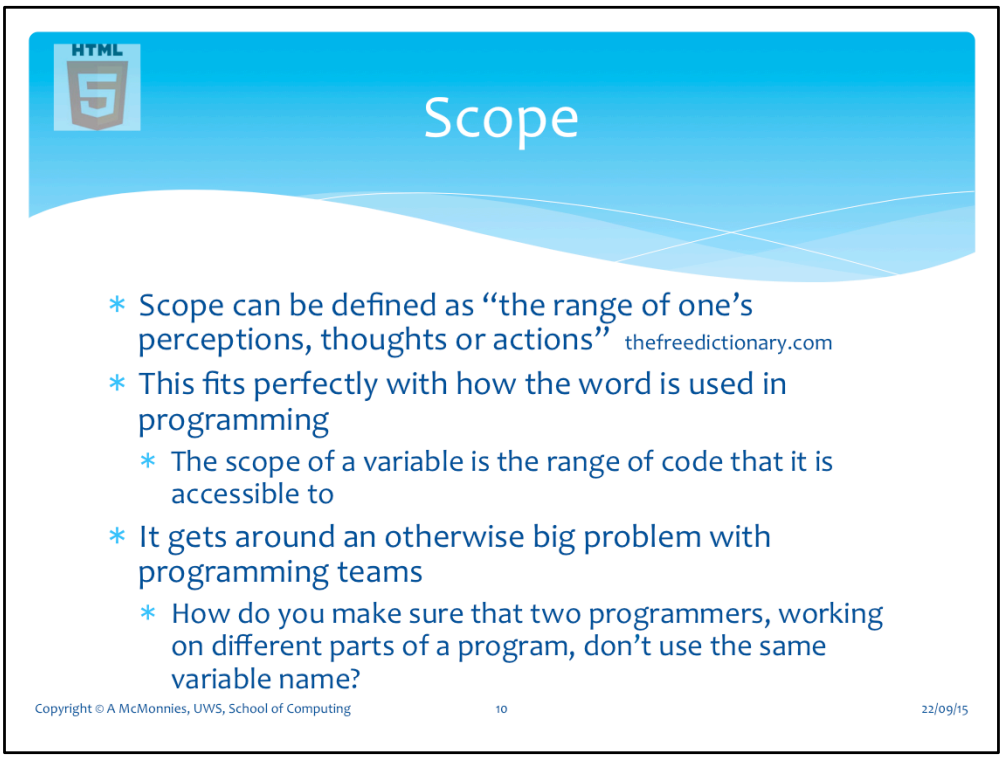
While it is largely a matter of style rather than syntax, this one example is the one that is always quoted as indicating that the K&R style should be followed. I'd go along with that because if you move on to C, C++ or C#, you'll find most code out there follows the style. However, I've never found a real-world instance of someone writing a return statement in that way – as an example, it's up there with don't ever trap your head in a mangle.



Here I'd explain a bit more about how a control statement differs from a non-control statement (and why). It is also worth explaining the for() structure as being a bit different, since it usually has executable elements in its make up. I'd point out that this can be true for while() and if() as well; just that it is less usual to think of conditions as being executable.
Some useful examples:

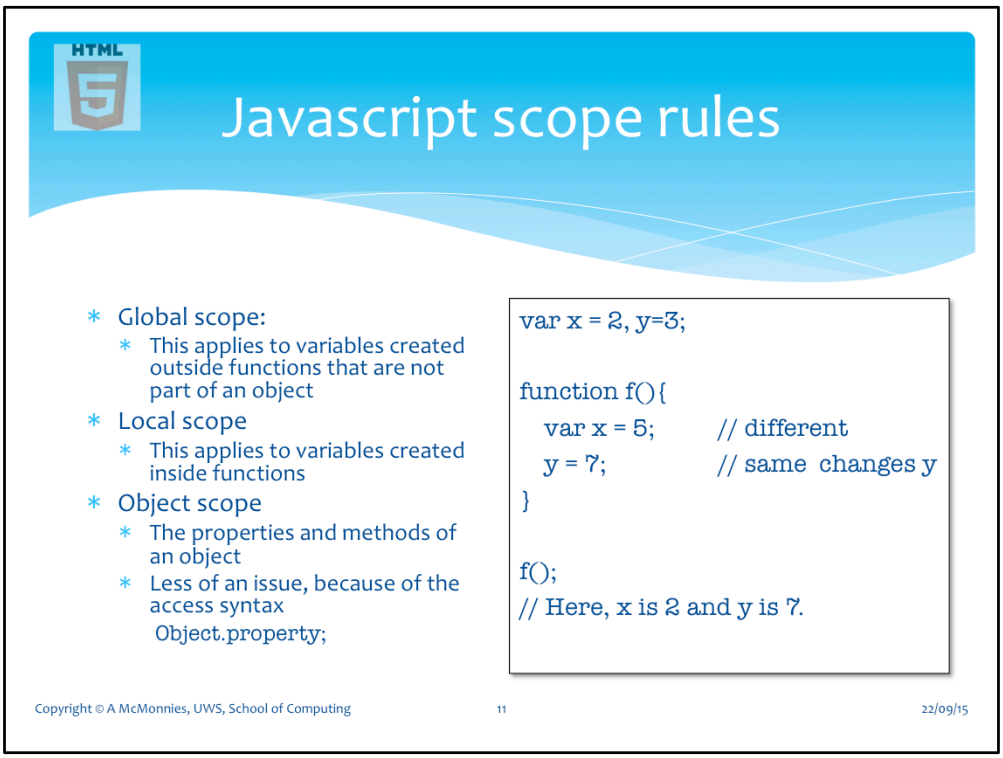
```
for(i=0, j=list.length; i<j; i++){          // Point out that this does a lot of execution within the for()
        // Repeated code                    // (and also that it is quicker because the length property is only accessed once.
}

do{
        // Code to repeat
} while(confirm("Go again?"));          // I'd count the confirm() here as executable
if( myObject.notSavedYet() ){           // Again, an executable condition.
        myObject.save();
}
```

Point out that since a semicolon is a statement on its own, using one after if(), while() etc. would be counter-productive.

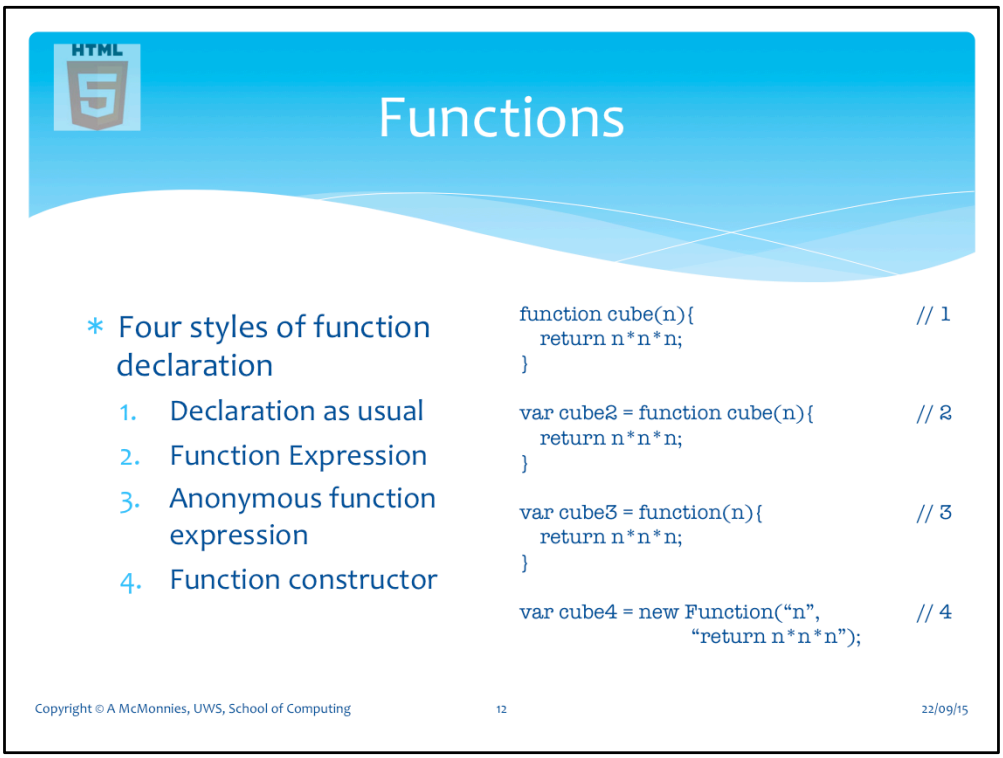


I usually start by explaining how the part-word 'scope' is used to define a field of view – e.g. telescope, microscope, really about affecting the size of the area you can see. Scope in programming is analogous to this, and works as a way of **controlling** what is visible in a block of code. Javascript is quote bad for this – allowing only local (i.e. in functions), global (i.e. in the whole application) and object (i.e. limited and narrowed by the dot notation to indicate properties of an object). No public, private, protected, inherited etc. as in other OOP languages.

JS provides one work-around – closures (the basis of a whole programming language – clojure) can be used to provide a kind of private scope. See later.



Should point out here that Javascript has few scope options compared to Java, for example.
Java has global, module, automatic local, static local, object and class scopes.

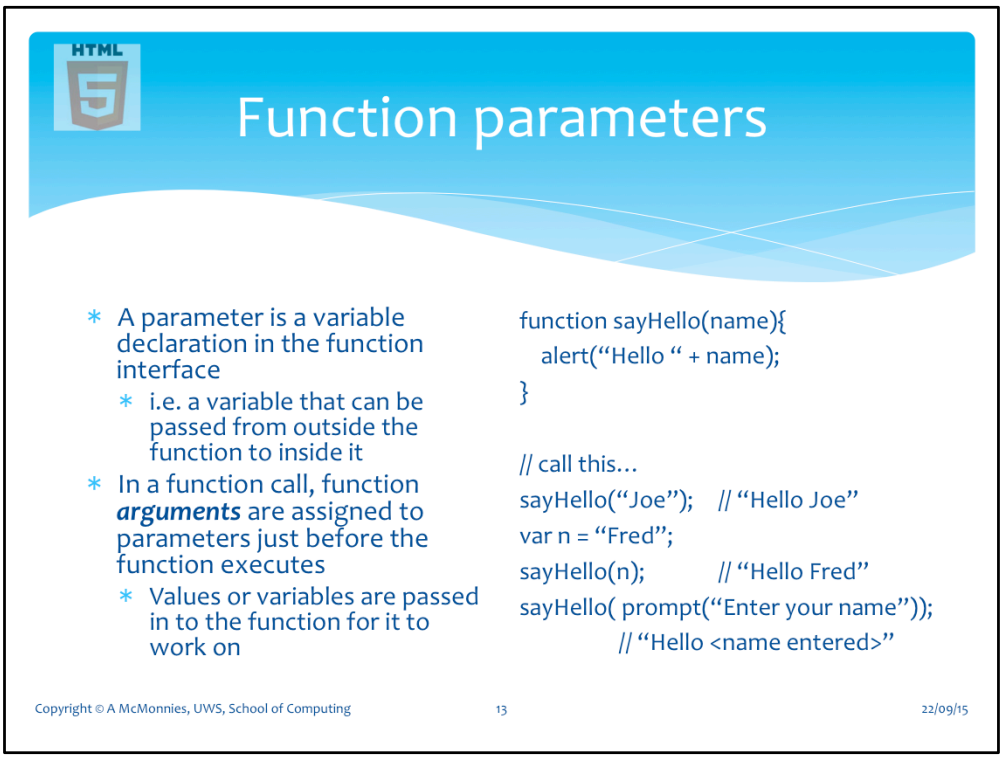


Best to explain these (at least the first 3) as largely compatible.

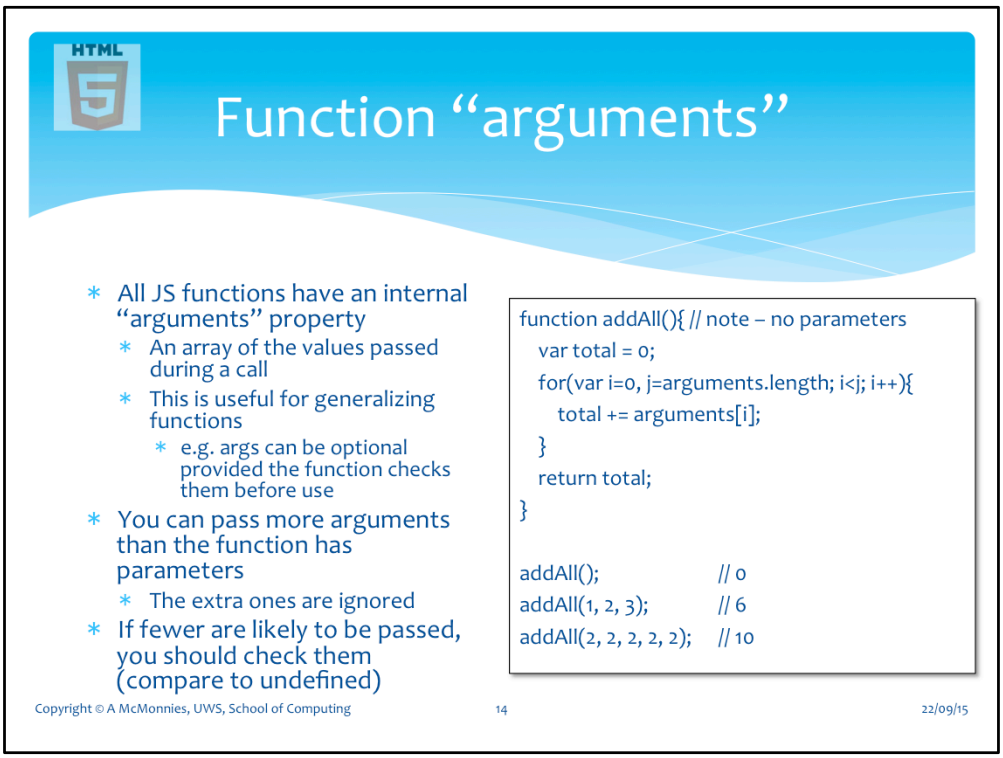
1. Is similar to the format in other languages (e.g. Java), and is most common.

2. It works the same as 1, but is more explicitly creating a function object and assigning it to a variable. This is good when mixed with object-oriented style of programming since it makes it more clear – e.g. how a function is bound to an object. The first (var) name is the one to call it with. The second is applied to the function's name property, so in a debugging situation, the second function name is the one you'd see in the object inspector.

3. This is just as callable as 2, but has no internal name. Often this doesn't matter; in many cases the assignment to a variable can be omitted making the anonymous function useful because it does not "pollute the namespace" – a term usually used to indicate that names in the JS global space need to be unique and every one that you use in some code could potentially clash with other library code you may be using, with unexpected results. The best use for anonymous function definitions is in declaring event handlers (dealt with next week), since objects that allocate event handlers will use internal names for the functions and these names won't be in scope anywhere else.



Parameters are often the first area that new programmers find 'inexplicable' or difficult. The use of parameters does require a bit of thought – abstractions for real values – always a tricky area. However, by explaining how parameters are used from the outside (i.e. as arguments), the need for them becomes a lot clearer – e.g. how would console.log() work if you could not pass some information into it? So the real explanation is about how what you place in between the brackets is passed into the executing function – this is a lot easier to justify and explain.



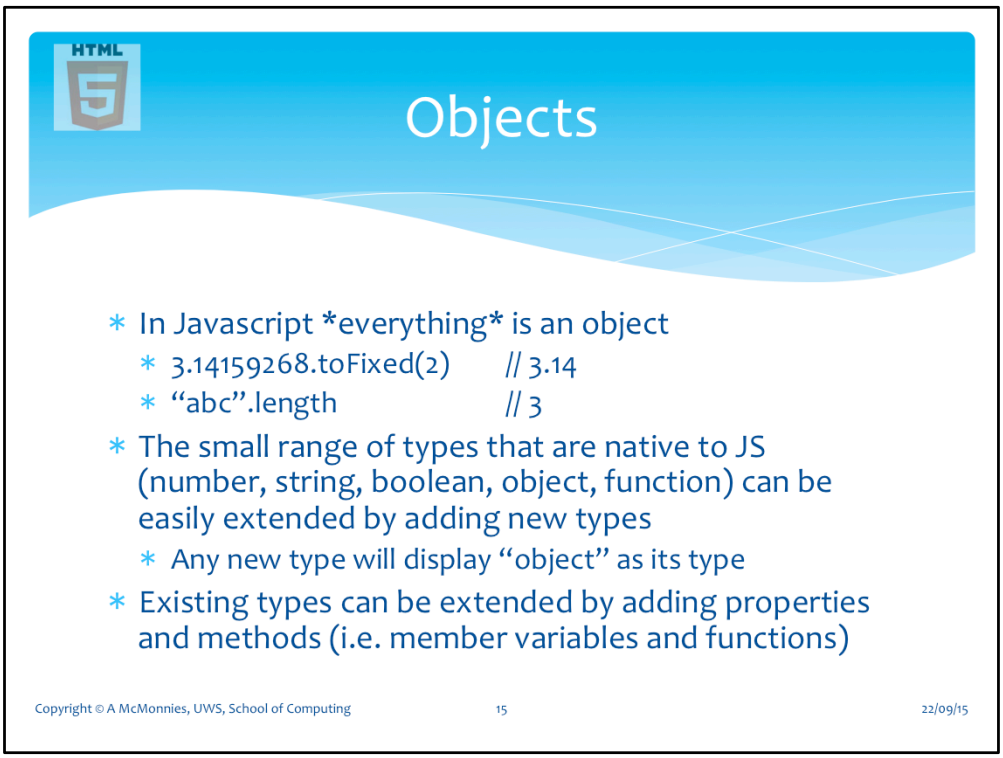
There’s potential for loads of examples in the lecture here. It is worth showing…

Arguments listed out, with:

```
for(i=0; i<arguments.length; i++){
   console.log(arguments[i], typeof arguments[i]);
}
```

Testing for an argument before using it:

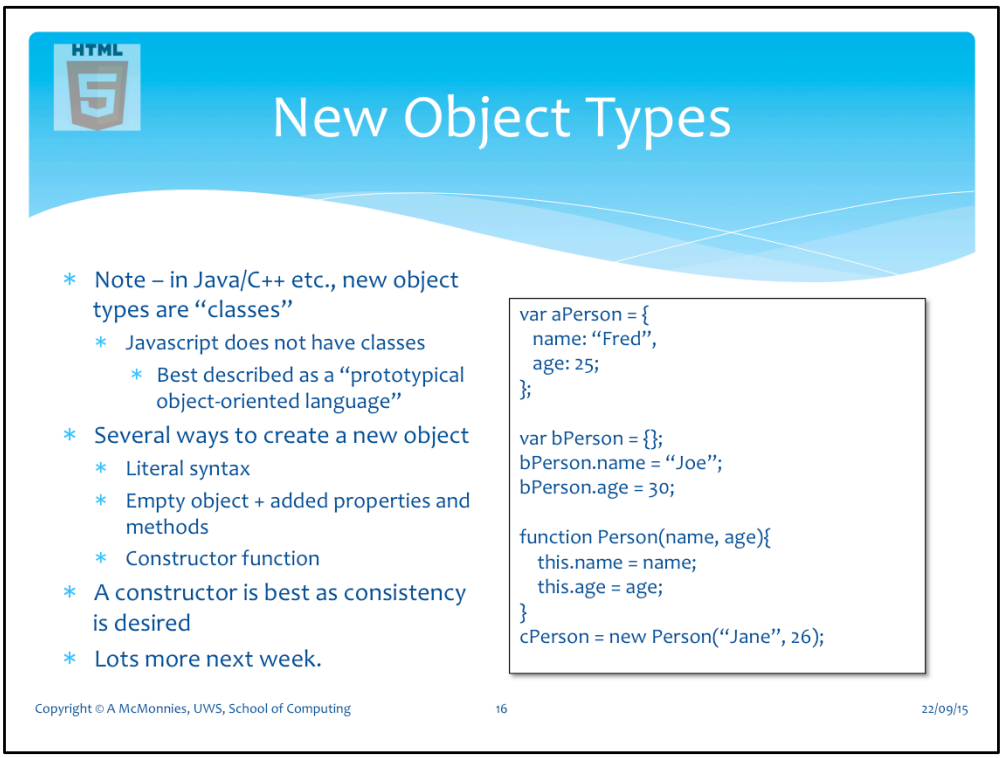
```
function info(name, dob, email){
   if(name) {
     console.log(“Hello “ + name);
   } else {
     console.log(“Hello anonymous person”);
   if(dob){
      var msPerYear = 1000*60*60*24*365.25; // millis*secs* mins*hrs*days
      var age = Math.floor((new Date()-dob)/msPerYear);
      console.log(“You are “ + age + “ years old”);
   } else {
      console.log(“How old?”);
```



It is worth showing the list of properties and methods for various types in a console window. The only one that won't work in the console is an integer number, since typing a '.' could just be to add a fractional part.

e.g. type "fred". (the dot is important) and you will be offered all of the string functions (toUpperCase(), split() etc.)

type 3.1415. and you will be offered toPrecision() ,toFixed() etc.



I'm suggesting for a very general, simple intro to OOP. Next week's work will concentrate on this.