
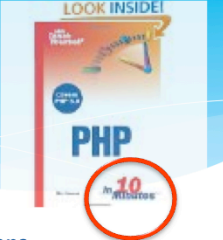


Here again?



# Tidy Programming



- \* Learn to program in 10 minutes???
- \* Maybe 10 years, but everyone has to start somewhere
- \* Conventional wisdom equates how difficult programming is to the difficulties in the language
  - \* C++ → Must be hard
  - \* Javascript → Piece of cake!
- \* Programming in any language can be easy
  - \* Just write very small programs that don't do much
  - \* The challenge is writing a big program... without losing the plot
  - \* The trick (not much of a trick really) is to be organised
- \* Even very messy people can be tidy programmers

Copyright © A McMonnies, UWS, School of Computing

2

22/09/15

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).



## Simple rules for organised programming

1. Separate concerns...
  - \* Don't put everything into one file, object or function
  - \* Break a program up into areas of responsibility
  - \* Concentrate on creating good *interfaces*
    - \* How a function is called is more important to get right than what it does
    - \* The variables and functions an object exposes is critical
2. Give things good names...
  - \* Appropriate for their purpose
  - \* Unambiguous
  - \* Short, but not so short as to introduce potential clashes
3. Write code that is clear and obvious...
  - \* Don't be tricky – e.g. avoid using undocumented language features
  - \* Aim for code that is readable, even if it takes longer to write
  - \* Document your code with comments as you write it (WebStorm is great for this)

Copyright © A McMonnies, UWS, School of Computing 3 22/09/15

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

The screenshot displays a web development environment with three main components:

- Project Explorer:** Shows a project structure with a 'Program' folder containing 'js' and 'program.html' files. Arrows indicate the relationship between these files and the code snippets.
- Code Editor:** Shows the content of 'program.html' with the following HTML structure:

```
1 <!DOCTYPE HTML>
2 <html>
3 <head>
4 <title>A Well Organised Program</title>
5 </head>
6 <body>
7 <header>
8 <h1>Code Extravaganza</h1>
9 </header>
10 </body>
11 <script src="js/program.js"></script>
12 </html>
```
- JavaScript Code:** A snippet showing a single line of JavaScript code: `document.writeln("Hello HTML5 & Javascript");`
- Browser Preview:** A window titled 'A Well Organised Program' showing the rendered HTML page. The title is 'A Well Organised Program' and the content is 'Hello HTML5 & Javascript'.

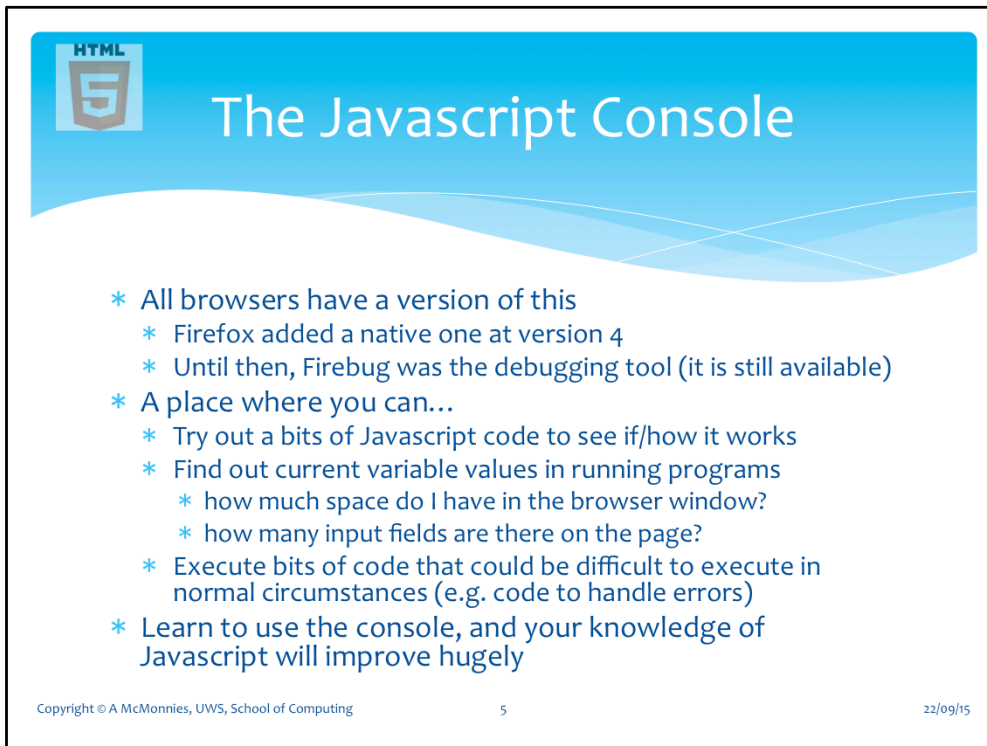
Below the browser preview, there are two bullet points:

- \* Separate HTML mark-up from Javascript code
- \* They do different things, and may be the responsibilities of different people

At the bottom left, there is a copyright notice: 'Copyright © A McMonnies, UWS, School of Computing'. At the bottom right, there is a date: '22/09/15'.

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.



## The Javascript Console

- \* All browsers have a version of this
  - \* Firefox added a native one at version 4
  - \* Until then, Firebug was the debugging tool (it is still available)
- \* A place where you can...
  - \* Try out a bits of Javascript code to see if/how it works
  - \* Find out current variable values in running programs
    - \* how much space do I have in the browser window?
    - \* how many input fields are there on the page?
  - \* Execute bits of code that could be difficult to execute in normal circumstances (e.g. code to handle errors)
- \* Learn to use the console, and your knowledge of Javascript will improve hugely

Copyright © A McMonnies, UWS, School of Computing 5 22/09/15

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...
```




The slide features a blue header with a white wavy line. On the left is the HTML5 logo, which consists of a shield-like shape with the letters 'HTML' above it and a large '5' inside. To the right of the logo, the title 'Programming languages' is written in a large, white, sans-serif font. Below the header, the slide has a white background with a bulleted list of topics. At the bottom, there is a small copyright notice on the left, a page number '6' in the center, and the date '22/09/15' on the right.

- \* In any programming language, you need to know
  - \* Data types – what types are available to you, and what rules do they play by
  - \* How logical terms (usually true and false) are represented (a tricky one in some languages)
  - \* How statements are recognised by the language machinery
  - \* How to group statements together (for loops, if.. etc.)
  - \* The rules of scope (which statements can access a bit of code)
  - \* How subroutines are created and used
    - \* especially how to pass values (parameters) in and out

Copyright © A McMonnies, UWS, School of Computing 6 22/09/15

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.



# JavaScript (JS)

- \* Data types are 'number', 'string' and 'object'
  - \* Also 'function', but that's a special case here
  - \* Variables are bound to values, not types
    - \* which means a variable can change type mid-program
- \* Logical values:
  - \* a logical expression (a comparison) returns true or false
  - \* JS tries too hard here:
    - \* `2 == 1+1` // true
    - \* `2 == '2'` // true (coercion is used to compare strings)
    - \* `2 === '2'` // false (because types are also compared)
  - \* Best to always use `===` for equality and `!==` for inequality.

Copyright © A McMonnies, UWS, School of Computing 7 22/09/15

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

1. Create variables and then check using `typeof`:

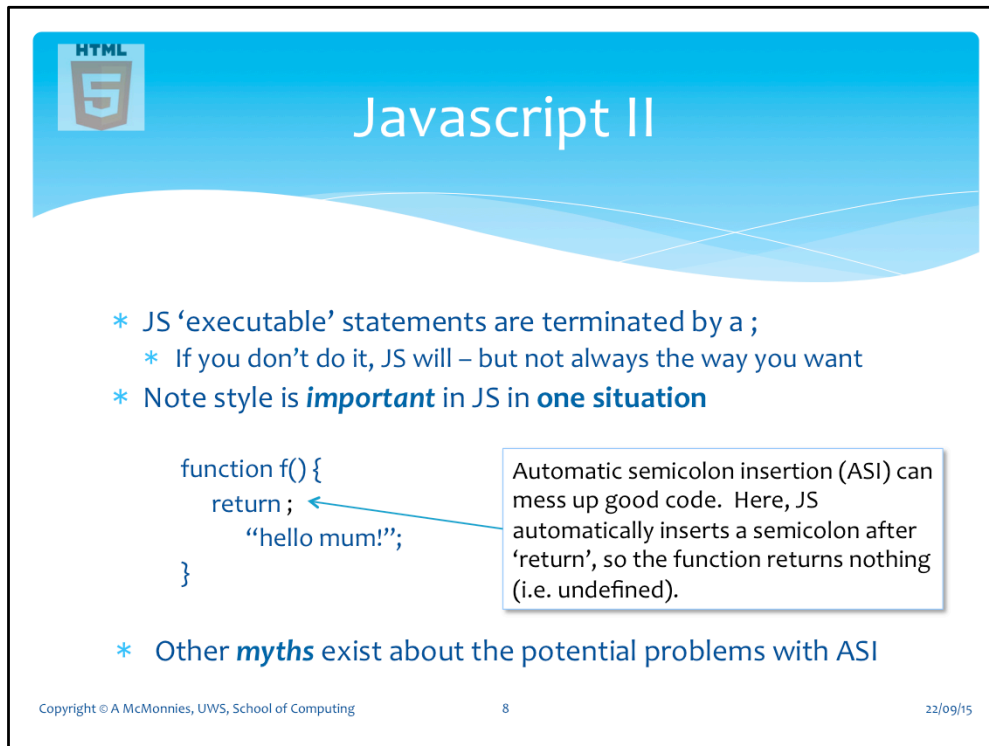
```
var v="12345";      typeof v;
v = 3;              typeof v;
v = function f(){return 0;}  typeof v
```

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
    };
}
```



## HTML5 Javascript II

- \* JS 'executable' statements are terminated by a ;
- \* If you don't do it, JS will – but not always the way you want
- \* Note style is **important** in JS in **one situation**

```
function f() {  
    return ;  
    "hello mum!";  
}
```

Automatic semicolon insertion (ASI) can mess up good code. Here, JS automatically inserts a semicolon after 'return', so the function returns nothing (i.e. undefined).

- \* Other **myths** exist about the potential problems with ASI

Copyright © A McMonnies, UWS, School of Computing 8 22/09/15


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.





## Executable and Control statements

- \* Executable statements are statements that do something
  - \* Assignment – e.g. `x = 3;`
  - \* Subroutine-call – e.g. `console.log("x = " + x);`
  - \* A bit of both – e.g. `y = Math.sqrt(x);`
- \* Control statements *modify* the execution of other statements, for...
  - \* Selection – e.g. `if(x > 0) y = Math.sqrt(x);`
  - \* Repetition – e.g. `while(x > 1) x /= 2;`
- \* Control statements are not terminated with a semicolon

Copyright © A McMonnies, UWS, School of Computing 9 22/09/15

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:

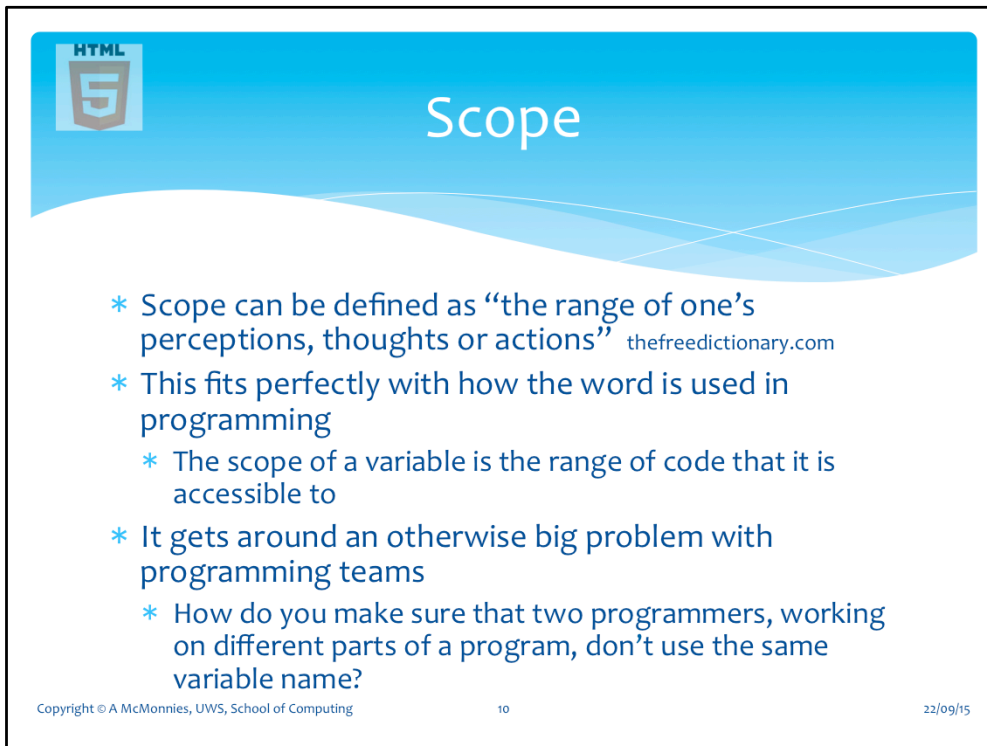
```

    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.




The slide features a blue header with the word "Scope" in white. To the left of the title is a logo consisting of a shield with the number "5" inside, and the word "HTML" above it. Below the header, there is a list of four bullet points in blue text. At the bottom of the slide, there is a small copyright notice on the left, the number "10" in the center, and the date "22/09/15" on the right.

- \* Scope can be defined as “the range of one’s perceptions, thoughts or actions” thefreedictionary.com
- \* This fits perfectly with how the word is used in programming
  - \* The scope of a variable is the range of code that it is accessible to
- \* It gets around an otherwise big problem with programming teams
  - \* How do you make sure that two programmers, working on different parts of a program, don’t use the same variable name?

Copyright © A McMonnies, UWS, School of Computing 10 22/09/15

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 quite 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.



## Javascript scope rules

- \* Global scope:
  - \* This applies to variables created outside functions that are not part of an object
- \* Local scope
  - \* This applies to variables created inside functions
- \* Object scope
  - \* The properties and methods of an object
  - \* Less of an issue, because of the access syntax  
Object.property;

```
var x = 2, y=3;


function f(){
  var x = 5;    // different
  y = 7;        // same changes y
}

f();
// Here, x is 2 and y is 7.
```

Copyright © A McMonnies, UWS, School of Computing 11 22/09/15

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.



# Functions

\* Four styles of function declaration

1. Declaration as usual
2. Function Expression
3. Anonymous function expression
4. Function constructor

```
function cube(n){ // 1
    return n*n*n;
}

var cube2 = function cube(n){ // 2
    return n*n*n;
}


var cube3 = function(n){ // 3
    return n*n*n;
}

var cube4 = new Function("n", // 4
    "return n*n*n");
```

Copyright © A McMonnies, UWS, School of Computing 12 22/09/15

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.



## Function parameters

- \* A parameter is a variable declaration in the function interface
  - \* i.e. a variable that can be passed from outside the function to inside it
- \* In a function call, function **arguments** are assigned to parameters just before the function executes
  - \* Values or variables are passed in to the function for it to work on


```
function sayHello(name){  
    alert("Hello " + name);  
}  
  
// call this...  
sayHello("Joe"); // "Hello Joe"  
var n = "Fred";  
sayHello(n);      // "Hello Fred"  
sayHello( prompt("Enter your name"));  
                // "Hello <name entered>"
```

Copyright © A McMonnies, UWS, School of Computing

13

22/09/15

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.



## Function “arguments”

- \* All JS functions have an internal “arguments” property
  - \* An array of the values passed during a call
  - \* This is useful for generalizing functions
    - \* e.g. args can be optional provided the function checks them before use
- \* You can pass more arguments than the function has parameters
  - \* The extra ones are ignored
- \* If fewer are likely to be passed, you should check them (compare to undefined)

```
function addAll(){ // note – no parameters
  var total = 0;
  for(var i=0, j=arguments.length; i<j; i++){
    total += arguments[i];
  }
  return total;
}

addAll();           // 0
addAll(1, 2, 3);    // 6
addAll(2, 2, 2, 2); // 10
```

Copyright © A McMonnies, UWS, School of Computing 14 22/09/15


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?");
  }
}
```



# Objects

- \* In Javascript *everything* is an object
  - \* `3.14159268.toFixed(2)` // 3.14
  - \* `"abc".length` // 3
- \* The small range of types that are native to JS (number, string, boolean, object, function) can be easily extended by adding new types
  - \* Any new type will display "object" as its type
- \* Existing types can be extended by adding properties and methods (i.e. member variables and functions)

Copyright © A McMonnies, UWS, School of Computing


15

22/09/15

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.



## New Object Types

- \* Note – in Java/C++ etc., new object types are “classes”
  - \* Javascript does not have classes
    - \* Best described as a “prototypical object-oriented language”
- \* Several ways to create a new object
  - \* Literal syntax
  - \* Empty object + added properties and methods
  - \* Constructor function
- \* A constructor is best as consistency is desired
- \* Lots more next week.

```
var aPerson = {  
  name: "Fred",  
  age: 25;  
};  
  
var bPerson = {};  
bPerson.name = "Joe";  
bPerson.age = 30;  
  
function Person(name, age){  
  this.name = name;  
  this.age = age;  
}  
cPerson = new Person("Jane", 26);
```

Copyright © A McMonnies, UWS, School of Computing

16

22/09/15

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