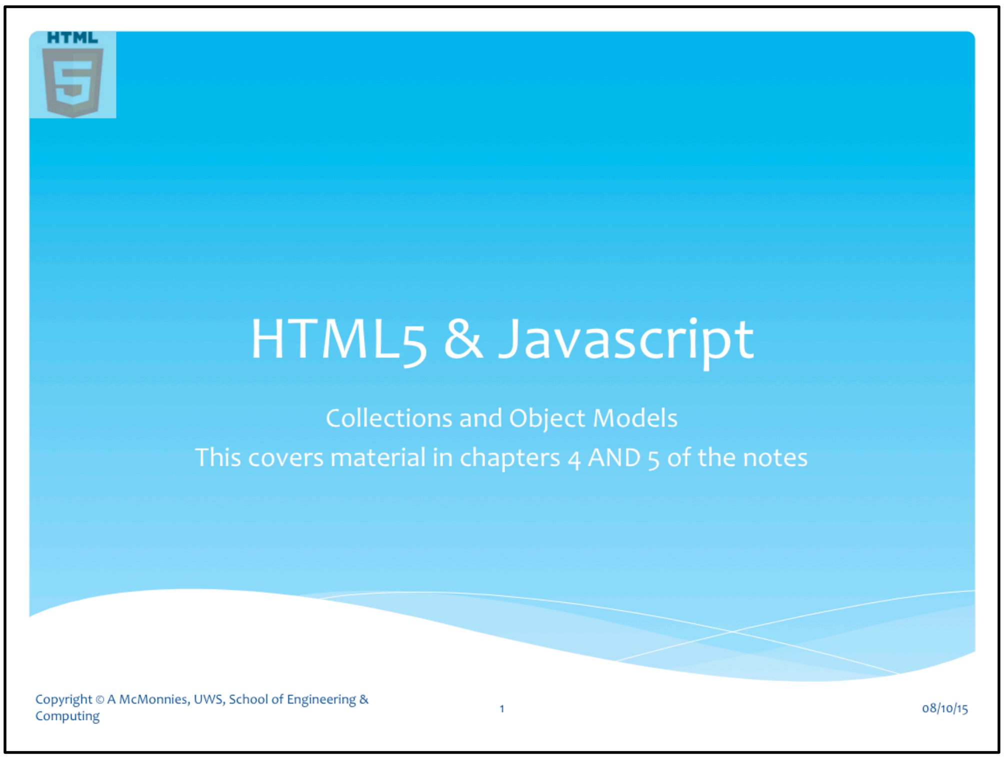
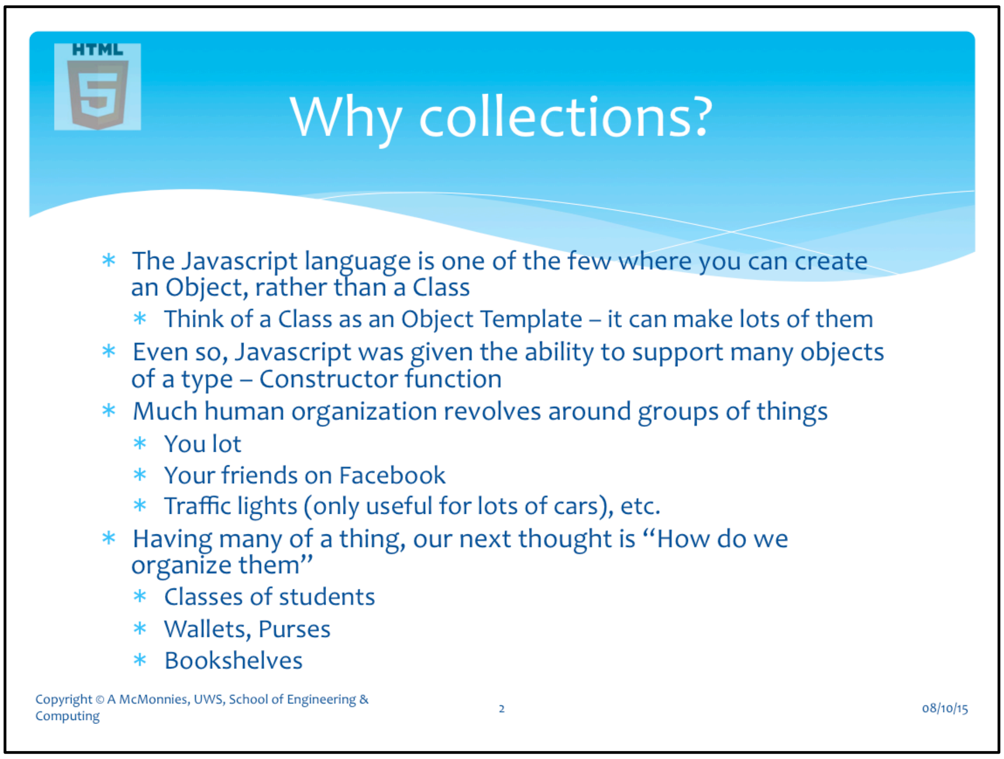


HTML5 & Javascript

Collections and Object Models
This covers material in chapters 4 AND 5 of the notes

1 08/10/15

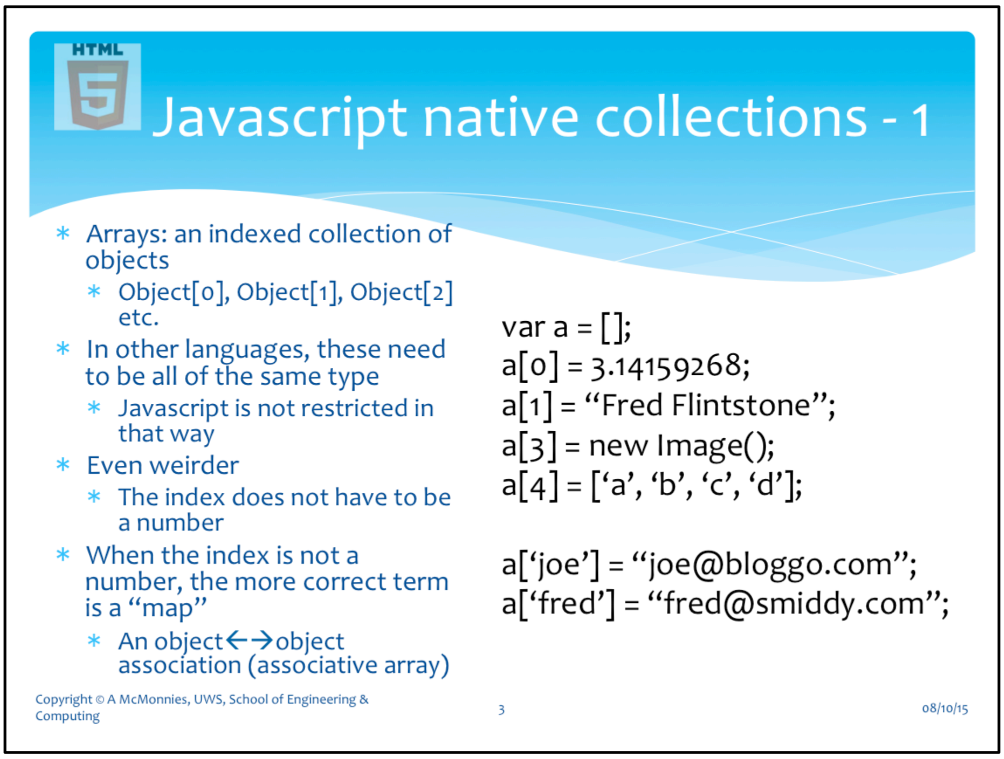
And so it begins….



Why collections?

- The Javascript language is one of the few where you can create an Object, rather than a Class
  - Think of a Class as an Object Template – it can make lots of them
- Even so, Javascript was given the ability to support many objects of a type – Constructor function
- Much human organization revolves around groups of things
  - You lot
  - Your friends on Facebook
  - Traffic lights (only useful for lots of cars), etc.
- Having many of a thing, our next thought is "How do we organize them"
  - Classes of students
  - Wallets, Purses
  - Bookshelves

Basically explaining the usefulness of collections, particularly in conjunction with classification.
Real-word examples might clarify things. E.g. bring a book along and show how the TOC and index let you find content without having to read the whole thing (obviously don't use a novel). Ask students to think of example of collections that they might have on their person (once they get to their phone, it might be difficult to stop them).



Javascript native collections - 1

- Arrays: an indexed collection of objects
  - Object[0], Object[1], Object[2] etc.
- In other languages, these need to be all of the same type
  - Javascript is not restricted in that way
- Even weirder
  - The index does not have to be a number
- When the index is not a number, the more correct term is a "map"
  - An object←→object association (associative array)

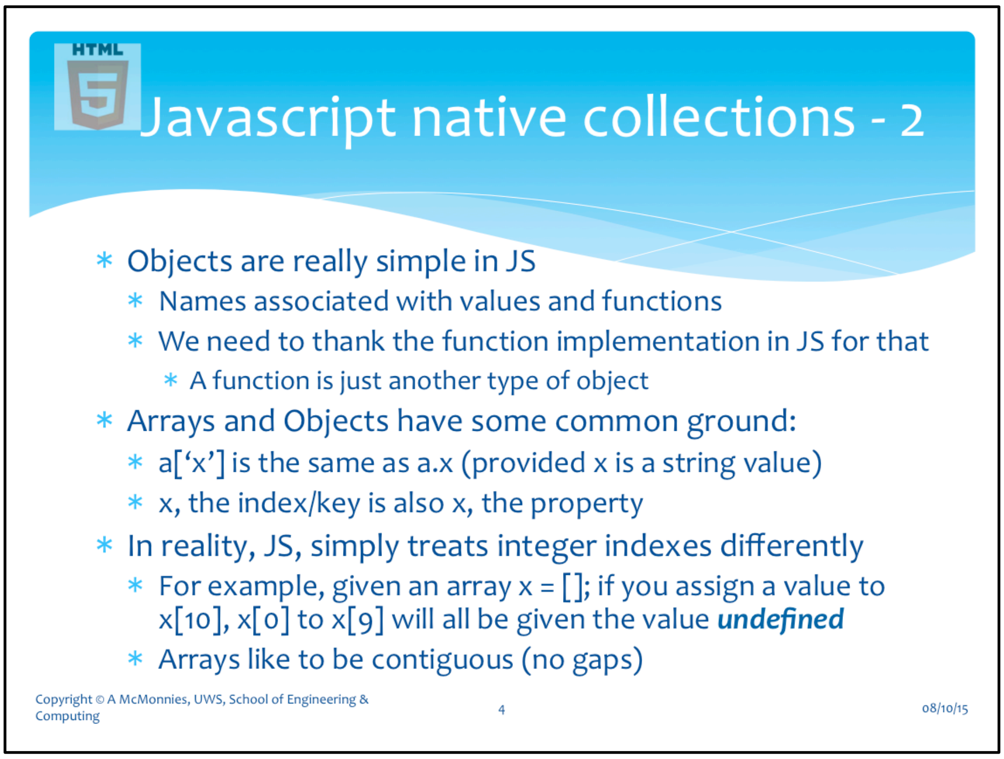
```
var a = [];
a[0] = 3.14159268;
a[1] = "Fred Flintstone";
a[3] = new Image();
a[4] = ['a', 'b', 'c', 'd'];

a['joe'] = "joe@bloggo.com";
a['fred'] = "fred@smiddy.com";
```

Copyright © A McMonnies, UWS, School of Engineering & Computing 3 08/10/15

It is worth discussing the pros and cons of mixed type arrays. The initial view of it could be 'wow – what power', but very quickly the cons become apparent – "I don't know what type of thing a[22] is!". Mono-type arrays may be boring, but for a programmer that just means predictable.

You could argue that all of object oriented programming is about creating mixed arrays - in C++ and Java, go we to a lot of trouble (VTables in C++, class hierarchies in Java) so that we can create truly polymorphic arrays) to provide this facility, when it's there for free in JS.

IMHO, the correct approach is to consider maps (which are just objects in JS) to be a different type of thing from an array. The language syntax gives us a way to separate arrays from maps (dot notation replaces index notation). Use arrays for array jobs (where possible sticking to a known type) and maps for map jobs (dictionaries, hashtables etc.). We'll look at a way of enforcing type with a custom collection later.



Javascript native collections - 2

- Objects are really simple in JS
  - Names associated with values and functions
  - We need to thank the function implementation in JS for that
    - A function is just another type of object
- Arrays and Objects have some common ground:
  - a['x'] is the same as a.x (provided x is a string value)
  - x, the index/key is also x, the property
- In reality, JS, simply treats integer indexes differently
  - For example, given an array x = []; if you assign a value to x[10], x[0] to x[9] will all be given the value ***undefined***
  - Arrays like to be contiguous (no gaps)

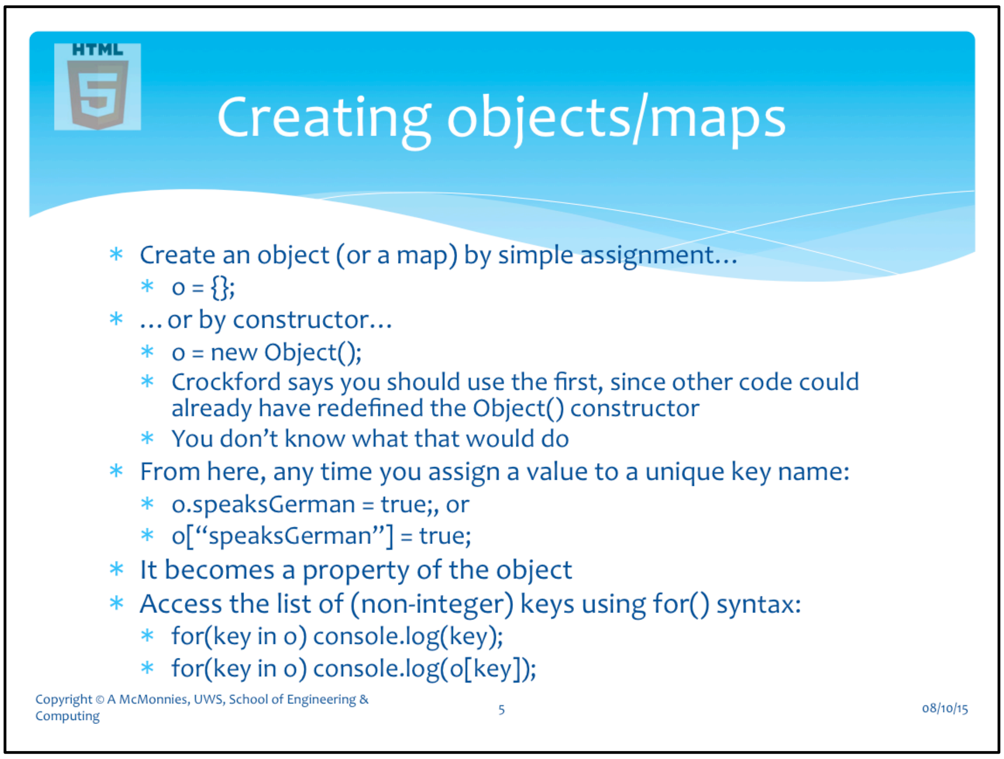
4 08/10/15

It is worth labouring the point about arrays with integer indexes – they are treated differently, and can coexist with non-integer index. For example (useful to show this in Console:

```
a = [];
a['fred'] = "fred@bloggo.com";
a["joe'] = "joe@smiddy.com";
a[0] = 100;
a[5] = 160;
```

a will now return [100, undefined, undefined, undefined, undefined, 160]. joe and fred are invisible, but if you try a.fred or a.joe you'll get the values back.
try: for(key in a) console.log(key); and you get all of the keys that values were attached to (i.e. 0, 5, fred & joe).
try: for (key in a) console.log(a[key); you'll get the four values back (100, 160, fred@smiddy.com, joe@bloggo.com)

I find this deeply weird, and although there is obviously a logic to it, it is not one that is easy to spell out. I can only assume that under the hood, an array is both an array and a map, with array elements assigned to integer keys and ALL elements mapped to specifically assigned keys.

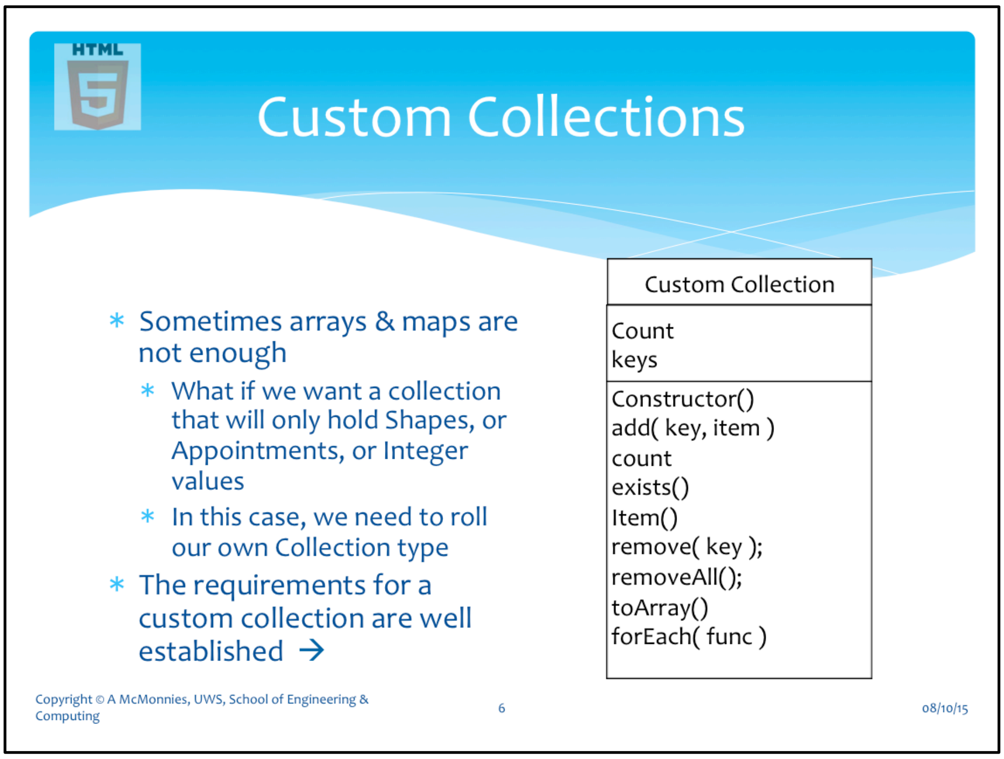


Creating objects/maps

- Create an object (or a map) by simple assignment…
  - o = {};
- …or by constructor…
  - o = new Object();
  - Crockford says you should use the first, since other code could already have redefined the Object() constructor
  - You don't know what that would do
- From here, any time you assign a value to a unique key name:
  - o.speaksGerman = true;, or
  - o["speaksGerman"] = true;
- It becomes a property of the object
- Access the list of (non-integer) keys using for() syntax:
  - for(key in o) console.log(key);
  - for(key in o) console.log(o[key]);

Copyright © A McMonnies, UWS, School of Engineering & Computing 5 08/10/15

The most different thing here is the use of the for(key in object) syntax. It is worth going through a couple of examples of this. If we were doing a *rigorous* job, we could add explanation of the need for the hasOwnProperty() method of Object() – I don't think it is necessary for this module, but a good understanding of why it is needed won't go amiss. See the Stefanov book for a good explanation.



Custom Collections

- Sometimes arrays & maps are not enough
  - What if we want a collection that will only hold Shapes, or Appointments, or Integer values
  - In this case, we need to roll our own Collection type
- The requirements for a custom collection are well established →

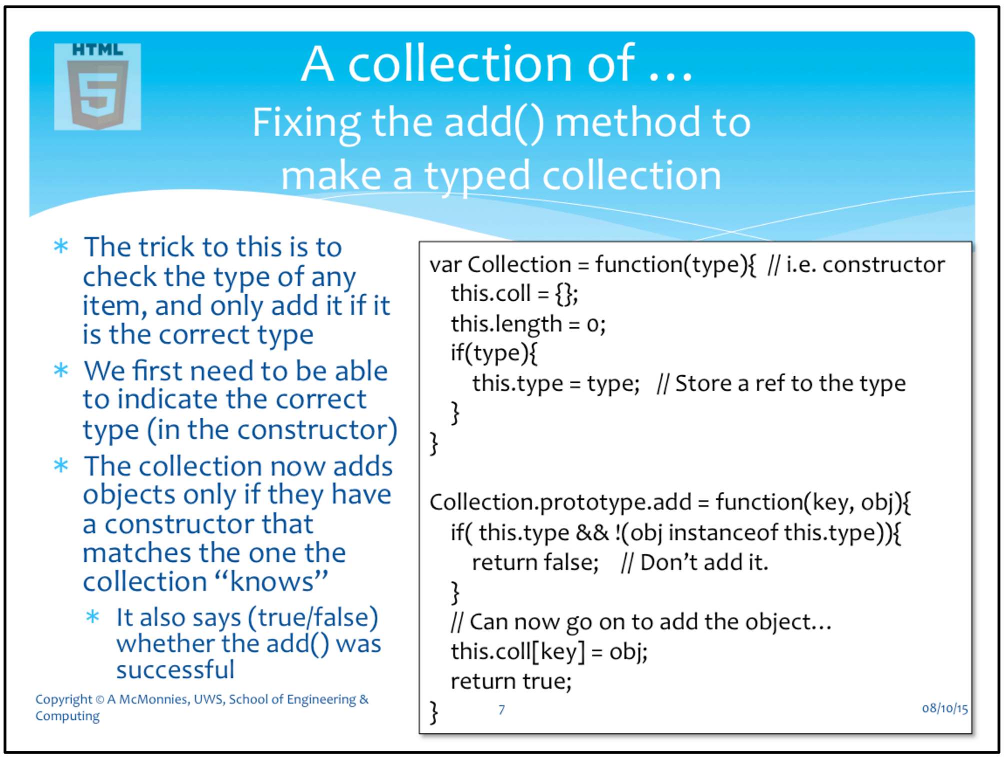
Custom Collection
Count keys

Constructor()
add(key, item)
count
exists()
Item()
remove(key);
removeAll();
toArray()
forEach(func)

Copyright © A McMonnies, UWS, School of Engineering & Computing 6 08/10/15

This is best explained with reference to the example code (in the notes chapter and as a demo project on Moodle. Download, unzip and open the project Collection and then open a console window. You'll see the results of several tests on the console. Ignore this and enter a sequence like this…

```
c = new Collection();
c.add('one', 1);
c.add('two', 2);
c.add('fred', 'fred@bloggo.com');
c.add('img', new Image());

c.keys();          // should show a list of keys

f = function( obj ) {
  console.log( obj.toString() );
};

c.forEach(f);              // Should list all objects on the console.
```



A collection of …

Fixing the add() method to make a typed collection

- The trick to this is to check the type of any item, and only add it if it is the correct type
- We first need to be able to indicate the correct type (in the constructor)
- The collection now adds objects only if they have a constructor that matches the one the collection "knows"
  - It also says (true/false) whether the add() was successful

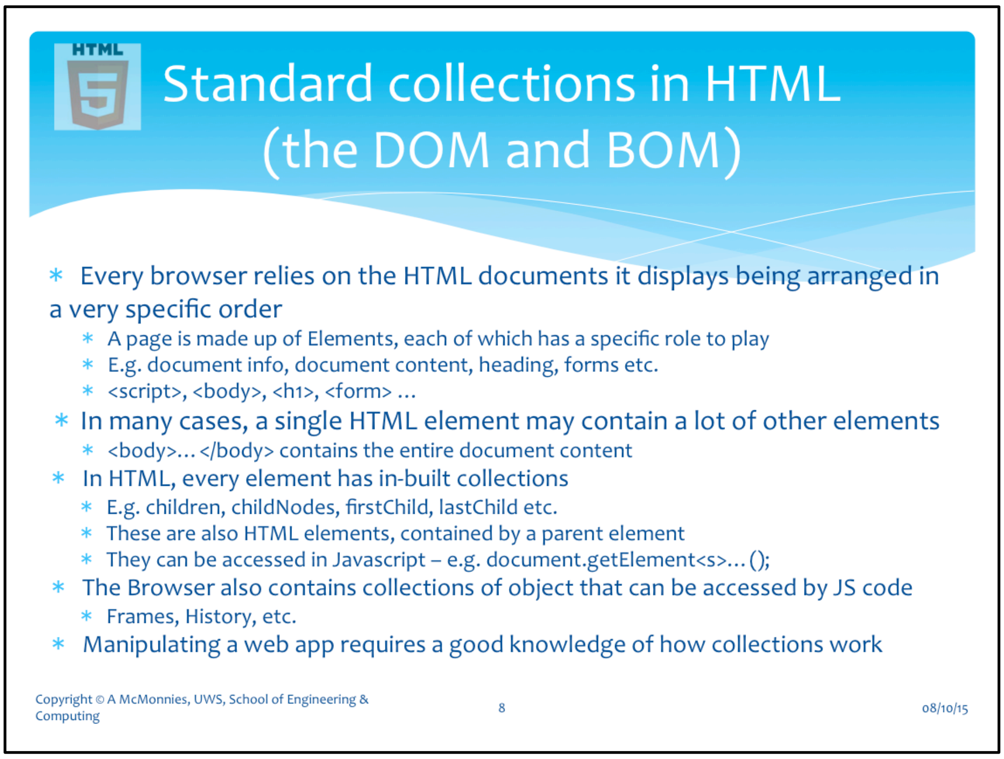
```
var Collection = function(type){  // i.e. constructor
   this.coll = {};
   this.length = 0;
   if(type){
      this.type = type;   // Store a ref to the type
   }
}

Collection.prototype.add = function(key, obj){
   if( this.type && !(obj instanceof this.type)){
      return false;    // Don't add it.
   }
   // Can now go on to add the object…
   this.coll[key] = obj;
   return true;
}
```

7 08/10/15

There are loads of ways to implement this – this version is simpler than testing the constructor to see if it matches, and better than testing by using typeof. typeof works for a specific type (e.g. testing if an object is an Image), but does not detect derived types. For example, if we have a Shape type and a Rectangle type that inherits from it, then we would usually like to be able to add a Rectangle to a collection of Shape objects – the Is-A principle again.

The big issue about this is - why do it? If you've come from a Java, .NET or C++ background, you're already used to the type-protection that these class-based languages provide. Try to add a dog to a collection of cats and you get a compile-time error. We can't do this in JS, so we need instead to test for these types of errors. Testing is easier if you have a collection of a set type, since trying to add the wrong type will give a false return.

It is worth pointing out the true == success pattern in this code. Explain that the return can be ignored, but provides a nice safety net if you feel the need.



Standard collections in HTML (the DOM and BOM)

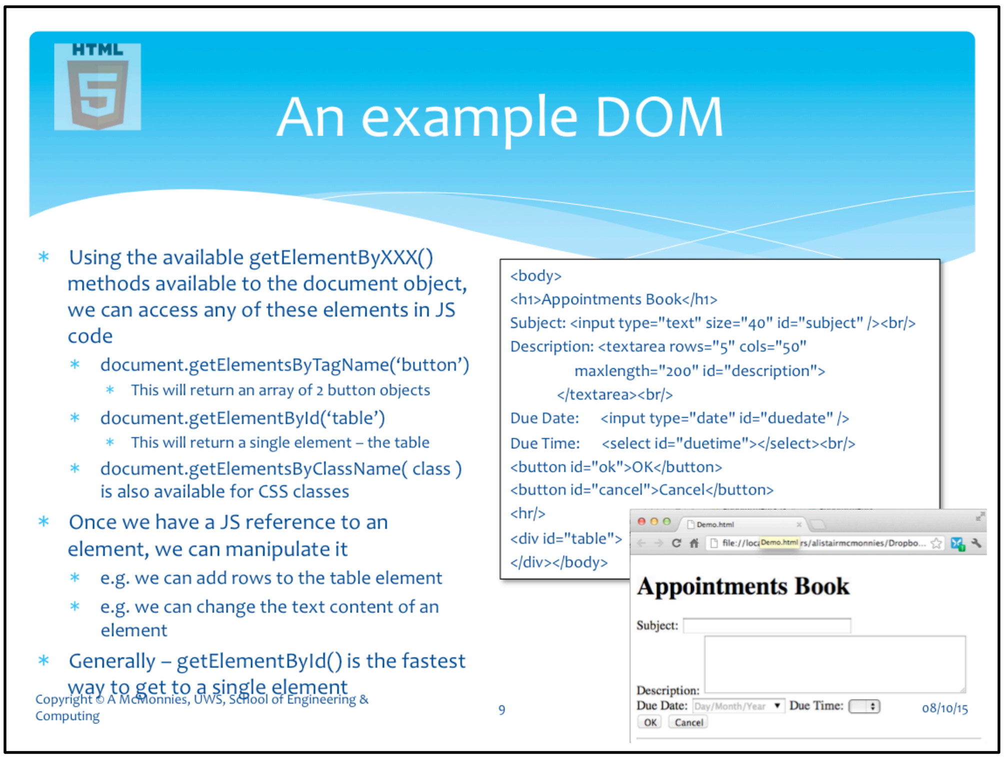
* Every browser relies on the HTML documents it displays being arranged in a very specific order
  * A page is made up of Elements, each of which has a specific role to play
  * E.g. document info, document content, heading, forms etc.
  * <script>, <body>, <h1>, <form> …
* In many cases, a single HTML element may contain a lot of other elements
  * <body>… </body> contains the entire document content
* In HTML, every element has in-built collections
  * E.g. children, childNodes, firstChild, lastChild etc.
  * These are also HTML elements, contained by a parent element
  * They can be accessed in Javascript – e.g. document.getElement<s>… ();
* The Browser also contains collections of object that can be accessed by JS code
  * Frames, History, etc.
* Manipulating a web app requires a good knowledge of how collections work

Copyright © A McMonnies, UWS, School of Engineering & Computing 8 08/10/15

This slide just points out that HTML is already awash with collections that you can access via Javascript code. There are several specialized ones (e.g. tables), and a generic “collection API” for the rest.

A good demo to the class would be to start with a fairly simple HTML document and add paragraphs and/or option elements to it via the Console. If you start with a HTML file with an empty <body> element, this would work…

body = document.getElementsByTagName(‘body’)[0];   // Note, the [0] at the end is because getElement***s***ByXXX returns an array
p = document.createElement(‘p’);
p.textContent = “Hello mum!”;   // Could also use innerHTML to same ends.
body.appendChild(p);
p2 = document.createElement(‘p’);
p2.textContent = “The Hulk says…”;
body.insertBefore(p2, p);



An example DOM

- Using the available getElementByXXX() methods available to the document object, we can access any of these elements in JS code
  - document.getElementsByTagName('button')
    - This will return an array of 2 button objects
  - document.getElementById('table')
    - This will return a single element – the table
  - document.getElementsByClassName(class) is also available for CSS classes
- Once we have a JS reference to an element, we can manipulate it
  - e.g. we can add rows to the table element
  - e.g. we can change the text content of an element
- Generally – getElementById() is the fastest way to get to a single element

```
<body>
<h1>Appointments Book</h1>
Subject: <input type="text" size="40" id="subject" /><br/>
Description: <textarea rows="5" cols="50"
          maxlength="200" id="description">
      </textarea><br/>
Due Date:    <input type="date" id="duedate" />
Due Time:    <select id="duetime"></select><br/>
<button id="ok">OK</button>
<button id="cancel">Cancel</button>
<hr/>
<div id="table">
</div></body>
```

Appointments Book

Subject:

Description:

Due Date: Day/Month/Year Due Time:

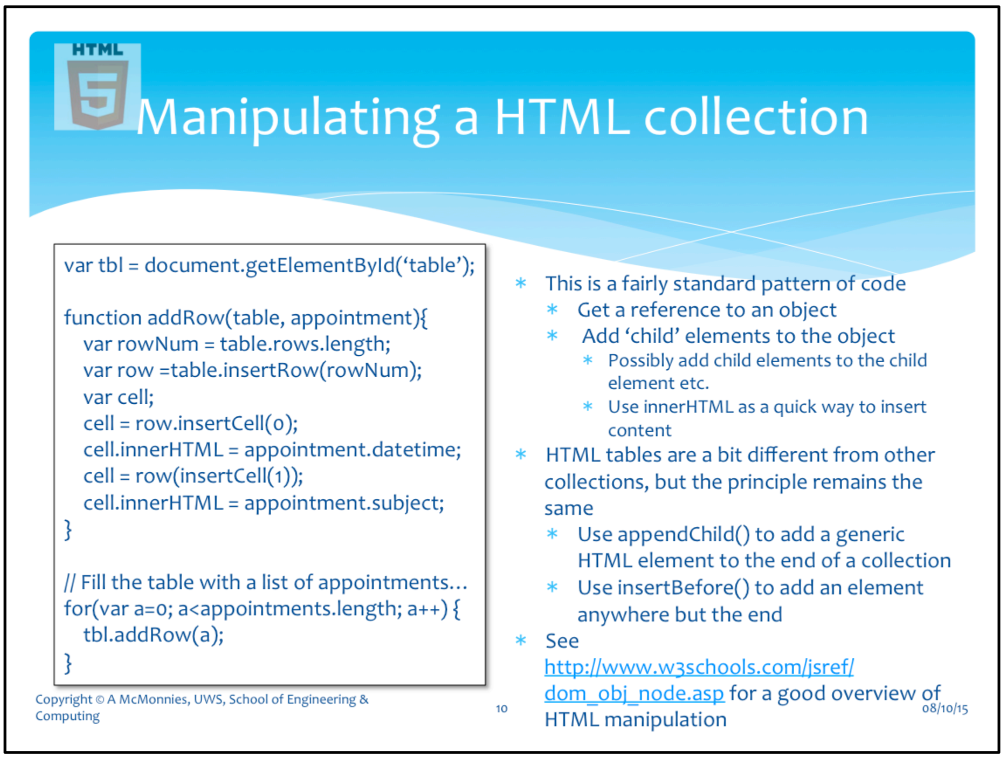
OK Cancel

Copyright © A McMonnies, UWS, School of Engineering & Computing 9 08/10/15

This slide is about one of the 'special' collections in HTML – tables have a different API to the other DOM collection manipulation stuff. The following slide covers some table manipulation in a bit of detail, so it is probably best to just gloss over this here.

It is worth emphasizing the different getElementBy… methods in document…
  getElementById() gets a single element (so no 's' after element)
  getElementsByTagName() gets an array of elements, even if there is only one, so the [0] is usually needed
  getElementsByClassName() is similar, but only useful if you're using CSS classes (although default HTML class names line 'h1' work
  getElementByName() does not appear in every browser – may be an IE thing, so I usually ignore it

If you need to add an element, the recommended approach is to use this sequence…
  el = document.createElement('<type>');
  el.textContent = "…" or el.innerText = "…"   or el.innerHTML = "…"
  parentElement.appendChild(el),   or parentElement.insertBefore(el, otherEl);



Manipulating a HTML collection

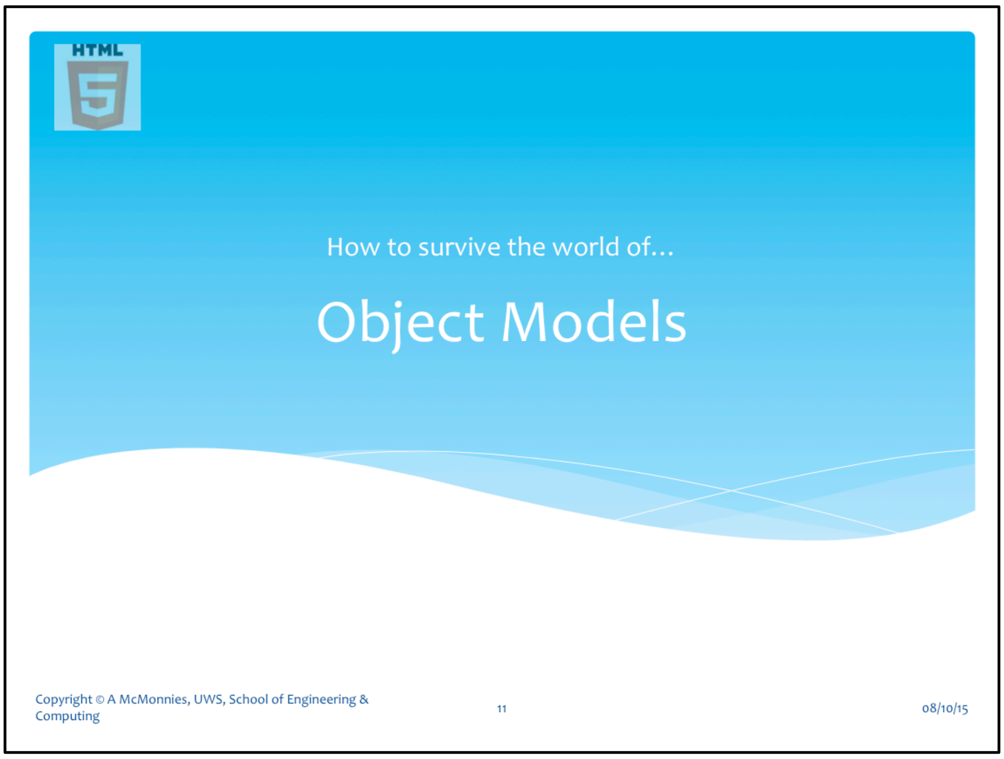
```
var tbl = document.getElementById('table');

function addRow(table, appointment){
   var rowNum = table.rows.length;
   var row =table.insertRow(rowNum);
   var cell;
   cell = row.insertCell(0);
   cell.innerHTML = appointment.datetime;
   cell = row(insertCell(1));
   cell.innerHTML = appointment.subject;
}

// Fill the table with a list of appointments…
for(var a=0; a<appointments.length; a++) {
   tbl.addRow(a);
}
```

- This is a fairly standard pattern of code
  - Get a reference to an object
  - Add 'child' elements to the object
    - Possibly add child elements to the child element etc.
    - Use innerHTML as a quick way to insert content
- HTML tables are a bit different from other collections, but the principle remains the same
  - Use appendChild() to add a generic HTML element to the end of a collection
  - Use insertBefore() to add an element anywhere but the end
- See http://www.w3schools.com/jsref/dom_obj_node.asp for a good overview of HTML manipulation

Copyright © A McMonnies, UWS, School of Engineering & Computing 10 08/10/15

As mentioned, this is a table-only pattern. It is useful because it indicates that having a table manipulation function is a worthwhile thing. Note though that working this way, if you wanted to update a whole table you would first have to remove all of its elements. In this event, I prefer to use the method shown in one of the lab programs (appointments – labs 2 & 3). Lab 2 builds an HTML table in text, and then replaces the entire table using the innerHTML property of the <div> that houses the table. There are pros and cons – you can't build a wrong table using the insertRow() and insertCell() methods (although you can still fail to add them, which you should really check for), while doing it the innerHTML way means that you could mess up the document structure and cause the page update to fail. On the other hand, you can check the HTML text as you build it up (printing it to Console for example) which can speed up debugging. You pays your money…

The URL is to a description of the more generic ways to manipulate HTML collecitons (appendChild(), insertBefore() etc.) – arguably this is more important, but it helps that the W3C page is one of their good ones.



How to survive the world of…

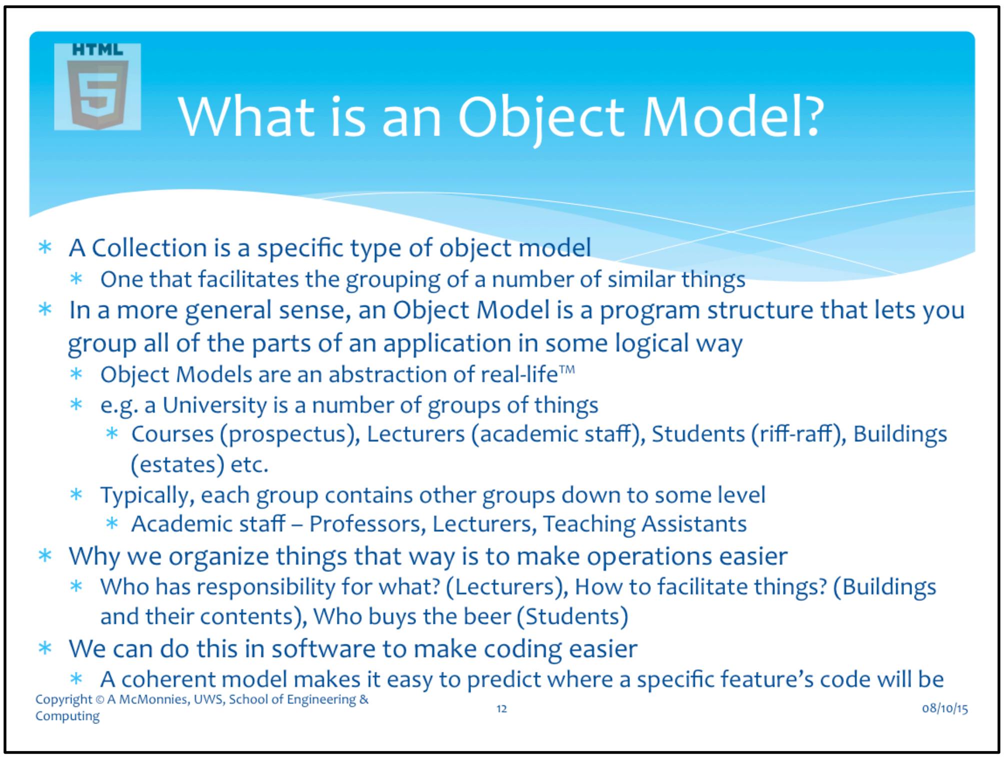
Object Models

Copyright © A McMonnies, UWS, School of Engineering & Computing

11

08/10/15

We're now on to chapter 5 of the notes



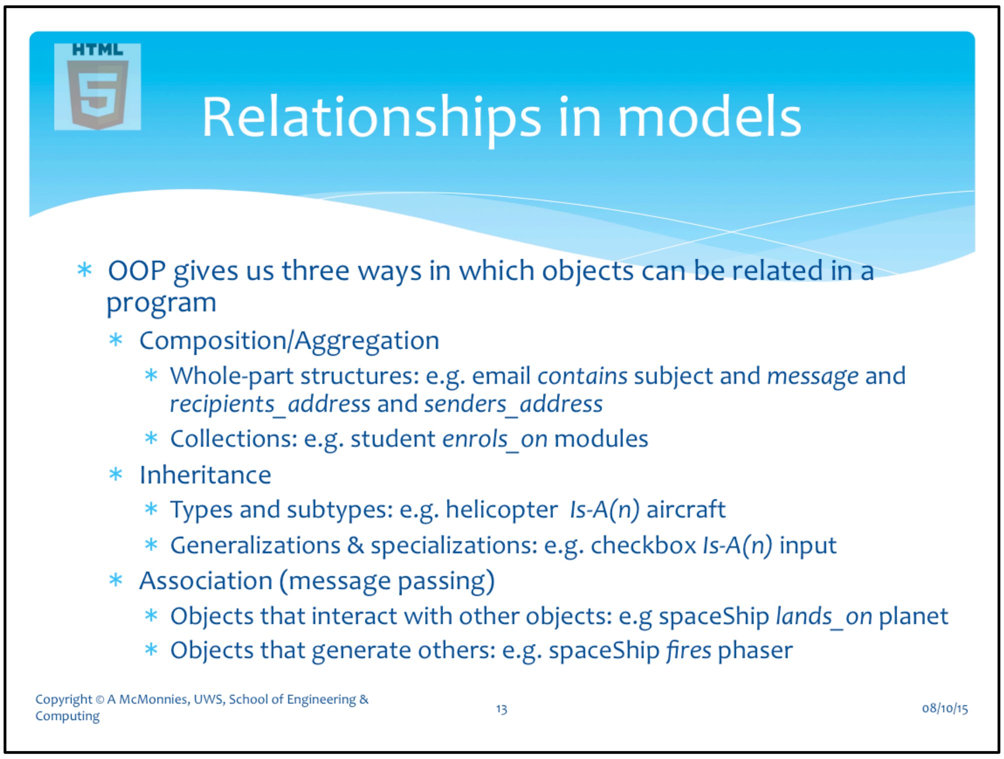
What is an Object Model?

- A Collection is a specific type of object model
  - One that facilitates the grouping of a number of similar things
- In a more general sense, an Object Model is a program structure that lets you group all of the parts of an application in some logical way
  - Object Models are an abstraction of real-life™
  - e.g. a University is a number of groups of things
    - Courses (prospectus), Lecturers (academic staff), Students (riff-raff), Buildings (estates) etc.
  - Typically, each group contains other groups down to some level
    - Academic staff – Professors, Lecturers, Teaching Assistants
- Why we organize things that way is to make operations easier
  - Who has responsibility for what? (Lecturers), How to facilitate things? (Buildings and their contents), Who buys the beer (Students)
- We can do this in software to make coding easier
  - A coherent model makes it easy to predict where a specific feature's code will be

Copyright © A McMonnies, UWS, School of Engineering & Computing 12 08/10/15

Object modelling can use both array and object syntax in JS. This slide is an attempt to provide some real-world context for what comes next – object modelling in code. Most inexperienced programmers have some difficulty in understanding why object modelling is important. The most useful explanation ought to start by explaining that modelling is something we don't think about in the real world (objects are components, and large objects are assemblies of smaller components). Try describing something familiar: e.g. the car example on the next slide.

There are other examples, more closely related to software, in the notes. e.g. the Drawing system, and some useful diagrams – BOM and DOM



Relationships in models

* OOP gives us three ways in which objects can be related in a program
  * Composition/Aggregation
    * Whole-part structures: e.g. email *contains* subject and *message* and *recipients_address* and *senders_address*
    * Collections: e.g. student *enrols_on* modules
  * Inheritance
    * Types and subtypes: e.g. helicopter *Is-A(n)* aircraft
    * Generalizations & specializations: e.g. checkbox *Is-A(n)* input
  * Association (message passing)
    * Objects that interact with other objects: e.g spaceShip *lands_on* planet
    * Objects that generate others: e.g. spaceShip *fires* phaser

Copyright © A McMonnies, UWS, School of Engineering & Computing 13 08/10/15

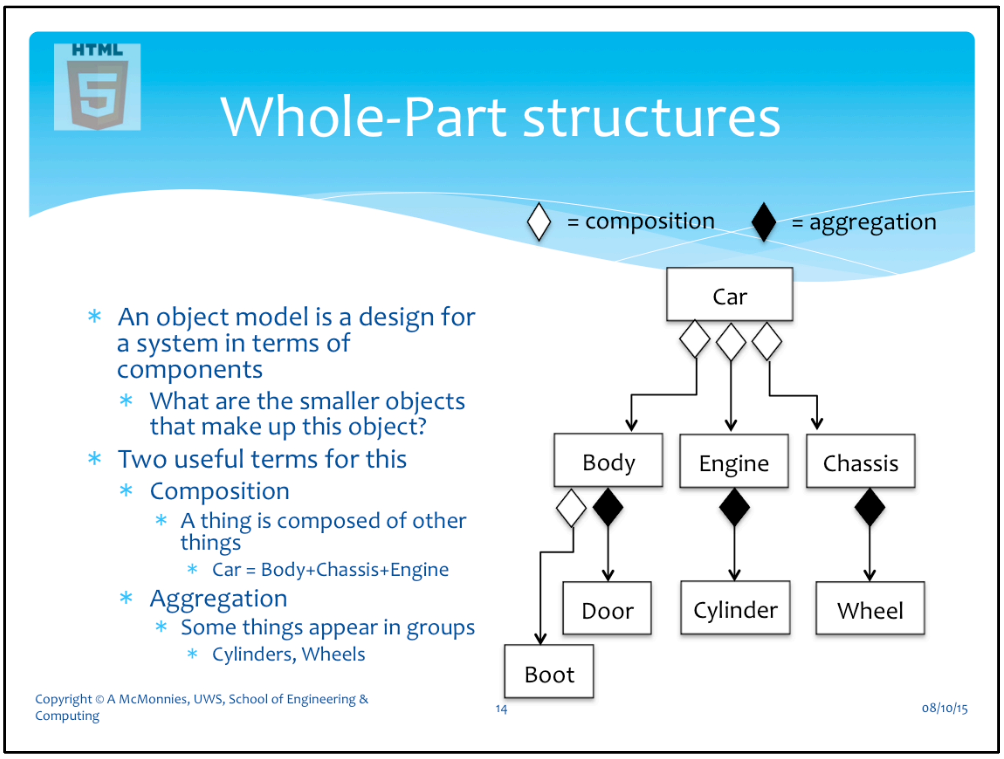
The obvious issue here is that we've already covered the three forms of relationship, although the Association one may not have been obvious. Providing examples will certainly help (thinking of good, but short, ones is a bit more problematic.

Composition:
This one is easy – most objects have multiple properties. E.g. The Appointment class has subject, datetime etc.

Aggregation:
This one isn't too hard either – the appointments collection has multiple Appointment instances. A more thematic one could be that a Contact can have multiple phone numbers and email addresses. Ask the students to look at the contacts app on their phone for confirmation

Inheritance:
We saw this with Shape: Rectangle, Shape: Line etc., so no need to elaborate further

Association:
More or less any line of code implements an association of some type, so the issue here is to find a less generic one. The onclick event of a button object is likely to form



Whole-Part structures

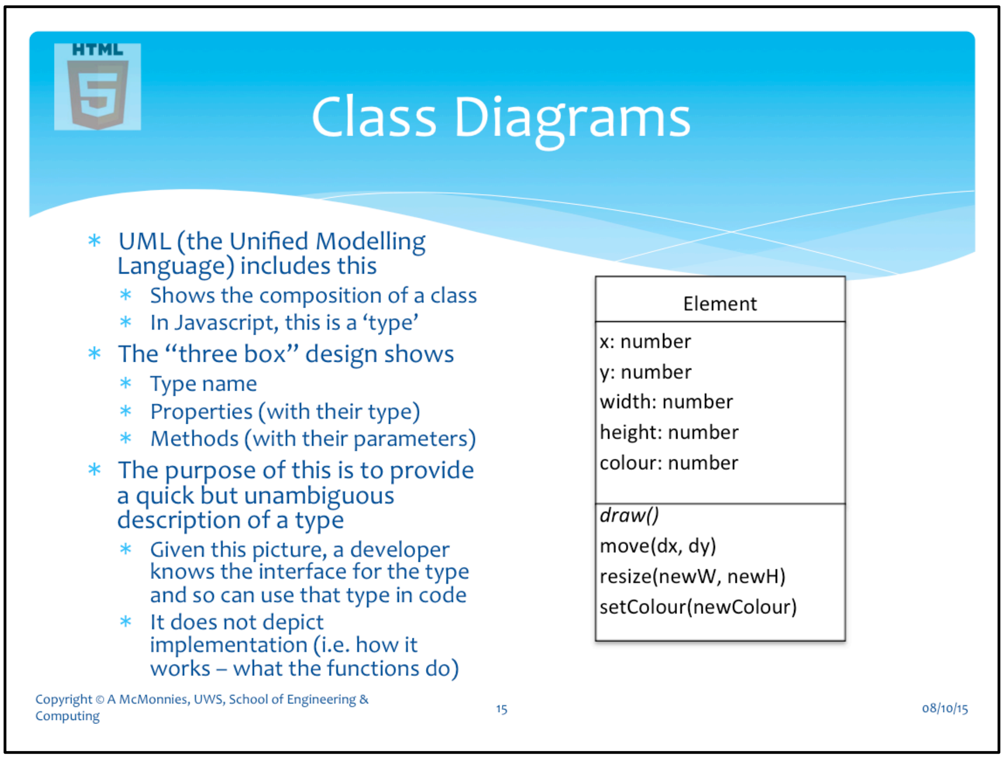
- An object model is a design for a system in terms of components
  - What are the smaller objects that make up this object?
- Two useful terms for this
  - Composition
    - A thing is composed of other things
      - Car = Body+Chassis+Engine
  - Aggregation
    - Some things appear in groups
      - Cylinders, Wheels

◇ = composition
◆ = aggregation
Car
Body
Engine
Chassis
Door
Cylinder
Wheel
Boot

14 08/10/15

There is some variation in the style of diagrams of this type – mostly to do with the distinction between composition & aggregation. Since both are instances of composition, that is to be expected. The main difference is t do with what's often called "…icity". The one shown here is the simplest variant, and shows composition as a one-to-one and aggregation as a one-to-many. Most software does a more precise job, showing a specific numerical relationship by putting a number at either end – typically 1 and the 'whole' end and either a specific number (e.g. 4 for wheels) or a range (0..n or 1..n) at the part end.

It is worth pointing out the options – perhaps even showing an example of software for systems design – visio, Enterprise Architect, StarUML (a good one since it's free) if you have one available. Don't suggest that this stuff is a core part of the module – it is here for completeness and to give additional information for those who are that interested.



Class Diagrams

* UML (the Unified Modelling Language) includes this
  * Shows the composition of a class
  * In Javascript, this is a 'type'
* The "three box" design shows
  * Type name
  * Properties (with their type)
  * Methods (with their parameters)
* The purpose of this is to provide a quick but unambiguous description of a type
  * Given this picture, a developer knows the interface for the type and so can use that type in code
  * It does not depict implementation (i.e. how it works – what the functions do)

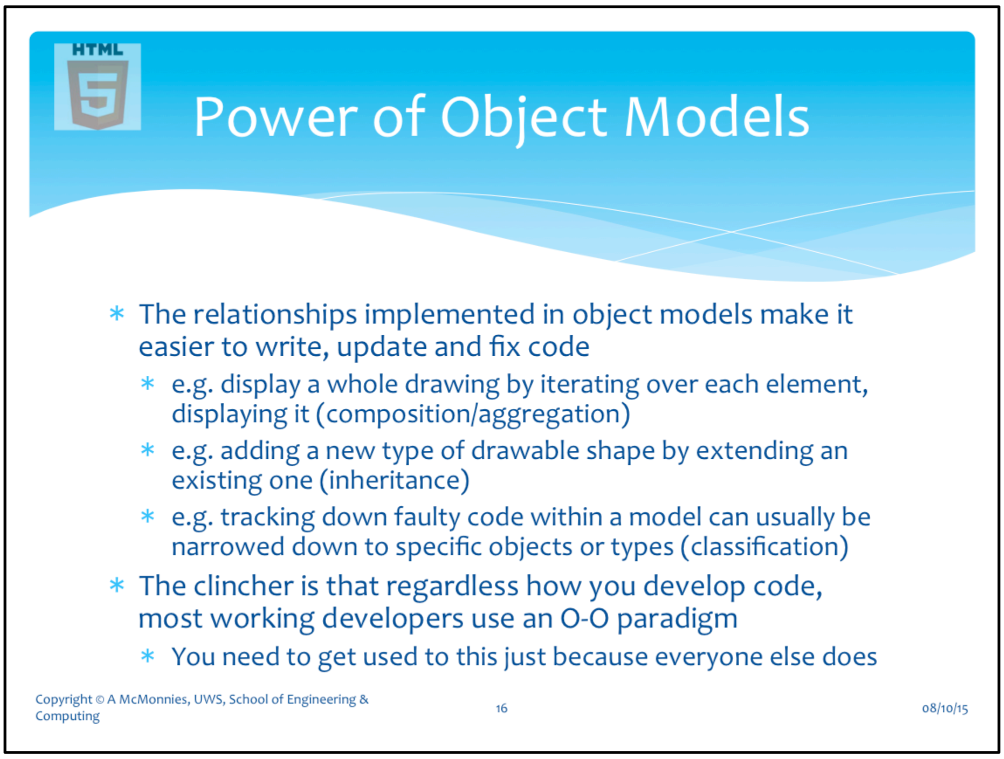
Element
x: number y: number width: number height: number colour: number

draw()
move(dx, dy)
resize(newW, newH)
setColour(newColour)

Copyright © A McMonnies, UWS, School of Engineering & Computing 15 08/10/15

Doing a couple of class boxes on the whiteboard is a good idea here. With a suitable example (e.g. the class hierarchy from the drawing app), you can show how a class box fits into a bigger class diagram (e.g. previous slide, the shape hierarchy etc.)

It is worth emphasizing the separation of this from the implementation, and why it is important. Interfaces are design elements, and while some languages support them completely (e.g. .NET, Java), they are always a strong link between a design and its implementation.



Power of Object Models

- The relationships implemented in object models make it easier to write, update and fix code
  - e.g. display a whole drawing by iterating over each element, displaying it (composition/aggregation)
  - e.g. adding a new type of drawable shape by extending an existing one (inheritance)
  - e.g. tracking down faulty code within a model can usually be narrowed down to specific objects or types (classification)
- The clincher is that regardless how you develop code, most working developers use an O-O paradigm
  - You need to get used to this just because everyone else does

16 08/10/15

By this stage, my guess is that either students are ok with the idea of objects, or they're not into programming. If they're not into programming, there is not much you can do apart from pointing out the inevitability of it given their chosen career path – no-one on this module can totally avoid program code, whether they write it or not. Recognising that now may be too late, but it is better than not recognising it at all.

It is worth reinforcing the notion of object models by going back to the developer tools for a browser and examining elements. The obvious structures – e.g. a <body> containing <p> elements – is mirrored in the Javascript object that accesses the <body> element (see slide 8)