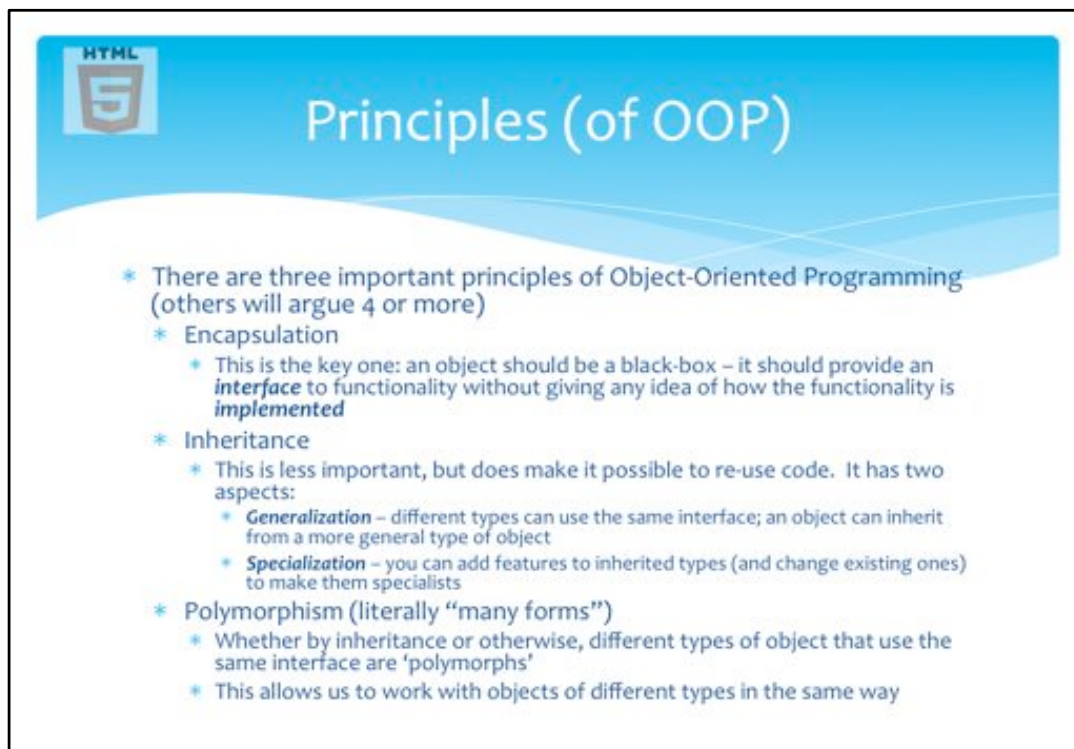




Ta-da!



## Principles (of OOP)

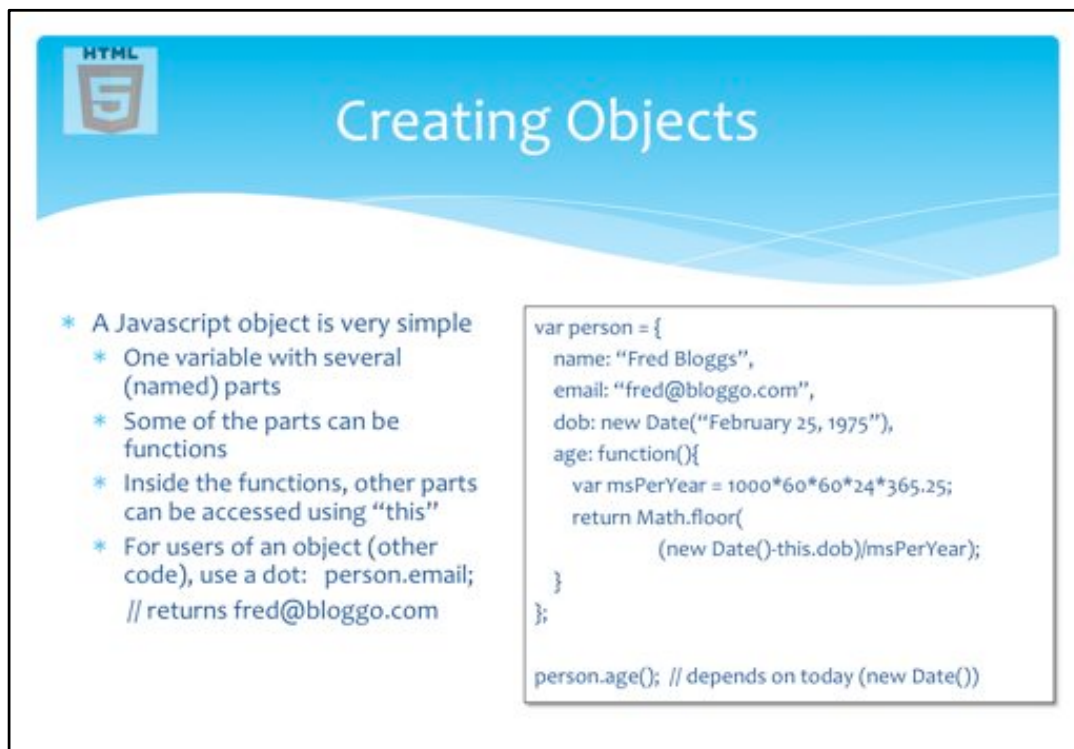
- \* There are three important principles of Object-Oriented Programming (others will argue 4 or more)
  - \* Encapsulation
    - \* This is the key one: an object should be a black-box – it should provide an **interface** to functionality without giving any idea of how the functionality is **implemented**
  - \* Inheritance
    - \* This is less important, but does make it possible to re-use code. It has two aspects:
      - \* **Generalization** – different types can use the same interface; an object can inherit from a more general type of object
      - \* **Specialization** – you can add features to inherited types (and change existing ones) to make them specialists
  - \* Polymorphism (literally “many forms”)
    - \* Whether by inheritance or otherwise, different types of object that use the same interface are ‘polymorphs’
    - \* This allows us to work with objects of different types in the same way

A lot of explaining is needed here – Interface, implementation, generalization, specialization, poly-whatsit. Providing an example of each of these concepts is probably the best approach, so...

“Others will argue 4 or more” – I’ve seen OOP described as having 4 core principles – encapsulation, inheritance, polymorphism and **\*\*operator overloading\*\*** - this is obviously oriented towards C++, but it is not a general OOP principle (which I take as the principles embodied in SmallTalk – more or less the prototype for everything since).

You’ll also see people suggesting Abstraction or Data Abstraction as a core principle – what do these guys think Encapsulation is about?? The only point of encapsulation is so that you can create an abstraction – public interface accessing private implementation. It seems to me that is similar to describing the three main principles of motor cars are Go-ey behaviour, an engine and a passenger cabin. The engine is the Go-ey thing. Then again I’m an opinionated sod.

Encapsulation – building an object as a black-box involves providing it with a number of functions (methods) that can be used to manipulate the object – either by retrieving data from inside it, or changing its data in some way. A good (real-world) example is a car dashboard. Instead of manipulating the various valves and levers directly, a driver uses the steering-wheel, gas pedal, gear-lever, brake pedal etc. to



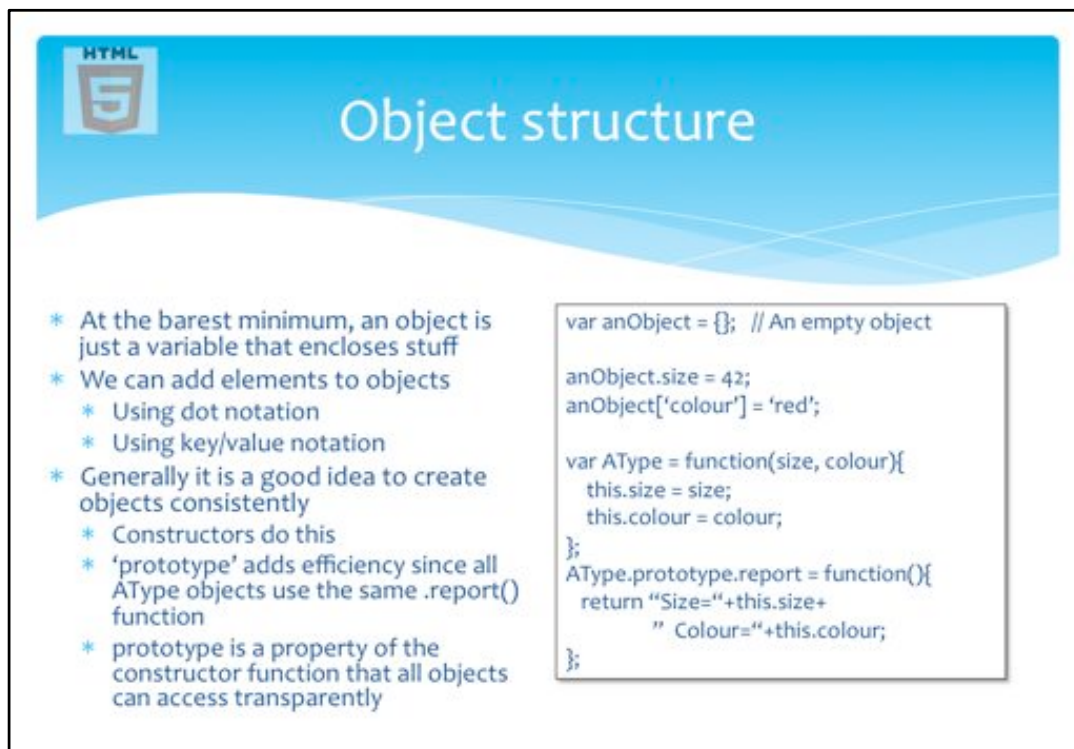
## Creating Objects

- \* A Javascript object is very simple
  - \* One variable with several (named) parts
  - \* Some of the parts can be functions
  - \* Inside the functions, other parts can be accessed using "this"
  - \* For users of an object (other code), use a dot: `person.email`;  
// returns `fred@bloggo.com`

```
var person = {  
  name: "Fred Bloggs",  
  email: "fred@bloggo.com",  
  dob: new Date("February 25, 1975"),  
  age: function(){  
    var msPerYear = 1000*60*60*24*365.25;  
    return Math.floor(  
      (new Date()-this.dob)/msPerYear);  
  }  
};  
  
person.age(); // depends on today (new Date())
```

It will be worth explaining various bits that are taken for granted in this slide.

`msPerYear` is a local variable (more on that later) used inside the function `Math.floor()` truncates a number (removes the digits after the d.p. `person` is a package of data and function code – this is very unlike Java, where the `person` class would be used to generate instances that contained data and **references to** function code. Javascript has the potential to be very inefficient, since adding a function to many object means having many copies of the function – storage inefficient and also a very good source of errors (many copies of a function makes it more likely there can be inconsistencies)



## Object structure

- \* At the barest minimum, an object is just a variable that encloses stuff
- \* We can add elements to objects
  - \* Using dot notation
  - \* Using key/value notation
- \* Generally it is a good idea to create objects consistently
  - \* Constructors do this
  - \* 'prototype' adds efficiency since all AType objects use the same .report() function
  - \* prototype is a property of the constructor function that all objects can access transparently

```
var anObject = {}; // An empty object

anObject.size = 42;
anObject['colour'] = 'red';

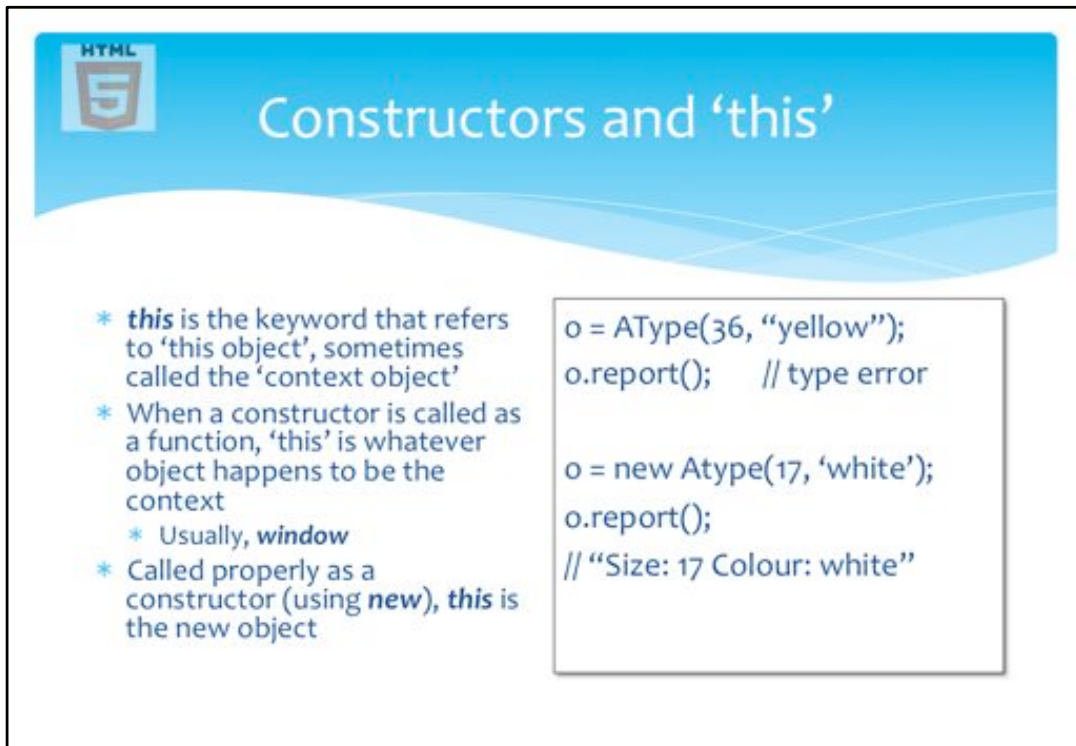
var AType = function(size, colour){
  this.size = size;
  this.colour = colour;
};
AType.prototype.report = function(){
  return "Size="+this.size+
    " Colour="+this.colour;
};
```

This slide is really about moving from objects to types of object.

It is worth explaining that for a situation where there will be many objects of a type this is a more consistent approach – all objects will be (to some degree) compatible. In the next slide, we'll be looking at the keyword 'new', so it is worth illustrating at this stage in a console window that using `o = AType(15, 'green');` does work, but does not have the effect you expect it to ...

```
o = AType(15, 'green'); // returns 'undefined'
                        // inside the constructor, 'this' is the current context object
                        // which is usually 'window' – test by typing window.colour in
the console
                        // o.report() gives a type error
```

```
o = new AType(22, 'blue'); // returns AType (an object reference)
                        // o.report() works.
```



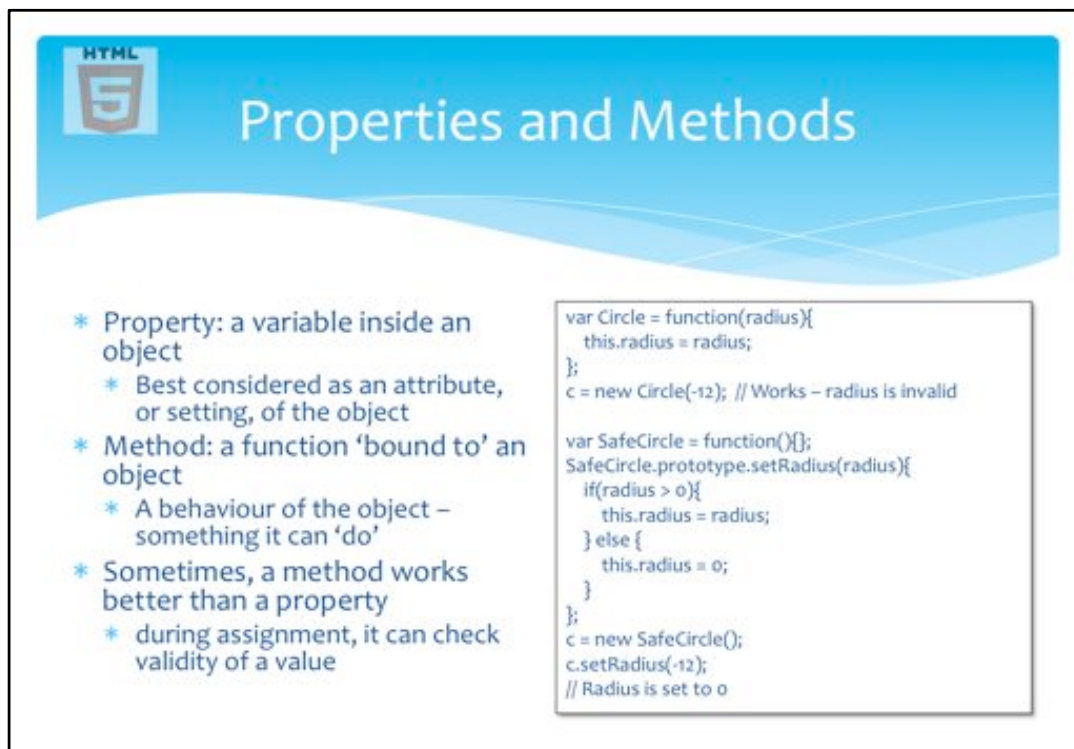
## Constructors and 'this'

- \* **this** is the keyword that refers to 'this object', sometimes called the 'context object'
- \* When a constructor is called as a function, 'this' is whatever object happens to be the context
  - \* Usually, **window**
- \* Called properly as a constructor (using **new**), **this** is the new object

```
o = AType(36, "yellow");  
o.report();    // type error  
  
o = new Atype(17, 'white');  
o.report();  
// "Size: 17 Colour: white"
```

As described in notes for the previous slide. It is well worth doing this in class using a console.

When you define the constructor and report function, you can either write all the code on one line (which looks awkward), or use ctrl+Enter to add new lines – the console will treat this like using Enter in an editor.



## Properties and Methods

- \* **Property:** a variable inside an object
  - \* Best considered as an attribute, or setting, of the object
- \* **Method:** a function 'bound to' an object
  - \* A behaviour of the object – something it can 'do'
- \* Sometimes, a method works better than a property
  - \* during assignment, it can check validity of a value

```

var Circle = function(radius){
  this.radius = radius;
};
c = new Circle(-12); // Works – radius is invalid

var SafeCircle = function(){
  SafeCircle.prototype.setRadius(radius){
    if(radius > 0){
      this.radius = radius;
    } else {
      this.radius = 0;
    }
  };
  c = new SafeCircle();
  c.setRadius(-12);
  // Radius is set to 0

```

Worth explaining that this style of programming needs more discipline – it is just as easy to change the `.radius` of a `SafeCircle` as to call its `.setRadius()` function. Explain the various strategies for getting around this problem (which is really the problem of not having a `Private` modifier for JS objects).

1. Mark private members in some way – most common one is one or two underscores at the start or end of a member's name. e.g.

```

var Circle = function(radius){
  this._radius = radius;
};

```

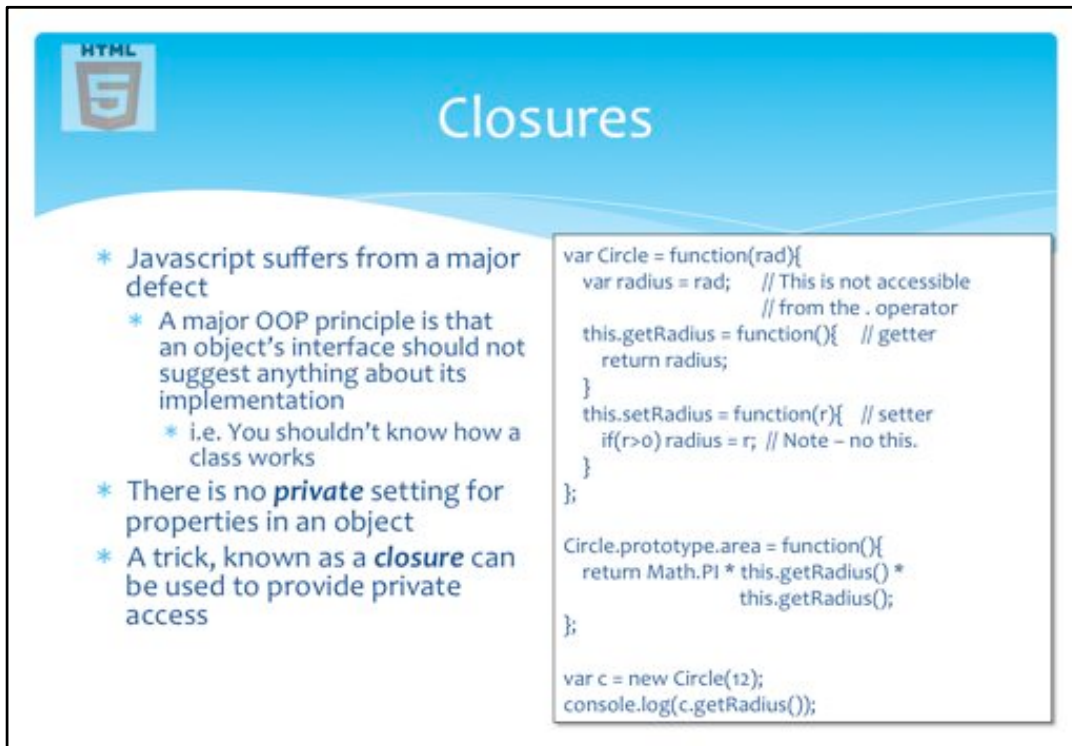
2. Use a closure. Define `getX` and `setX` functions within the constructor that access a local variable (declare with `var`) in the constructor:

```

var Circle = function(rad){
  var radius = rad;          // This is not an accessible member of a Circle
  this.getRadius = function(){ // But this is (and its prototype has a pointer to
    return radius;            radius
  }
}

```





## Closures


- \* Javascript suffers from a major defect
  - \* A major OOP principle is that an object's interface should not suggest anything about its implementation
    - \* i.e. You shouldn't know how a class works
  - \* There is no **private** setting for properties in an object
  - \* A trick, known as a **closure** can be used to provide private access

```
var Circle = function(rad){  
  var radius = rad; // This is not accessible  
                    // from the . operator  
  this.getRadius = function(){ // getter  
    return radius;  
  }  
  this.setRadius = function(r){ // setter  
    if(r>0) radius = r; // Note - no this.  
  }  
};  
  
Circle.prototype.area = function(){  
  return Math.PI * this.getRadius() *  
    this.getRadius();  
};  
  
var c = new Circle(12);  
console.log(c.getRadius());
```

It is worth spending a bit of time making the importance of this clear.

OOP's principle of encapsulation can not be implemented in the standard Javascript mechanisms. However, the trick of creating a closure makes private members (and therefore privileged members) possible.

The actual mechanism is due to an internal pointer to the members of a class within a function. In the Circle constructor, radius is a local variable. In Java or C, this would be an automatic variable, i.e. one that was created as the function was called and destroyed as the function exited. In JS, the inner functions which access radius keep a pointer to it. As a result, the radius variable stays alive outwith function calls and can still be accessed once the constructor call has exited.



## 'Classes' in Javascript

- \* Javascript is a **prototypical** object-oriented language
  - \* This has its own strengths
    - \* Flexibility, power
- \* Other OOP languages (e.g. Java) are Classical
  - \* Object types are strongly defined in classes
  - \* The major advantage of this is consistency
    - \* When you access an object, you know what you're going to get – **Strong Typing**
- \* We can mimic classical behaviour in Javascript
  - \* Constructor functions

```
var Person = function(name, email, bday){
    this.name = name;
    this.email = email;
    this.dob = bday;
};


var p1 = new Person("Joe",
    "joe@bloggo.com", "June 2, 1980");
var p2 = new Person("Fred",
    "fred@smiddy.com", "March 11, 1978");
// p1 & p2 are compatible
p2.jobDescription = "Programmer";
delete p2.email;
// Now they aren't
```

The main point being made here is that without the protection of a rigid class, it is not possible to always know what is in an object. If class-like behaviour is to be expected in Javascript, it is necessary to always follow certain rules – the main one being objects created from constructors should not be altered adversely. This does not mean giving up facilities like the ability to add new members and methods to an existing object, but members should never be replaced (at least not without some consideration).

Using a constructor to create a 'class' means accepting certain responsibilities along with the power.

Polymorphism depends on either classical behaviour (i.e. being able to predict that certain methods are available) or "duck-typing", which, for robustness, depends on checking a method exists before calling it.





## Prototypical Objects

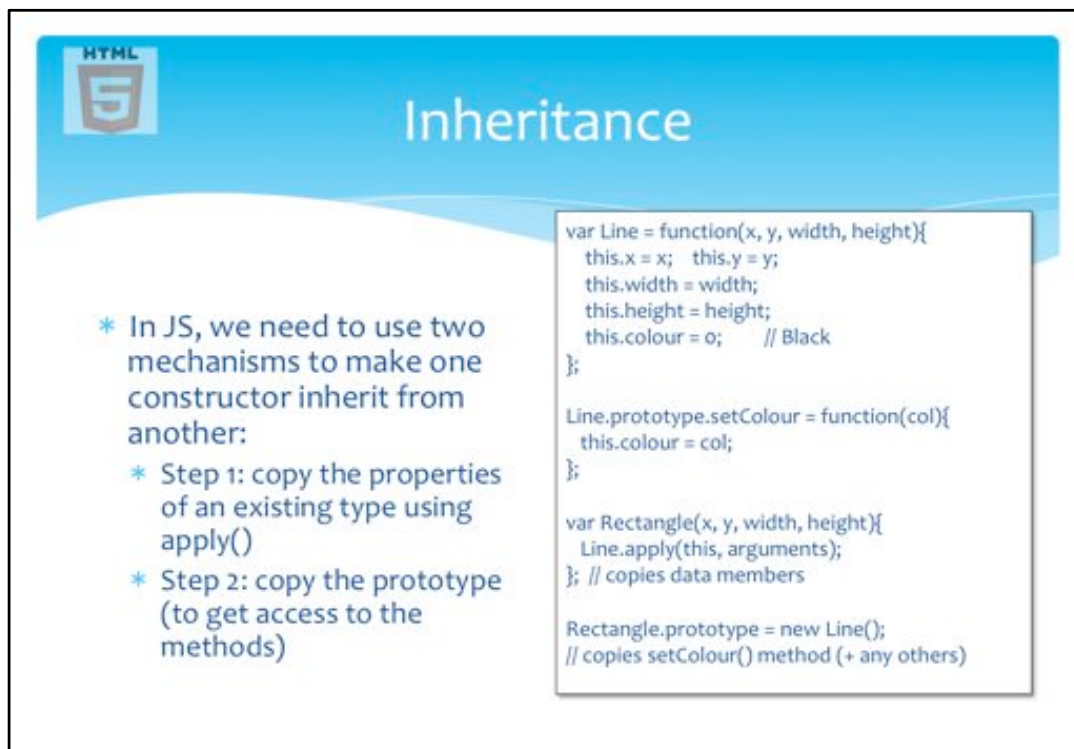
- \* The **prototype** member of an object provides access to the whole family
  - \* Existing objects can be “upgraded”
  - \* Care needs to be taken in doing this
    - \* e.g. changing a method could break existing code
- \* More usefully, a constructor's prototype can be used to provide a method to a whole set of objects in a single definition
  - \* This provides some of the power of a ‘class’ without sacrificing the flexibility of objects

```
String.prototype.times = function(count) {  
  if(count < 1) {  
    return "";  
  } else {  
    var s = "";  
    while(count) {  
      s += this;  
    }  
    return s;  
  }  
}  
  
"Hello".times(3); // "HelloHelloHello";
```

The prototype member provides access to the underlying definition of an object type through its Constructor function.

Generally, prototype is only useful for accessing and/or amending/adding methods – object properties are per-object values. It should only be used with constructors (e.g. see slide) – not with instances.

The MAIN thing about a prototype is that it is the only way to define behaviour for a whole class (meh) of objects. We need to use the prototype of a constructor to give objects compatible behaviour. It has the added side-benefit of ensuring that one function per *\*type of\** object is the norm (not one per object).



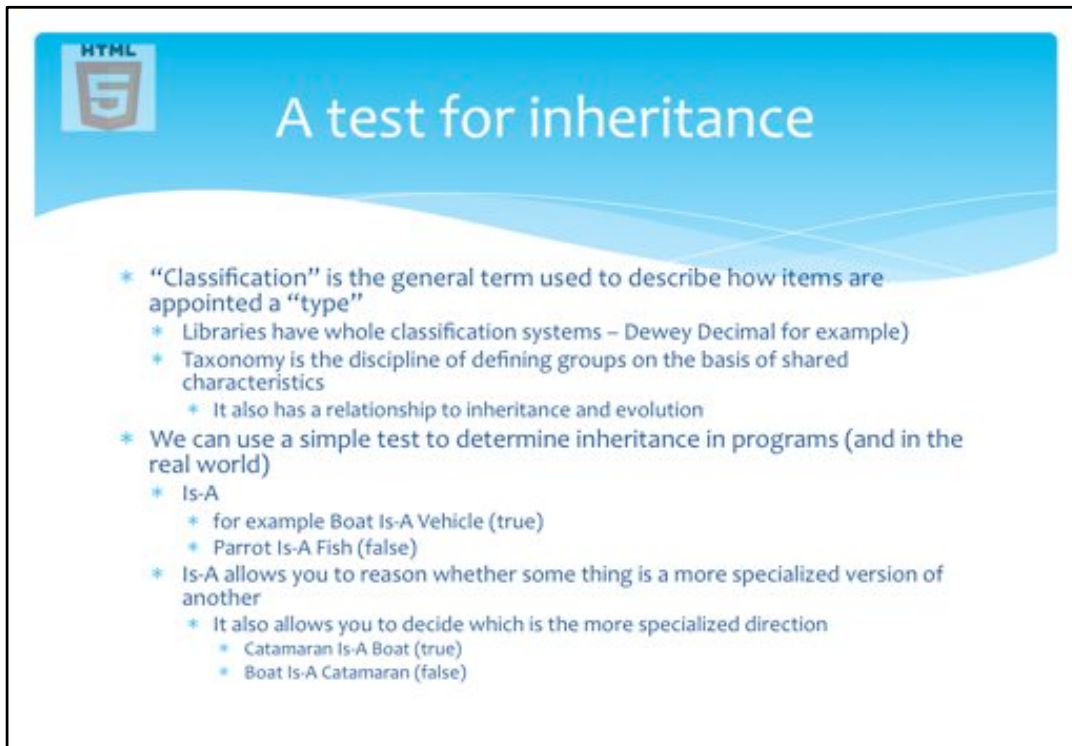
## Inheritance

- \* In JS, we need to use two mechanisms to make one constructor inherit from another:
  - \* Step 1: copy the properties of an existing type using `apply()`
  - \* Step 2: copy the prototype (to get access to the methods)

```
var Line = function(x, y, width, height){  
  this.x = x;  this.y = y;  
  this.width = width;  
  this.height = height;  
  this.colour = 0;  // Black  
};  
  
Line.prototype.setColour = function(col){  
  this.colour = col;  
};  
  
var Rectangle(x, y, width, height){  
  Line.apply(this, arguments);  
}; // copies data members  
  
Rectangle.prototype = new Line();  
// copies setColour() method (+ any others)
```

It is worth pointing out that Inheritance was an afterthought in JS. V1 of LiveScript did not have the prototype object, so composition would have been the only obvious method for basing one class on another. It is also worth pointing out that Inheritance in JS is mainly a re-use mechanism. In classical languages, Inheritance serves two purposes – re-use and the specification of a system of type. In Java and C++, inheritance frees a programmer from having to cast an object to make it compatible with some code, because the rigid type structures in this language are there to stop programmers calling functions with inappropriate parameters. In Javascript this is irrelevant because the only thing that matters is what the object can do (duck typing). However, this does not help the designer of an application – and IMHO is a crappy underpinning for big polymorphic systems. Interfaces are important in describing objects.

What this slide shows is Javascript mimicing a classical language – the fact that it takes two steps to inherit from an existing constructor is a strong clue, as is the need to call an obscure function (`apply`) to do it. However, it fits with the mindset of most programmers and is a fairly easy pattern to remember.



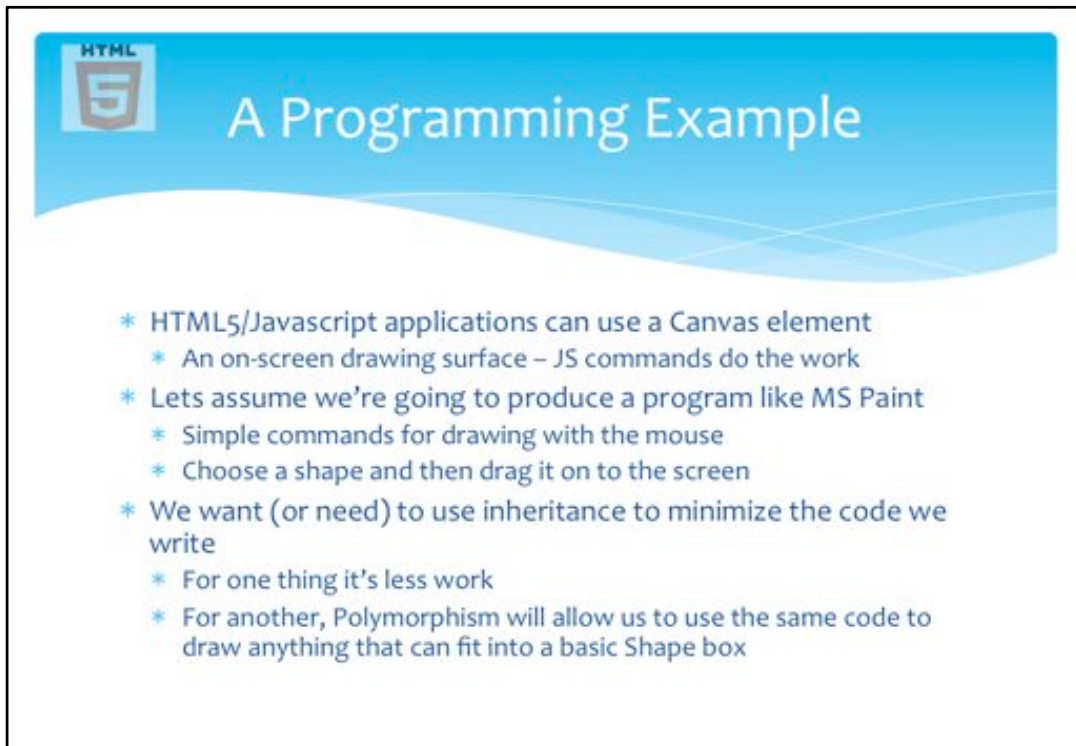
## A test for inheritance

- \* "Classification" is the general term used to describe how items are appointed a "type"
  - \* Libraries have whole classification systems – Dewey Decimal for example)
  - \* Taxonomy is the discipline of defining groups on the basis of shared characteristics
    - \* It also has a relationship to inheritance and evolution
- \* We can use a simple test to determine inheritance in programs (and in the real world)
  - \* Is-A
    - \* for example Boat Is-A Vehicle (true)
    - \* Parrot Is-A Fish (false)
  - \* Is-A allows you to reason whether some thing is a more specialized version of another
    - \* It also allows you to decide which is the more specialized direction
      - \* Catamaran Is-A Boat (true)
      - \* Boat Is-A Catamaran (false)

This all seems very abstract, but since the topic of the slide is about how to correctly apply abstractions, I don't see a problem with it.

The notion of this slide is that you should be able to apply similar tests to concepts in the arena of the program/system you are developing. In the lab, the students will be working on a graphical program that uses OOP, Inheritance and Polymorphism to minimize the amount of code (and simplify the overall abstraction) for a simple shape-drawing program (it's no accident that almost the first program written for new OOP languages is this type of thing – the abstraction works perfectly and everyone gets it immediately).


So – Shape is a general (generalized) type that has all of the behaviours of any type of shape but none of the specialisms that would make it possible to implement – position, size, colour, drawing style etc. Rectangle Is-A Shape (but certainly NOT the other way round): Circle Is-A Shape etc.



## A Programming Example

- \* HTML5/Javascript applications can use a Canvas element
  - \* An on-screen drawing surface – JS commands do the work
- \* Lets assume we're going to produce a program like MS Paint
  - \* Simple commands for drawing with the mouse
  - \* Choose a shape and then drag it on to the screen
- \* We want (or need) to use inheritance to minimize the code we write
  - \* For one thing it's less work
  - \* For another, Polymorphism will allow us to use the same code to draw anything that can fit into a basic Shape box

This turns out to be a good example. Because as well as the code, students can actually see the inheritance and polymorphism mechanisms in action is a very small number of lines. I've had lots of students mention that this example make the whole thing clear to them.

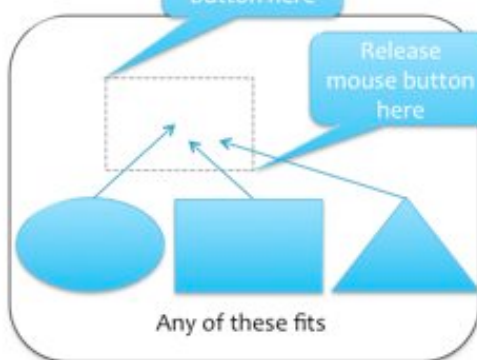


## Drawing Shapes

- \* Begin drawing something by selecting from a Shape menu
- \* Then click at the top-left corner of the shape we want to draw (X & Y)
- \* Then drag to the desired width & height and release the mouse (Width & Height)
- \* Implementation-wise, the only thing that *\*needs\** to be different for different shapes is the draw() method
  - \* square.draw(); draws a square
  - \* circle.draw() draws a circle etc.

Hold mouse button here

Release mouse button here

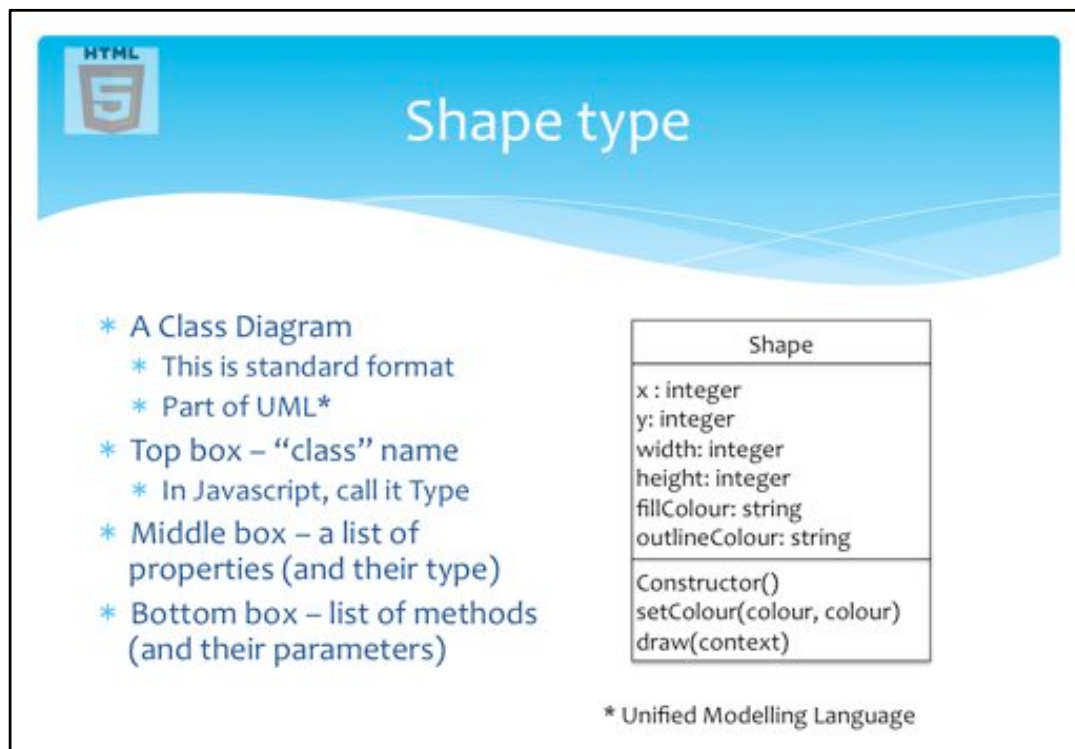


Any of these fits

This turns out to be a really good example for explaining inheritance (all shapes inherit position, size and colour properties plus associated methods (setColour(), move() etc.), but all need to specialize their drawing code. For explaining polymorphism, the example is even better since you can have code like:

```
for(int i=0; i<shapeList.length; i+=1){  
    shapeList[i].draw();  
}
```

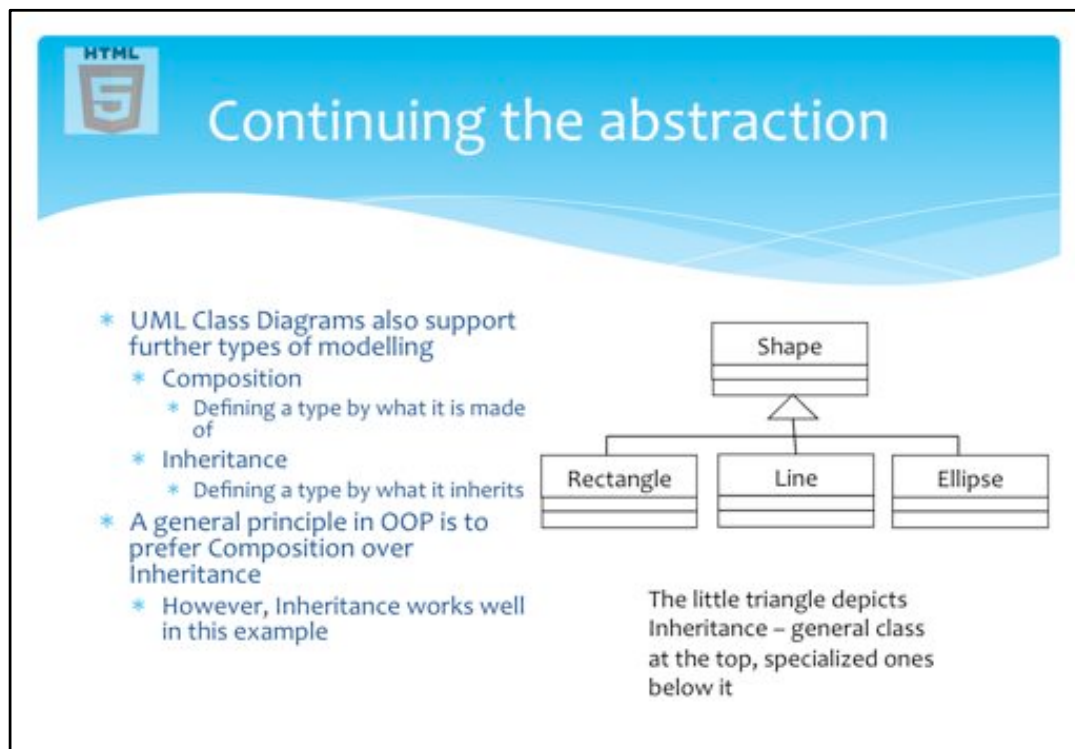
and each shape instance will “draw itself” according to its own methods. The visible effect of this on the display (and while using a debugger) is immediate and can often cause the “penny to drop”.



This should be an easy explanation – you might want to cover the origins of UML (from Gang of Three – Jacobson, Yourdon, Rumbagh) and explain that this is simply a way of depicting types (classes) diagrammatically – usually easier to get across that text.

The use of a “Class” diagram can’t go un-remarked. Obviously there are not classes in JS, but since most programmers come to OOP via classical languages like C++, Visual Basic, C#, Java etc., then the abstractions fit better into most system descriptions. (I’m quite blinkered in this area – I can’t quite see how the benefits of scale a classical paradigm can provide would also work with a prototypical language. Sure, it is more flexible, but much of OOP is about creating types of things that you can then work with in the knowledge that they behave as you expect. Duck typing is OK provided you actually check the types as you go (this is why Test-Driven-Development was developed in the first place), but the general chaos of objects with no set definition scares the crap out of me.)

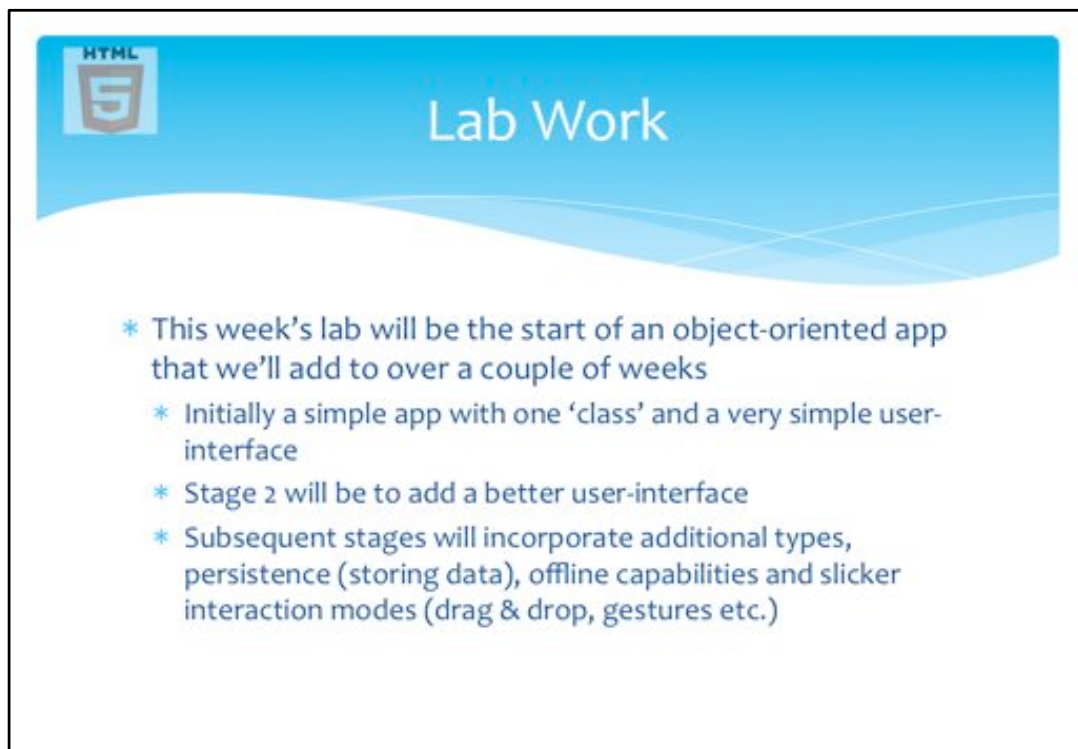




This diagram only shows Inheritance, and most people agree that Composition is a much more important principle. It might be worthwhile doing a whiteboard diagram of a composition relationship to make that clear too. I don't want to take this module too far down the classical OOP path – the students are not likely to be aspiring software designers.

For example, if you take the colour and style settings of a Shape (line style, line colour, fill style, fill colour), that can be devised as a separate type, then every shape has one of them as a member. I've not implemented the simple example like this, but it would not take much time to change it (a refactoring exercise?).

See the demo HTML-drawing.zip in Moodle.



The slide is titled "Lab Work" and features an HTML5 logo in the top left corner. The background is a light blue gradient with a wavy line separating the header from the content area. The content area contains a bulleted list of four items, each preceded by an asterisk.

- \* This week's lab will be the start of an object-oriented app that we'll add to over a couple of weeks
  - \* Initially a simple app with one 'class' and a very simple user-interface
  - \* Stage 2 will be to add a better user-interface
  - \* Subsequent stages will incorporate additional types, persistence (storing data), offline capabilities and slicker interaction modes (drag & drop, gestures etc.)

The lab target is a small Appointments app – I decided on this because it uses a simple type but one that has expansion potential. Over this and next week, it will become an Appointments book with a half-decent UI. Eventually, we ought to be able to bring in PIM type behaviour, local persistence, integration with the cloud (I already have a Google App Engine framework that would suit well) etc.

I'll be putting a fully working version of the graphics app on Dropbox soon – it will still be possible to use it as a lab exercise by adding features (no prizes for adding cool ones but I'd like to see them). Added shapes should be easy – semicircles/ellipses, diamonds, bow-tie shapes etc. Added features, such as text elements (for which a new property and supporting methods will be required, U-I elements (drag & drop, colour palette etc.) would be very nice.

I usually offer a mars-bar type prize for the best solution to an exercise or problem. Nobody's ever interested in the mars bar, but competitive types often get into it anyway.