PROGRAMMING FOR MOBILE DEVICES

Programming in Javascript

Yet more revision (this is getting too easy)

Javascript in some detail

- In Javascript, everything is an object:
  - It is a 'Dynamic Programming Language', in which variable types (and therefore how they behave) are decided during program execution
    - `var v;` does not say anything about what the object will be.
    - `var s = "Hello world";` makes it a String type.
    - `var n = 12.6;` makes it a number
  - Since everything is an ***object***, we can expect the string and number values to have other information and functions associated with them
    - `s.length = 11;`
    - `s.toUpperCase() = "HELLO WORLD";`
    - `n.toPrecision(8) = 12.600000;` (8 digit precision)
- There are standard functions to convert between types:
  - `var ns = n.toString();`
  - `var ss = "121";` `var sn = ss.valueOf();`
- Note that object ***properties*** are values associated with an object
  - Do not use brackets to access a property: `s.length`
  - Must use brackets to access a method (function): `s.toUpperCase()`

String Objects

- These have a special place in Javascript because they are used a lot in HTML

```
<html>
<body>

<script type="text/javascript">

var txt = "Hello World!";

document.write("<p>Big: " + txt.big() + "</p>");
document.write("<p>Small: " + txt.small() + "</p>");

document.write("<p>Bold: " + txt.bold() + "</p>");
document.write("<p>Italic: " + txt.italics() + "</p>");

document.write("<p>Fixed: " + txt.fixed() + "</p>");
document.write("<p>Strike: " + txt.strike() + "</p>");

document.write("<p>Fontcolor: " + txt.fontcolor("green") +
"</p>");
document.write("<p>Fontsize: " + txt.fontsize(6) + "</p>");

document.write("<p>Subscript: " + txt.sub() + "</p>");
document.write("<p>Superscript: " + txt.sup() + "</p>");

document.write("<p>Link: " +
txt.link("http://www.w3schools.com") + "</p>");
```

Big: Hello World!

Small: Hello World!

Bold: **Hello World!**

Italic: *Hello World!*

Fixed: Hello World!

Strike: ~~Hello World!~~

Fontcolor: Hello World!

Fontsize: Hello World!

Subscript: Hello World!

Superscript: Hello World!

- See http://www.w3schools.com/jsref/jsref_obj_string.asp for a complete reference of Javascript string methods & properties

Javascript Arrays

- Like everything else, an Array in JS is an object
  - Array constructor
    - **var arr = new Array();** - alternatively, **var arr = [];**
  - One useful property
    - **Array.length**
  - Several useful methods:
    - **Array.join()** makes an array into a string
    - **Array.reverse()** reverses the order of the members
    - See http://www.w3schools.com/jsref/jsref_obj_array.asp for all of them
- Similar way of manipulating JS arrays to that used in C++ & Java

This one is preferred by jsLint – an open source Javascript standards checker.

```
var arr = [];
arr[0] = "First";
arr[1] = "Second";
arr[2] = "Third";
alert(arr.join());      // "First, Second, Third"
```

Object Structure

- An object in Javascript is a map of names (of properties and methods) and values (the values of properties, the code of methods)
  - In essence, a Javascript object is an *associative array* of elements (a.k.a. a map)
  - Objects can be created as literals

```
person = {
        name: "Joe Bloggs",
        email: "joe@bloggo.com",
        telephone: "555 1234",
        dob: new Date(1975, 1, 25);  // year, month, day
        show: function() { alert(this.name+'\n'+this.email); }
}
```

```
person = {};   // (or new Object();)
person.name = "Joe Bloggs";
person.email = joe@bloggo.com;
person.telephone = "555 1234";
person.dob = new Date(1975, 1, 25);
person.show = function() {
        alert(this.name+'\n'+this.email);
}
```

  - Note – this is very different from Java & C++, where you ***must*** create a class
  - More normally, use the ***new Object()*** notation
- Note that in both of these cases, the object definition does not need to match any specific template (not very good programming practice)

Constructor functions

- Remember, in Javascript, everything is executable (even a declaration)
- A constructor in Javascript is a function that creates an object:

```
function Person(name, email, dob) {
        this.name = name;
        this.email = email;
        this.dob = dob;
}
Person.prototype.show = function () {
        alert(this.name+'\n'+this.email);
};
```

Note the convention – constructors are Capitalized

- Using a constructor function lets us create a ***type*** of object
  - Every object created with this function will have name, email, telephone, dob members and a show() method

Class Prototype

- Every class (object type created by a function) has a ***prototype*** associated with it
  - Using this, we can modify the class by adding (or changing) members or methods
  - By using the prototype instead of a specific object, we make this function available to ALL objects of the type

```
Person.prototype.getAge = function() {
        today = new Date();    // Current date.
        return today.getFullYear() – this.dob.getFullyear();
};

p = new Person("Fred Bloggs", "fred@bloggo.com" , new Date(1975, 1, 25));
alert(p.getAge());
```

- Note the big consequence of this
  - You can extend classes even when you don't have the original code
  - Some programmers hate Javascript – this is one of the reasons why

Javascript classes and Inheritance

- Javascript does not have classes, so there is no 'standard' inheritance method
  - The best method (according to Crockford) is to inherit in two steps
    1. Invoke parent constructor on the new object
    2. Attach parent prototype to the new type

```
function Employee(name, email, dob, jobtitle, salary) {
        Person.apply(this, arguments);   // Assign common parameters
        this.jobtitle = jobtitle;        // and the new ones
        this.salary = salary;
}
Employee.prototype = new Person();  // Attach the prototype functions
```

- Note that our new type of object has access to all of the parent class's members (except the function that was overridden)
- Of course, the point of inheritance is to ***extend*** and/or ***specialize*** the existing code

```
Employee.prototype.show = function () {
        alert(this.name + ", " + this.jobtitle);
};
Employee.prototype.monthsPay = function () {
        return this.salary / 12;
}
```

Polymorphism

- A class that ***extends*** an existing class inherits its fields and methods
  - Because of this, any field or method of the original class can also be accessed in a member of the new class (unless it has been replaced)
- We describe this as an ***Is-A*** relationship
  - From the previous examples an ***Employee*** is-a ***person***
  - That means that any code that we use a ***Person*** in, we can always use an ***Employee*** in without any changes
- Polymorphism – literally ***many forms*** describes this capability
- Essentially it means that when we call a method of an object, it could be one of several versions, depending on the class the object belongs to
  - Because of this, we talk of "sending a message" to an object instead of "calling a method"

```
var people = [];
people[0] = new Person(....);
people[1] = new Employee(...);
people[2] = ...
...
...
...
for(var i=0; i<people.length; i++){
        people[i].show();
        // This will call different versions
        // of show() appropriate to the
        // individual objects
}
```

OOP in Web-Apps

- ***Object-Oriented Programming*** is used to break up a complex program into simpler, individual objects
  - Each object should have a clearly defined responsibility
  - Objects communicate with other objects to accomplish an overall task
- In a web app, we typically use objects to represent well defined 'things'
  - e.g. a customer, a document, a sprite in a game etc.
  - Using objects makes keeping track of each thing easier
  - Using inheritance and polymorphism makes keeping track of *collections of things* easier

Inheritance Example – web-app lists

- The goal (apart from saving time writing code) is to create classes that have *similar* behaviour
- Easier to use, easier to update
- The code defines two new classes
  - ToDoItem – an entry in a to-do list
  - Appointment – a ToDoItem with an added location
- By defining this way, every Appointment gets all of the ToDoItem properties and methods
- An Appointment object can be treated in code as if it was a ToDoItem
- A method inherited from ToDoItem by Appointment can be ***overridden*** to provide type-specific behaviour

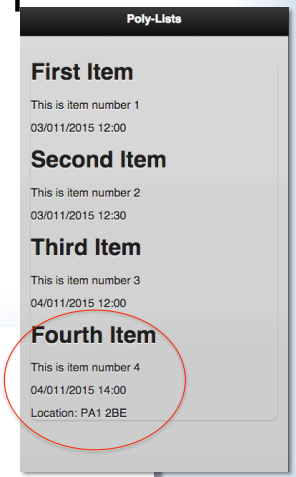
```
function ToDoItem(name, description, date, time){
   this.name = name;
   this.description = description;
   this.datetime = new Date(date + " " + time);
   this.completed = false;
}

ToDoItem.prototype.toListItem = function(){
   var li = "<li>";
   li += "<h1>" + this.name + "</h1>";
   li += "<p>" + this.description + "</p>";
   li += "<p>" + toDateTimeString(this.datetime) + "</p>";
   return li + "</li>";
}

function Appointment(name, description, date, time, location) {
   ToDoItem.apply(this, arguments);
   this.location = location;
}
Appointment.prototype = new ToDoItem();

Appointment.prototype.toListItem = function() {
   var s = ToDoItem.prototype.toListItem.call(this);
   s = s.substring(0, s.length - 5);
   s += "<p>Location: " + this.location + "</p></li>";
   return s;
}
```

Inherit ToDoItem properties and methods

Override ToDoItem method

Using Polymorphism

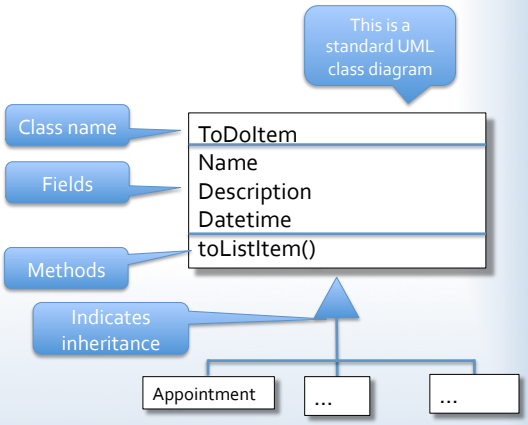
- Appointment is now compatible with code that applied to ToDoItem
  - E.g. see updateListView(), below
- The benefit of this is that the code that uses both types doesn't need to know which type it is dealing with
  - i.e. it leaves it up to the object to respond appropriately




```
function updateListView(list) {
   var lv = $("#list"), index;
   for(index = 0; index < list.length; index += 1){
      lv.append(list[index].toListItem());
   }
}
items.push(new ToDoItem("First Item", "This is item number 1", "2015-02-03", "12:00"));
items.push(new ToDoItem("Second Item", "This is item number 2", "2015-02-03", "12:30"));
items.push(new ToDoItem("Third Item", "This is item number 3", "2015-02-04", "12:00"));
items.push(new Appointment("Fourth Item", "This is item number 4", "2015-02-04", "14:00", "PA1 2BE"));
updateListView(items);
```

'Business' classes

- A "Business Class" is a type definition that performs some core feature of an application
  - e.g. A Diary app might need definitions for To-Do items and Appointments
- This distinguishes these types from "Utility" classes, which perform more generic functions
  - Handling user-interfaces, manipulating data, accessing the Internet
  - It would be unusual to take a Business type definition from one application and drop it into another
  - Utility types can be used in lots of different applications



Additional Lab Exercise

- Do this if you have time in the lab (or at home)
  - An app needs to handle two types of URL – web addresses and mail addresses
    - The distinction between these is that a mail URL takes the form
      - mailto:somebody@somedomain.com
    - While a web address takes the form
      - www.somedomain.com
  - With inheritance and polymorphism in mind, create a URL class definition using a Javascript Constructor function and prototype method (.address())
  - Create a web-page that shows a list of URLs, some of which are web addresses and some of which are mail addresses

References

- HTML5 Up and Running
  - Mark Pilgrim, O'Reilly press
  - Also the Dive into HTML5 website http://diveintohtml5.org/
- Javascript Object-Oriented Programming
  - Ryan Frishberg website
  - http://www.sitepoint.com/oriented-programming-1/
- Javascript: The Good Parts
  - Douglas Crockford, O'Reilly press
  - Also his website http://javascript.crockford.com/
  - and blog http://googlecode.blogspot.com/2009/03/doug-crockford-javascript-good-parts.html , which contains a good presentation from the author
- Javascript in 10 minutes
  - http://javascript.infogami.com/Javascript_in_Ten_Minutes