JavaScript on the Client

A primer on the JavaScript programming language

Overview

- History
- JavaScript the language
- DOM programming
- OO programming in JavaScript

A little history…

- **JavaScript was created by Netscape circa 1995**
  - JavaScript has nothing to do with Java (was going to be LiveScript!)
  - “A-list” browsers support version 1.6 == ECMA-262 edition 3 (1999)

- **JavaScript is:**
  - **dynamic**: *eval("1 + 2");*
  - **weakly type-checked**: *i = "23" * 2 + false;*
  - **functional**: *map(f, [1, 2, 3, 4, 5]);*
  - **object-oriented**: *s = obj.toString( );*
  - **case-sensitive**: *convention.useCamelCasing( );*
  - **C-like syntax**:

```
if (condition) {
  statement;
  statement;
}
```

Types

- **JavaScript offers 4 data types:**
  - *boolean*: true or false
  - *number*: 64-bit real number, including *NaN* and *Infinity*
  - *string*: 0 or more Unicode chars, literals "…" or '…'
  - *object*

```
x = 1.0;
document.write(typeof(x)); // prints 'number'
```

- **Special values:**
  - *null*: null reference (no object)
  - *undefined*: variable has no value
  - *false*, *0*, *""*, *null*, *undefined* are all treated as *false*

Declaring variables

- **In JavaScript, variable declarations are *optional***
  - this means spelling mistakes yield a *new* variable (ouch)
  - best practice is to declare before you use, so intent is clear

- **Declaring variables:**
  - use the *var* statement
  - variables are untyped, initial value is *undefined*
  - only two scopes: local or global (undeclared vars are global)
  - avoid reserved words (such as *char*, *final*, *for*)

```
var someVar = 1.0;

someVar = someVar + 1;
someVar = "And the answer is " + someVar;
alert(someVar);
```

Statements

- **JavaScript supports the usual culprits…**
  - if, switch, while, for, break, return, throw, try-catch-finally

```
var sum = 0.0;
var someArray = [1, 2, 3, 4, 5];

for (var i=0; i < someArray.length; i++)
  sum += someArray[i];

alert(sum);

// we also have a foreach-like statement:
var sum2 = 0.0;

for (var i in someArray)  // for each index, not element:
  sum2 += someArray[i];

alert(sum2);
```

Operators

- **JavaScript supports the typical operators**
  - +, -, *, /, etc.
  - beware of type coercion, especially with + operator:

```
alert(1 + "23" + 4);  // displays "1234"
```

  - && and || are short-circuited, handy for *guards* and *defaults*:

```
var name  = obj && obj.name;  // guard against null ref
var value = i || 1;           // default to 1 if i undefined
```

  - prefer === and !==, which check equality without type coercion:

```
var s = "";
alert( s == 0  );  // true!
alert( s === 0 );  // false, as you would expect
```

Arrays are not arrays!

- **Arrays are also dictionaries, but with a *length* property…**
  - you are indexing by string-based numeric keys, not integer indices
  - arrays can be sparse & full of holes…

```
var someArray = [10, 20, 30, 40, 50];

alert( someArray["3"] );  // 40

for (var i in someArray)  // indices are strings!
  alert( typeof i );
```

```
var A1 = new Array(), A2 = [];  // create an empty array:

for (var i = -100; i <= 100; i++)  // add 201 elements at -100…100:
  A1[i] = i;

alert(A1.length + ", " + A1[-100] + "..." + A1[100]);
```

101, -100…100

```
var A2 = []; // create another empty array:

A2[123] = "Pooja";  // sparse array with 3 elements, length of 790:
A2[456] = "Kim";
A2[789] = "Drago";
```

Declaring functions

- **Functions in JavaScript are untyped**
  - functions can return any value, or return nothing (*undefined* is returned)

```
//
// returns sum of all elements in the given array:
//
function sumAll(someArray)
{
  var sum = 0.0;

  for (var i in someArray)
    sum += someArray[i];

  return sum;
}
```

Functions are first-class objects

- **Functions are objects, e.g. that can be passed as parameters:**

```
// apply the function F across all the elements of collection C:
function applyToAll(F, C)
{
  for (var i in C)
    F( C[i] );
}
```

```
// alert each element of the given array:
function alertAll(someArray)
{ applyToAll(alert, someArray); }

// another way to write the above, demonstrating JavaScript's flexibility:
function alertAll_v2(someArray)
{ applyToAll(new Function("e", "alert(e);"), someArray); }

// returns sum of all elements in the given array:
function sumAll(someArray)
{
  var temp = 0.0;
  applyToAll(function addToTemp(e) { temp += e; }, someArray);
  return temp;
}
```

Closures

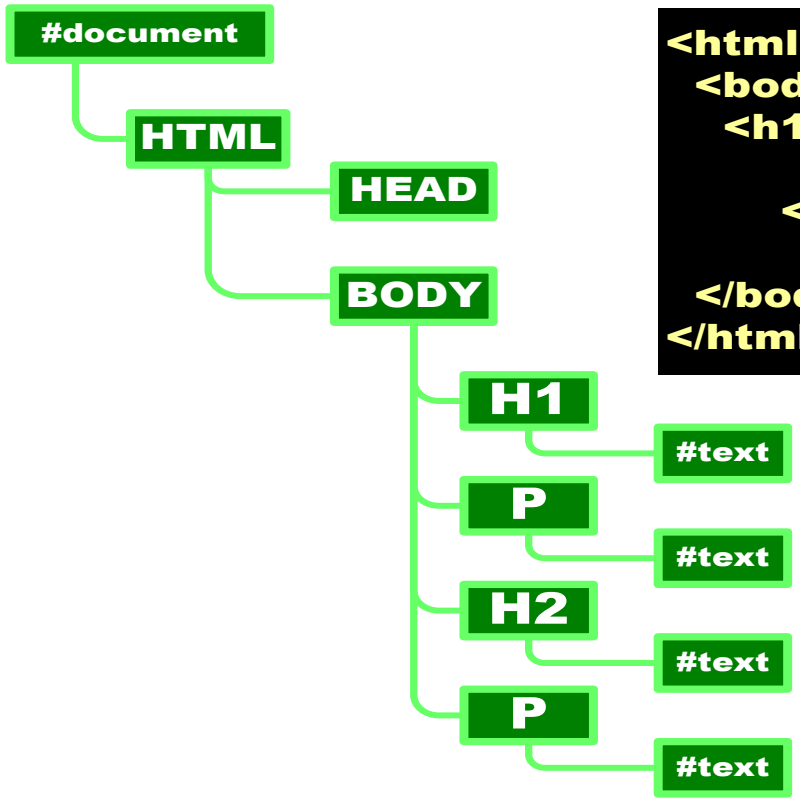
- **JavaScript supports the notion of** *closures*
  - idea that functions can be closed over the environment — i.e. their outside arguments and variables — that they need to execute…

```
// returns sum of the given array:
function sumAll(someArray)
{
  var temp = 0.0;
  applyToAll(function addToTemp(e) { temp += e; }, someArray);
  return temp;
}
```

addToTemp needs a non-local variable in order to accumulate the sum — so we take advantage of JavaScript's closure mechanism to use the local variable "temp" declared in the outer function…

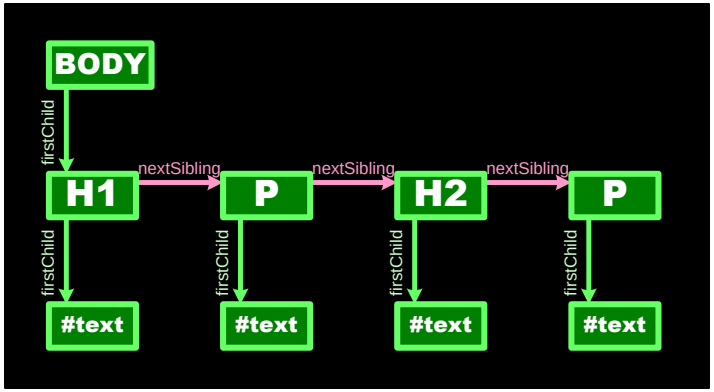
The DOM: Document Object Model

- Programming against the DOM is a common client-side task




```
<html>
  <body>
    <h1>Heading 1</h1>
          <p>Paragraph.</p>
      <h2>Heading 2</h2>
          <p>Paragraph.</p>
  </body>
</html>
```

Walking the DOM

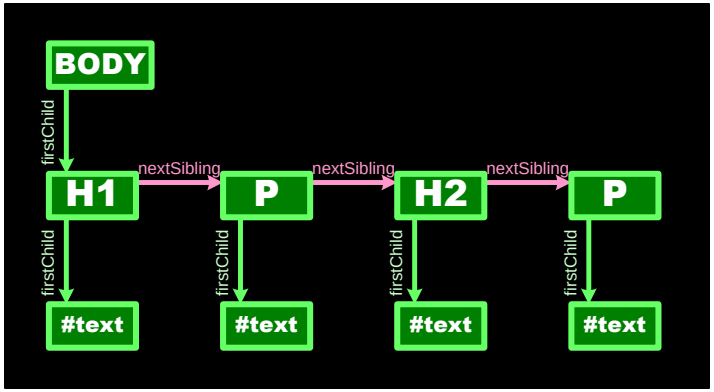


- **A function to recursively traverse the DOM, from any node:**

```
// walk the DOM starting at node, applying the function F to each node:
function walkTheDOM(node, F)
{
  F(node);
  node = node.firstChild;
  while (node)
  {
     walkTheDOM(node, F);
     node = node.nextSibling;
  }
}
```

```
walkTheDOM( document.getElementById("thebody"),
  function (node)
  { alert(node.nodeName + "," + node.nodeValue); } );
```

Manipulating the DOM



- **Example — creating and linking new elements:**

```
function onClick()
{
  var h3  = document.createElement("h3");
  var txt = document.createTextNode("Heading 3");

  h3.appendChild(txt);

  var body = document.getElementById("thebody");
  body.appendChild(h3);

  walkTheDOM( document.getElementById("thebody"), ...);
}
```

So far, so good…

- **Okay, at this point, JavaScript isn't all that different**
  - except for closures, and maybe ===
- **So what's the big deal?**
  - objects…

Objects are not what you think

- **JavaScript does not contain the notion of a** *class*
- **So how do we define objects?**

- **Solution? In short:**
  - *objects are collections of key/value pairs, where the keys represent the field & method names, and the values represent field values & method bodies…*
  - `this` *refers to current owner of executing method (**not like** C# - be careful!)*

```
var employee = new Object();

employee["name"]   = "Pooja";    // define Name key/value pair
employee["salary"] = 72001.44;   // define Salary key/value pair

employee["issuePaycheck"] = function() {  // define issuePaycheck pair
  var monthly = this["salary"] / 12;
  alert(this["name"] + ": " + monthly);
};
```

```
employee["issuePaycheck"]();
```

Object syntax

- **Observation #1:** *objects are dictionaries (key/value pairs)*
- **Observation #2:** *the object notations . and [ ] are interchangeable*
- **Observation #3:** *different ways to create an object, with same result*

```
var employee = new Object();

employee["name"] = "Pooja";
employee["salary"] = 123.00;
:
:
employee["issuePaycheck"]();
```

```
var employee = {};  // empty obj

employee.name = "Pooja";
employee.salary = 123.00;
:
:
employee.issuePaycheck();
```

```
var employee = {
  "name"   : "Pooja",
  "salary" : 123.00,
  :
  :
};

employee.issuePaycheck();
```

Issue #1: class definition

- **Issue #1:** *how to write a class definition in JavaScript?*
- **Solution?** *define function to act as both class def & constructor*

```
function Employee(name, salary)
{
  this.name = name;         // add Name key/value pair, with value name
  this.salary = salary;     // add Salary key/value pair, with value salary

  this.issuePaycheck = function()   // add issuePaycheck key/value pair…
  {
    var monthly = this.salary / 12;
    alert(this.name + ": " + monthly);
  };
}
```

  - *we'll call this function a constructor function*

Issue #2: creating object instances

- **Issue #2:** *how to create individual instances based on class definition?*
- **Solution?** *create empty object & invoke constructor function on this object*

```
function Employee(name, salary)
{
   .
   .
   .
}
```

```
var employee;
employee = {};
Employee.call(employee, "Pooja", 72001.44);
```

which is equivalent to this shorter, much more familiar version…

```
var employee;
employee = new Employee("Pooja", 72001.44);
```

Issue #3: sharing method definitions

- **Issue #3:** *every object instance ends up with its own set of methods (i.e. function objects), which is a waste of memory*
- **Solution?** *every object in JavaScript has a prototype object that acts as an extension of the object itself — by storing method references in the prototype object associated with the constructor function, these methods will be inherited by object instances…*

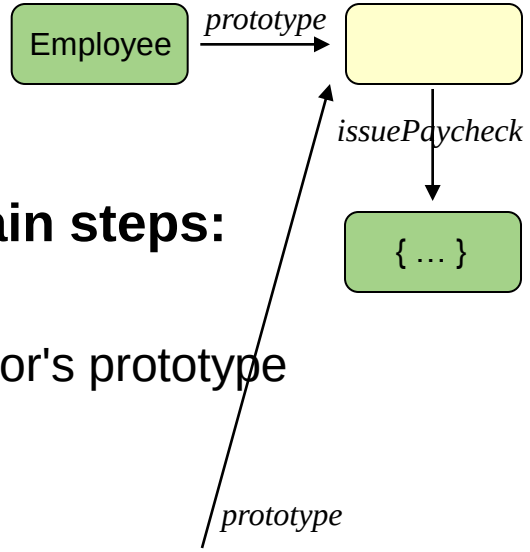
```
function Employee(name, salary)
{
  this.name = name;
  this.salary = salary;
}

Employee.prototype.issuePaycheck = function()
{
  var monthly = this.salary / 12;
  alert(this.name + ": " + monthly);
};
```

```
function Employee(name, salary)
{
  .
  .
  .
  this.issuePaycheck = function()
  {
    var monthly = this.salary / 12;
    alert(this.name + ": " + monthly);
  };
}
```

X

What "new" really does…



- **In JavaScript, the *new* operator performs 3 main steps:**
  - creates new, empty object
  - defines new object's prototype based on constructor's prototype
  - invokes constructor with new object as "this"

```
var e1;
e1 = new Employee("Pooja", 72001.44);
```

which is roughly equivalent to…

```
var e1;

e1 = {};  // (1) create new object
e1.prototype = Employee.prototype;  // (2) copy object ref
Employee.call(/*this*/ e1, "Pooja", 72001.44);  // (3) init
```

Issue #4: static members

- **Issue #4:** *how to implement static fields & methods?*
- **Solution?** *associate fields & methods with constructor function*

```
function Employee(name, salary)
{
  this.name = name;
  this.salary = salary;
}

Employee.prototype.issuePaycheck = function()
{
  var monthly = this.salary / 12;
  alert(this.name + ": " + monthly);
};

Employee.companyName = "Global Widgets, Inc.";
Employee.lookup = function(name)
{
  // async call to lookup employee on server...
};
```

```
alert( Employee.companyName );
alert( Employee.lookup("Pooja").salary );
```

Issue #5: private members

- **Issue #5:** *how about private fields & methods?*
- **Solution?** *use closures…*

```
function Employee(name, salary)
{
  this.name = name;  // public name field:

  // public property methods to access hidden salary field:
  this.get_salary = function()          { return salary; };
  this.set_salary = function(newSalary) { salary = newSalary; };
}

Employee.prototype.issuePaycheck = function()
{
  var monthly = this.get_salary() / 12;
  alert(this.name + ": " + monthly);
};
```

- *Note that methods *must* be rewritten to call property methods…*

Private methods…

- Here's an example of using closures to define a private method…

```
function Employee(name, salary)
{
  // private field and method to keep track of trouble-makers:
  var probations = 0;
  var doubleSecretProbation = function()
  {
    alert(probations);
  };

  // public method to put an employee on probation:
  this.putOnProbation = function()
  {
    probations++;
    if (probations > 2)  // time for double-secret probation!
      doubleSecretProbation();
  };
  .
  .
  .
```

Issue #6: private members revisited

- **Issue #6:** *closures are confusing, and expensive (private copies of methods, etc.). Is there a better way?*
- **Solution?** *No. So Microsoft recommends convention of using _ prefix to denote private members, even though members are really public…*

```
function Employee(name, salary)
{
  this._name = name;       // private fields by convention
  this._salary = salary;
  this._probations = 0;
}

Employee.prototype._doubleSecretProbation = function() { … };

Employee.prototype.get_name = function()        { return this._name };
Employee.prototype.set_name = function(newName) { this._name = newName };
Employee.prototype.get_salary = function()          { … };
Employee.prototype.set_salary = function(newSalary) { … };

Employee.prototype.issuePaycheck = function() { … };
Employee.prototype.putOnProbation = function() { … };
```

OOP in JavaScript

- **There you go, objects and classes in JavaScript!**

```
var e1, e2, e3;
e1 = new Employee("Pooja", 72001.44);
e2 = new Employee("Kim", 16000.00);
e3 = new Employee("Drago", 60000.48);
.
.
.
e3.issuePaycheck();
```

  - *we'll tackle inheritance and interfaces in the next module…*

Some implications of the JS approach…

- **Since objects are dictionaries, this has some implications**
  - you can add new members anytime you want:

```
var e1 = new Employee("Pooja", 72001.44);
 :
 :
e1.myOwnFavoriteField = "Some String Value!";
```

  - you can foreach across an object's members:

```
for(var key in e1)
  alert( "key:" + key + ", value:" + e1[key] );
```

  - to foreach across *true* members (ignoring prototype members):

```
for(var key in e1)
  if (e1.hasOwnProperty(key))
    alert( "key:" + key + ", value:" + e1[key] );
```

_name, _salary, _probations, myOwnFavoriteField

Summary

- **JavaScript is one of the most popular languages on the planet**
- **JavaScript is one of the most misunderstood languages on the planet**

- **JavaScript is:**
  - case-sensitive with a C-like syntax
  - weakly type-checked & very flexible
  - object-oriented with a functional underpinning
  - dictionary & prototype-based in its approach to OOP

References

- **Web sites:**
  - Douglas Crockford: http://javascript.crockford.com/
  - JavaScript syntax: http://en.wikipedia.org/wiki/JavaScript_syntax
  - "*What ASP.NET devs should know about JavaScript*", http://odetocode.com/Articles/473.aspx

- **Books and articles:**
  - "*JavaScript: Create Advanced Web Apps…*", MSDN Magazine, May 2007
  - "*JavaScript: The Definitive Guide*", by David Flanagan (O'Reilly)