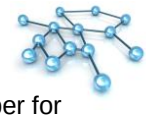
Rich Internet Applications

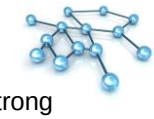
2. Core JavaScript

The importance of JavaScript



- Many choices open to the developer for server-side
  - Can choose server technology for development and deployment
  - ASP.NET, PHP, Ruby on Rails, etc
- No choice for development of native browser application
  - Must work with the user's browser
  - The only programming language which is universally supported is JavaScript

Why learn JavaScript?



- No just browser apps – there is strong interest in JavaScript for server applications (node.js) and desktop apps (Windows 8 Metro, with HTML 5)
- Libraries such as jQuery are very popular
- Can aid development by simplifying syntax and abstracting browser differences
- However, important to understand the underlying language, which has features which can seem "alien" to C# developers

Resources

- Reading
  - JavaScript: The Definitive Guide (6th Edition), David Flanagan (very thorough!)
  - JavaScript: The Good Parts, Douglas Crockford
  - W3Schools
- Useful development tools include:
  - Text editor, or Visual Studio
  - Firefox with Firebug for debugging
  - jsFiddle for quick experimenting and sharing

Similarities to C#

- Operators + expressions
- Statements;
- Code blocks {}
- if statements
- for and while loops
- switch statements
- try-catch
- …

Types and values

- Values can be of primitive and object types
- Primitive – strings, numbers, boolean, special types (null, undefined)

```
// JavaScript supports several types of values
x = 1;                  // Numbers.
x = 0.01;               // One Number type for integers and reals.
x = "hello world";      // Strings of text in quotation marks.
x = 'JavaScript';       // Single quote marks also delimit strings.
x = true;               // Boolean values.
x = false;              // The other Boolean value.
x = null;               // Null  means "no value".
x = undefined;          // Undefined is like null.
```

- Any value that is not one of these is an object

Variables

- Declare with **var** key word
  - No type specified - dynamic typing
  - **Not** the same as var (or dynamic) in C#

```
// Variables can hold values of any type
var i = 10;
i = "ten";
```

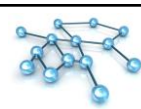
- Scope is code block in which declared
- Can have global variables – if not declared, scope is automatically global

Objects (reference types)

- Object is collection of name/value pairs
- Value can itself be an object – can have complex object structures
- Can create with object literal expressions (similar to JSON)

```
// An object is a collection of name/value pairs, or a string to value map.
var book = {              // Objects are enclosed in curly braces.
    topic: "JavaScript",  // The property "topic" has value "JavaScript".
    fat: true             // The property "fat" has value true.
};                        // The curly brace marks the end of the object.
// Access the properties of an object with . or []:
document.write(book.topic + "<br/>");              // => "JavaScript"
document.write(book["fat"] + "<br/>");             // => true: another way to access
property values.
book.author = "Flanagan";  // Create new properties by assignment.
book.contents = {};        // {} is an empty object with no properties.
```


Arrays



- Numerically indexed lists of values

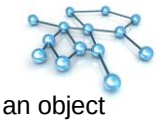
```
var primes = [2, 3, 5, 7]; // An array of 4 values, delimited with [
and ].
document.write(primes[0] + "<br/>");                  // => 2: the
first element (index 0) of the array.
document.write(primes.length + "<br/>");               // => 4: how
many elements in the array.
primes[4] = 9;             // Add a new element by assignment.
primes[4] = 11;            // Or alter an existing element by
assignment.
var empty = [];            // [] is an empty array with no elements.
```

Arrays and objects

- Arrays can contain objects
- Objects can have array properties

```
var points = [             // An array with 2 elements.
    {x: 0, y: 0 },            // Each element is an object.
    {x: 1, y: 1 }
];
var data = {               // An object with 2 properties
    trial1: [[1, 2], [3, 4]],  // The value of each property is an array.
    trial2: [[2, 3], [4, 5]]   // The elements of the arrays are arrays.
};
```

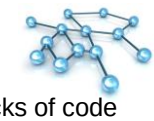

For-in loop



- Iterates through the properties of an object
- Loop variable is the name of the property
- If object is an array, the loop variable is the index
- Can access value using property name or index

```
for (var p in a) {
    document.write(a[p] + " ");
}
```


Functions



- Functions are parameterised blocks of code which can be invoked over and over again
- Functions are values which can be assigned to variables
- Can be passed as parameters to other functions
- Often used to define callback functions
  - Similar to delegates in C#
- Can be nested

Functions

```
function plus1(x) {        // Define a function named "plus1" with parameter "x"
    return x + 1;          // Return a value one larger than the value passed in
}

var y = 3;
plus1(y);                 // => 4: y is 3, so this invocation returns 3+1

var cube = function (x) { // Functions are values and can be assigned to vars
    return x * x * x;           // Compute the function's value
};                         // Semicolon marks the end of the assignment.
cube(plus1(y));            // => 64: invoke two functions in one expression

var p1 = plus1;
p1(cube(y));               // => 28: invoke two functions in one expression

function hypotenuse(a, b) {   // Nested functions
    function square(x) {
        return x*x;
    }
    return Math.sqrt(square(a) + square(b));
}
hypotenuse(5,12);            // => 13
```


Methods

- Functions can be assigned to the properties of an object
- These functions are methods of the object
- Can be added dynamically

```
var points = [             // An array with 2 elements.
    {x: 0, y: 0 },            // Each element is an object.
    {x: 1, y: 1 }
];
points.dist = function () { // Define a method to compute distance between points
    var p1 = this[0];      // First element of array we're invoked on
    var p2 = this[1];      // Second element of the "this" object
    var a = p2.x - p1.x;     // Difference in X coordinates
    var b = p2.y - p1.y;     // Difference in Y coordinates
    return Math.sqrt(a * a + // The Pythagorean theorem
                     b * b); // Math.sqrt() computes the square root
};
```


Global object

- When the JavaScript interpreter starts (or when page loads), it creates a new global object and gives it an initial set of properties :
  - global properties like undefined, Infinity, and NaN
  - global functions like isNaN(), parseInt() and eval()
  - constructor functions like Date(), RegExp(), String(), Object(), and Array()
  - global objects like Math and JSON
- In the browser there is also a Window object which defines other globals

Classes

- Classes are important when we use patterns and frameworks which provide structure in JavaScript applications, e.g. MVC
- Need classes for “model”
- JavaScript is object-oriented, but the syntax for creating “classes” is very different from languages like C# and Java
- Prototype-based programming
- Similar languages include ActionScript, Lua

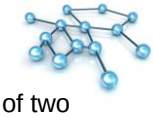
Classes

- JavaScript does not have class definitions
- Instead of class definition, we can use:
  - Constructor function, which defines properties
  - Prototype object associated with the constructor function, which defines methods
- Instances are created with the **new** keyword
- Instances inherit methods from prototype

Class example

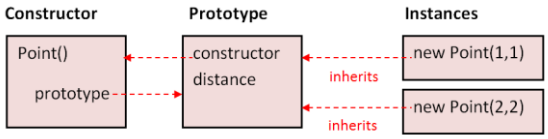
```
// Define a constructor function to initialize a new Point object
function Point(x, y) {      // By convention, constructors start with capitals
    this.x = x;             // this keyword (current context) is the new object being
                            // initialized
    this.y = y;             // Store function arguments as object properties
}                           // No return is necessary, implicitly returns instance

// Define methods for Point objects by assigning them to the prototype
// object associated with the constructor function.
Point.prototype.distance = function () {
    return Math.sqrt(       // Return the square root of x² + y²
        this.x * this.x +   // This is the Point object on which the method...
        this.y * this.y     // ...is invoked.
    );
};

// Use a constructor function with the keyword "new" to create instances
var p = new Point(1, 1);   // The geometric point (1,1)

// Now the Point object p (and all future Point objects) inherits the method
distance()
p.distance()                    // => 1.414
```

Constructors and prototypes



- This works through the existence of two properties of JavaScript objects
- **constructor**
  - a reference to the function that created the instance's prototype
- **prototype**
  - An object from which other objects can inherit properties
- Instances inherit properties from their constructor function's prototype

Constructors and prototypes



Accessing constructor and prototype

- Access constructor through instance or constructor prototype

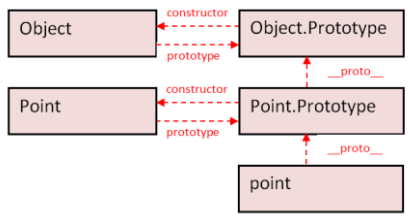
```
var point = new Point(2, 2);
document.write(point.constructor + "<br/>");
document.write(Point.prototype.constructor + "<br/>");
```

- Can't directly access prototype through instance, but can use non-standard __proto__ property in some browsers

Prototype chain

- When retrieving a property, JavaScript first looks to see if the property is defined directly in the object
- If not, it then looks at the object's prototype to see if the property is defined there
- This continues up the *prototype chain* until reaching the root prototype, Object.prototype
- Object is the prototype for Point in our example

Prototype chain



Prototype chain example

- One method defined in object, one in prototype

```
function Point(x, y) {      // Constructor function
    this.x = x;                                 // own properties
    this.y = y;
    this.angle = function() {                   // own method
        return Math.atan(this.y / this.x);
    };
}

Point.prototype.distance = function () {              // prototype method
    return Math.sqrt(
        this.x * this.x +
        this.y * this.y
    );
};

// create object
var point = new Point(2, 2);
// call own method
document.write(point.angle() + "<br/>");
// call prototype method
document.write(point.distance() + "<br/>");
// call method not defined in this object - follows prototype chain to Object
document.write(point.toString() + "<br/>");
```

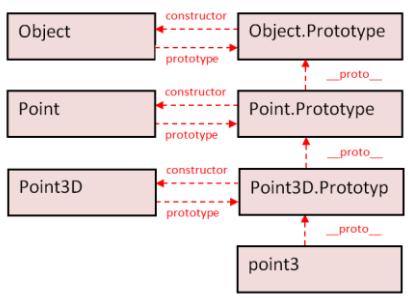
Changing the prototype

- Prototype is dynamic, like any other JavaScript property
- Can add new properties to a prototype, and existing and new instances will inherit these
- Can replace prototype with a new object – existing instances retain link to their original prototype
- Can replace instance prototype through __proto__

Subclasses

- Create a subclass by attaching a new constructor function to a prototype chain
- New class inherits prototype properties
- Can override and extend through properties in new prototype
- Can pass parameters to superclass constructor using JavaScript's **apply** function

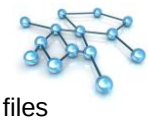
Extended prototype chain



Subclass example

```
// "subclass" constructor
function Point3D(x, y, z) {
    this.z = z;
    Point.apply(this, [x, y]);
}

// "subclass" prototype - hooks into  "superclass" prototype chain
Point3D.prototype = new Point();

// Now add properties to the prototype
Point3D.prototype.distance3D = function () {
    return Math.sqrt(// Return the square root of x² + y² + z²
        this.x * this.x +
            this.y * this.y +
                this.z * this.z);
};

var point3 = new Point3D(1, 1, 1);
document.write(point3.distance().toFixed(2));   // 1.41: inherited method
document.write("<br/>");
document.write(point3.distance3D().toFixed(2));    // 1.73: extension method
document.write("<br/>");
```


Modules and namespaces

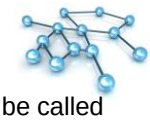


- JavaScript modules are simply .js files
- JavaScript does not define language constructs for working with modules
- Writing modular code is a case of following certain coding conventions
- Classes are global by default
- Can create new object to be the namespace for classes within a module
- Include namespace when calling constructor

Namespaces example

```
// Define a constructor function as a property of the Models namespace
Models.PointN = function(x, y) {
    this.x = x;
    this.y = y;
};

// Define method using prototype
Models.PointN.prototype.distance = function () {
    return Math.sqrt(
        this.x * this.x +
        this.y * this.y
    );
};

// Call constructor through the Models namespace
var p = new Models.PointN(1, 1);
document.write(p.distance().toFixed(2));  // => 1.41
document.write("<br/>");

// "import" namespace - would do this if namespace was in a different script file
(module)
var PointN = Models.PointN;
var p2 = new PointN(1, 1);
document.write(p2.distance().toFixed(2));  // => 1.41
```

Are constructors unsafe?



- Constructors are intended only to be called using the new key word
- Switches the context (this) to be the object which is being constructed
- However, there is nothing to stop constructor function being called like any other function, without new
- Now, context is global
- Can have unintended consequences…

Example of unsafe constructor

```
// Define constructor function
var Person = function (name, location) {
    this.name = name;
    this.location = location;
};

// Create one instance of Person
var goodguy = new Person('Alice', 'Glasgow');

// Create another instance of Person, forgetting to use new
var badguy = Person('Bob', 'Glasgow');
```

- In a browser, the global context is **window** object
- Window has location property, which we change if we forget **new**
- Browser will redirect

Constructing with a factory

- Use a method with an explicit return value
- If called in global context, will call itself in correct context using **new**, and return result

```
// Factory function
// If called in context of global object, calls itself again
// in correct context and returns instance which is created
function PointF(x,y) {
    if (this === window) {
        return new PointF(x,y);
    }

    this.x = x;
    this.y = y;
}

var p = PointF(1, 1);
```

What's next

- Client-side JavaScript - putting JavaScript to work in the web browser