

Object-Oriented JavaScript

CSE 190 M (Web Programming) Spring 2008
University of Washington

Except where otherwise noted, the contents of this presentation are © Copyright 2008 Marty Stepp, Jessica Miller, and Jim George, and are licensed under the Creative Commons Attribution 2.5 License.



Lecture outline

- background / motivation
- object-oriented JavaScript
- creating classes

Why use classes and objects?

- JavaScript treats functions as *first-class citizens*
- small programs are easily written without adding any classes or objects
- larger programs become cluttered with disorganized functions
- grouping *related data and behavior* into objects helps manage size and complexity, promotes code reuse

Interacting with objects

You have already *used* many types of JavaScript objects:

- Strings
- arrays
- HTML / XML DOM objects
- Prototype: `Ajax.Request`
- Scriptaculous: `Effect`, `Sortable`, `Draggable`

Creating a new anonymous object

```
var name = {  
  fieldName: value,  
  ...  
  fieldName: value  
};
```

JS

```
var pt = {  
  x: 4,  
  y: 3  
};  
alert(pt.x + ", " + pt.y);
```

JS

- in JavaScript, you can create a new object without creating a class
- the above is like a `Point` object; it has fields named `x` and `y`
- the object does not belong to any class; it is the only one of its kind

You've already done this...

```
new Ajax.Request(  
  "http://example.com/app.php",  
  {  
    method: "get",  
    onSuccess: ajaxSuccess  
  }  
);  
  
new Effect.Opacity("my_element",  
  {  
    duration: 2.0,  
    from: 1.0,  
    to: 0.5  
  }  
);
```

- the sets of parameters between `{ }` that you passed to Prototype and Scriptaculous were actually anonymous objects

Objects with behavior

```
var name = {  
  ...  
  methodName: function(parameters) {  
    statements;  
  },  
  ...  
};
```

JS

```
var pt = {  
  x: 4,  
  y: 3,  
  distanceFromOrigin: function() {  
    return this.x * this.x + this.y * this.y;  
  }  
};  
alert(pt.distanceFromOrigin()); // 5
```

JS

- like in Java, objects' methods run "inside" that object
 - inside an object's method, the object can refer to itself as `this`
 - unlike in Java, the `this` keyword is mandatory in JS

A paradigm shift: prototypes

*What if we want to create an entire new class, not just one new object?
(so that we could say `new Point()`)*

- JavaScript supports objects and is considered an object-oriented language
 - but, unlike Java, *JavaScript does NOT have classes*
 - JS instead supports a concept called *prototypes* (not to be confused with the Prototype library)
- **prototype**: a "super-object," an ancestor of a JavaScript object
 - like a superclass from inheritance, but on the level of individual objects
 - every object has a prototype (its "daddy") and can use the prototype's behavior

Using prototypes

- A prototype can be used to create a new type of objects, much like a class.
- Think of a prototype as a template object that we fill with all relevant behavior for each object of the "class" we're creating.
- Steps to creating a new type using prototypes:
 1. Write a constructor for the new type.
 2. Initialize any object state in the constructor.
 3. Add any desired behavior (methods) to the prototype.

Syntax for prototypes

```
// constructor
function className(parameters) {
  this.fieldName = value;
  ...
  this.fieldName = value;
}

// adding a method to the prototype
className.prototype.methodName = function(parameters) {
  statements;
}
```

JS

- inside the constructor and methods, can refer to the current object as `this`

Prototype example

```
// Constructs a new Point object at the given initial coordinates.
function Point(initialX, initialY) {
  this.x = initialX;
  this.y = initialY;
}

// Computes the distance between this Point and the given Point p.
Point.prototype.distance = function(p) {
  var dx = this.x - p.x;
  var dy = this.y - p.y;
  return Math.sqrt(dx * dx + dy * dy);
};

// Returns a text representation of this Point object.
Point.prototype.toString = function() {
  return "(" + this.x + ", " + this.y + ")";
};
```

PHP

-
- the above code could be saved into a file `Point.js`
 - the `toString` method works similarly as in Java

Creating classes

How Prototype (uppercase P) adds class semantics to JavaScript

Classes and prototypes

- limitations of prototype-based code:
 - unfamiliar / confusing to many programmers
 - somewhat unpleasant syntax
 - difficult to get inheritance-like semantics (subclassing, overriding methods)
- Prototype library's `Class.create` method makes a new class of objects
 - essentially the same as using prototypes, but uses a more familiar style and allows for richer inheritance semantics

Creating a class

```
className = Class.create({  
  // constructor  
  initialize : function(parameters) {  
    this.fieldName = value;  
    ...  
  },  
  
  functionName : function(parameters) {  
    statements;  
  },  
  ...  
});
```

JS

- constructor is written as a special `initialize` function

Class.create example

```
Point = Class.create({
  // Constructs a new Point object at the given initial coordinates.
  initialize: function(initialX, initialY) {
    this.x = initialX;
    this.y = initialY;
  },

  // Computes the distance between this Point and the given Point p.
  distance: function(p) {
    var dx = this.x - p.x;
    var dy = this.y - p.y;
    return Math.sqrt(dx * dx + dy * dy);
  },

  // Returns a text representation of this Point object.
  toString: function() {
    return "(" + this.x + ", " + this.y + ")";
  }
});
```

PHP

Inheritance

```
className = Class.create(superclass, {
  ...
});
```

JS

```
// Points that use "Manhattan" (non-diagonal) distances.
ManhattanPoint = Class.create(Point, {
  // Computes the Manhattan distance between this Point and p.
  // Overrides the distance method from Point.
  distance: function(p) {
    var dx = Math.abs(this.x - p.x);
    var dy = Math.abs(this.y - p.y);
    return dx + dy;
  },

  // Computes this point's Manhattan Distance from the origin.
  distanceFromOrigin: function() {
    return this.x + this.y;
  }
});
```

Referring to superclass: \$super

```
name: function($super, parameters) {  
  statements;  
}
```

JS

```
ManhattanPoint3D = Class.create(ManhattanPoint, {  
  initialize: function($super, initialX, initialY, initialZ) {  
    $super(initialX, initialY); // call Point constructor  
    this.z = initialZ;  
  },  
  
  // Returns 3D "Manhattan Distance" from p.  
  distance: function($super, p) {  
    var dz = Math.abs(this.z - p.z);  
    return $super(p) + dz;  
  },  
  
  // Overrides Point's toString method.  
  toString: function() {  
    return "(" + this.x + ", " + this.y + ", " + this.z + ")";  
  }  
};
```

-
- can refer to superclass as \$super in code