

11. JavaScript's Object Model

Class-Based vs. Prototype-Based Languages

Class-based object-oriented languages have two distinct kinds of entities: classes and instances.

A *class* defines all the properties characterizing a certain set of objects.

It's something abstract.

An *instance* is the instantiation of a class (one of its members).

A prototype-based language – such as JavaScript – has just objects.

A *prototypical object* is used as a template from which to get the initial properties for a new object.

Any object can specify its own properties, either when it is created or at run time.

Any object can be the prototype of another so that the second share's the first's properties.

In talking about JavaScript, we'll use the term “instance” informally to mean an object created using a particular constructor.

Comparison of Class-Based and Prototype-Based Object Systems

Class-Based	Prototype-Based
Class and instance are distinct	All objects are instances
Define a class with a class definition; instantiate a class with constructor methods.	Define and create a set of objects with constructor functions.
Create a single object with the <code>new</code> operator.	Same
Construct an object hierarchy by using class definitions to define subclasses of existing classes.	Construct an object hierarchy by assigning an object as the prototype associated with a constructor function.
Inherit properties by following this class chain.	Inherit properties by following the prototype chain.
Class definition specifies all properties of all class instances – can't add properties dynamically.	Constructor function or prototype specifies an <i>initial set</i> of properties. Can add or remove properties dynamically to individual objects or to the entire set of objects.

Creating Objects

The most generic “class” is `Object`, with constructor `Object()`.

We can also create an object with an object literal:

A comma-separated list of property specifications enclosed in curly braces.

Each property specification consists of the property name followed by a colon and the property value.

Example:

This also illustrates that an object can be the value of the property of another object.

```
<script type = "text/javascript">
  var bart = new Object(),
      homer = {
        name: "Homer Simpson",
        age: 34,
        son: bart
      };

  bart.name = "Bart Simpson";
  bart.age = 10;
  bart.father = homer;

  document.writeln( "The age of the father of Bart: "
                    + bart.father.age + "<br>");
  document.writeln( "The age of the son of Homer: "
                    + homer.son.age + "<br>");
</script>
```

```
The age of the father of Bart: 34
The age of the son of Homer: 10
```

Constructors

To define a prototype object in JavaScript, define a constructor function (a syntactically normal function) that assigns default values to the properties of `this` (a keyword).

Example

```
function Employee ( ) {  
    this.name = "";  
    this.dept = "general";  
}
```

An instance is created with the `new` operator, e.g.,

```
mark = new Employee;
```

We can change the value of a property by an assignment – e.g.,

```
mark.name = "Doe, Mark";
```

We can add new properties to any object at run time – e.g.,

```
mark.age = 25;
```

Example

(We now show .js files in our examples.)

```
function Employee() {
    this.name = "";
    this.dept = "general";
}

jim = new Employee;

jim.name = "Jones, James";
jim.age = 25;

document.writeln( jim.name );
document.writeln( "<br>" + jim.dept );
document.writeln( "<br>" + jim.age );
```

```
Jones, James
general
25
```

Constructors with Parameters

Consider

```
function Employee (name, dept) {
  this.name = name || "";
  this.dept = dept || "general";
}
```

When `new Employee (...)` is executed, if only zero or one parameter is supplied, then `dept` has value `undefined`, so

```
dept || "general"
evaluates to "general".
```

If two parameters are supplied, `dept` gets the value of the second, and

```
dept || "general"
evaluates to that value.
```

So

```
this.dept = dept || "general";
```

lets us either

- initialize the `dept` property of an `Employee` object to a specified value or
- accept the default value.

Example

```
function Employee(name, dept) {
    this.name = name || "";
    this.dept = dept || "general";
}

jim = new Employee;
mark = new Employee( "Doe, Mark" );
fred = new Employee( "Smith, Fred",
                    "engineering" );

document.writeln( jim.name );
document.writeln( "<br>" + jim.dept );

document.writeln( "<br><br>" + mark.name );
document.writeln( "<br>" + mark.dept );

document.writeln( "<br><br>" + fred.name );
document.writeln( "<br>" + fred.dept );
```

general

Doe, Mark
general

Smith, Fred
engineering

Object Oriented JavaScript

Instance Variables

Every object has its own copies of its instance variables.

By default, any object property is an instance variable.

To be true to object oriented programming, however, we might say that instance variables in JavaScript are those initialized in an object by the constructor function.

Example:

```
function Circle(x, y, r)
{
  this.x = x;
  this.y = y;
  this.r = r;
}
var c = new Circle(0, 1, 2.5);
```

Then `c` has three instance variables:

`c.x` is 0.

`c.y` is 1.

`c.r` is 2.5.

Instance Methods

We define an instance method for a class by setting a property in the constructor's prototype object to a function value.

- This can be done in several ways, all relying on the fact that functions are data – see below.

Then all objects created by that constructor share a reference to the function.

Instance methods use the `this` keyword to refer to the instance they're operating on.

Example:

This shows three ways to set a property in a constructor's prototype object to a function value:

1. First define a function then, in the constructor function definition, assign that function to the method.

For example, where function `Circle_circumference` has already been define,

```
this.circumference = Circle_circumference;
```

2. Assign an anonymous function to the method in the constructor – e.g.,

```
this.diameter = function() { return 2 * this.r; };
```

3. First define a function then, outside the constructor function definition, assign that function to the method in the prototype of the “class” – e.g., where `Circle_area` has already been defined,

```
Circle.prototype.area = Circle_area;
```

This also illustrates the special status of the method `toString()`:

As long as `toString()` is defined in `Circle` and `c` is an instance of `Circle`, an occurrence of `c` where a string is expected is the same as `c.toString`.

```
function Circle_circumference()
{
    return 2 * Math.PI * this.r;
}

function Circle(x, y, r)
{
    this.x = x;
    this.y = y;
    this.r = r;
    this.circumference = Circle_circumference;
    this.diameter =
        function() { return 2 * this.r; };
}

function Circle_area()
{
    return Math.PI * this.r * this.r;
}

Circle.prototype.area = Circle_area;
Circle.prototype.toString =
    function()
    {
        return "[Circle of radius " + this.r +
            ", centered at (" +
            this.x + ", " + this.y + ").]";
    };

var c = new Circle(0, 2.5, 1);

document.writeln( "The circle: ", c, "<br>" );
document.writeln( "Circumference: ",
    c.circumference(), "<br>",
    "Diameter: ",
    c.diameter(), "<br>",
    "Area: ", c.area() );
```

The circle: [Circle of radius 1, centered at (0, 2.5).]
Circumference: 6.283185307179586
Diameter: 2
Area: 3.141592653589793

Class Variables

A *class* (or *static*) *variable* in Java (not JavaScript) is a variable associated with a class itself.

There's always only one copy of a class variable.

An example of a JavaScript class variable is `Number.MAX_VALUE`.

Class variables are essentially global variables.

But they're associated with a class and have a position in the JavaScript name space where they're unlikely to be overwritten.

We simulate a class variable in JavaScript by defining a property of the constructor function itself (not of the prototype object)—e.g.,

```
Circle.PI = 3.14;
```

Class Methods

A *class* (or *static*) *method* is a method associated with a class rather than with an instance of it.

Class methods are invoked through the class.

`Date.parse()` is an example of a JavaScript class method.

Because class methods aren't invoked through a particular object, they can't meaningfully use the `this` keyword.

Class methods are essentially global.

But they have a position in the JavaScript name space, which prevents name collisions.

To define a class method in JavaScript, make the appropriate function a property of the constructor (not of the prototype object).

Example:

```
function Circle_max(a, b)
{
  if ( a.r > b.r )
    return a;
  else
    return b;
}
```

```
Circle.max = Circle_max;
```

Example:

We here define a class for complex numbers.

```
// First define the constructor function.

function Complex(real, imaginary)
{
  this.x = real;
  this.y = imaginary;
}

// Next define the instance methods.

Complex.prototype.magnitude = function() {
  return Math.sqrt(this.x * this.x +
                  this.y * this.y);
}

Complex.prototype.negative = function() {
  return new Complex(-this.x, -this.y);
}

Complex.prototype.toString = function() {
  return "{" + this.x + "," + this.y + "}";
}

Complex.prototype.valueOf = function() {
  return this.x;
}

// Now define the class methods.

Complex.add = function(a, b) {
  return new Complex(a.x + b.x, a.y + b.y);
}
```

Continued next page

Continued from previous page

```
Complex.subtract = function(a, b) {
    return new Complex(a.x - b.x, a.y - b.y);
}

Complex.multiply = function(a, b) {
    return new Complex(a.x * b.x - a.y * b.y,
                       a.x * b.y + a.y * b.x);
}

/* Some useful predefined complex numbers defined
 * as class variables, used as "constants"
 * (although they aren't really read-only).
 */

Complex.zero = new Complex(0,0);
Complex.one  = new Complex(1,0);
Complex.i    = new Complex(0,1);

// Test

var c = new Complex(2, 2);

document.writeln( "c: ", c, "<br>",
                  "The magnitude of c: ",
                    c.magnitude(), "<br>",
                  "The negative of c: ",
                    c.negative(), "<br>",
                  "The value of c: ",
                    c.valueOf(), "<br>",
                  "c plus 1: ",
                    Complex.add(c, Complex.one), "<br>",
                  "c minus i: ",
                    Complex.subtract(c, Complex.i), "<br>",
                  "c times c: ",
                    Complex.multiply(c, c) );
```

c: {2,2}

The magnitude of c: 2.8284271247461903

The negative of c: {-2,-2}

The value of c: 2

c plus 1: {3,2}

c minus i: {2,1}

c times c: {0,8}

Objects as Associative Arrays

Besides using the `.` operator to access the properties of an object, we can use the `[]` operator.

So the following have the same value:

```
object.property    – the property name is an identifier
object[property]  – the property name is a string
```

Recall that a JavaScript program can create any number of properties in any object.

But, when you use the `.` operator, the identifier denoting the property must be typed literally into your program.

On the other hand, when you access a property with the `[]` operator, the property is denoted by a string, which can be created at run-time.

Example:

```
var customer = new Object;

customer[ "firstName" ] = "John";
customer[ "lastName" ]  = "Smith";
customer[ "address0" ]  = "Apt. 1B";
customer[ "address1" ]  = "100 Elm St.";
customer[ "address2" ]  = "Greensboro, NC 27413";

var addr = "";
for ( var i = 0; i < 3; i ++ )
    addr += customer["address" + i] + "<br>";

document.writeln( addr );
```

```
| Apt. 1B
| 100 Elm St.
| Greensboro, NC 27413
|
```

Here `"address0"`, `"address1"`, `"address2"` are like property names, but, in the for loop, they're constructed at run-time.

There are cases where only the [] notation will do.

Example:

```
var portfolio = new Object(),
    value;

initialize( portfolio );

value = 0;

for ( var stock in portfolio )
    value += get_share_value( stock )
            * portfolio[ stock ];

document.writeln( "Your portfolio value is ",
                  value, "<br>" );

function initialize( portfolio )
{
    var stock_name;

    while ( stock_name
            = window.prompt( "Stock name", "" ) )
        portfolio[ stock_name ]
        = window.prompt( "Number of shares", 0 );
}

function get_share_value( stock )
{
    return window.prompt( "Value of " + stock, 0 );
}
```

When the user, presented with a prompt box, clicks the **Cancel** button or strikes the **Esc** key, the prompt call returns undefined, which counts as false.

So the while loop in this example terminates when the user, prompted for a stock name, clicks the **Cancel** button or strikes the **Esc** key.

An object used in this way is an *associative array*.

JavaScript objects are implemented internally as associative arrays.

An associative array is an instance of `Object` – *cf.* above,

```
var portfolio = new Object();
```

The real power of the `for/in` loop appears with associative arrays.

For example, in the example above, we have

```
for ( var stock in portfolio )  
    value += get_share_value( stock )  
            * portfolio[ stock ];
```

We can't write this without a `for/in` loop since the names of the stocks aren't known in advance.