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

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

```javascript
<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

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

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

```javascript
mark = new Employee;
```

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

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

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

```javascript
mark.age = 25;
```
Example
(The next two examples show .js files.)

```javascript
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

```javascript
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

```
depth || "general"
```
evaluates to “general”.

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

```
depth || "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:

```javascript
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,
   ```javascript
   this.circumference = Circle_circumference;
   ```

2. Assign an anonymous function to the method in the constructor – e.g.,
   ```javascript
   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,
   ```javascript
   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`. 
<script type = "text/javascript">
    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() );
</script>
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.,

```javascript
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 space 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.

```javascript
// 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
### Internet Systems

```javascript
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));
</script>
```
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}
Inheritance

To achieve inheritance, assign the prototypical instance to the `prototype` property of the child.

*Example:*

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

function Manager ( )  {
    this.reports = [ ];
}
Manager.prototype = new Employee;

function WorkerBee ( )  {
    this.projects = [ ];
}
WorkerBee.prototype = new Employee;
```
Suppose you create the mark object with

```javascript
mark = new WorkerBee;
```

When JavaScript sees the `new` operator, it creates a new generic object.

The new object is passed to the `WorkerBee` constructor as the value of the `this` keyword.

The constructor function explicitly sets the value of the `projects` property (to `[ ]`).

And it sets the value of the internal `__proto__` property to the value of `WorkerBee.prototype`.

The `__proto__` property determines the prototype chain to search for property values.

Then JavaScript returns the new object.

And it’s assigned to the variable `mark`.

This doesn’t explicitly put (local) values in the `mark` object for the properties `mark` inherits from the prototype chain.

When a property value is sought, JavaScript returns the value if it exists in the object in question.

If it doesn’t, the prototype chain is searched (using the `__proto__` property).
Local and inherited values can be changed by assignment – e.g.,

```javascript
mark.name = "Doe, Mark";
mark.projects = ["manhattan"];```

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

```javascript
mark.bonus = 3000;
```

If you add a new property to an object used as the prototype for a constructor function, you add that property to all objects inheriting properties from the prototype (including its instances).

For example,

```javascript
Employee.prototype.speciality = "none";
```

causes the `mark` object to have a `speciality` property with value “none”.

Because one prototype property can be inherited by many objects, JavaScript must enforce an asymmetry between reading and writing property values.

When you read property `p` of object `o`, JavaScript first checks whether `o` has a property `p`.

If not, it checks whether the prototype object of `o` has a property named `p`.

But, when you write the value of property `p` of object `o`, JavaScript doesn’t use the prototype object.

If it did, it would change the value of `p` for a whole class.
Example
(We show the .js file here.)

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

function Manager () {
    this.reports = [];
}
Manager.prototype = new Employee;

function WorkerBee () {
    this.projects = [];
}
WorkerBee.prototype = new Employee;

function SalesPerson () {
    this.dept = "sales";
    this.quota = 100;
}
SalesPerson.prototype = new WorkerBee;

function Engineer () {
    this.dept = "engineering";
    this.machine = ""
}
Engineer.prototype = new WorkerBee;
Continued from previous page

jim = new Engineer;

jim.name = "Jones, James";
jim.projects = ["manhattan"];
jim.machine = "cyclotron";

jim.age = 25;

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

The rendering is:

Jones, James
eering
manhattan
cyclotron
25
A constructor with parameters can add non-local properties if it directly calls the constructor function for an object higher in the prototype chain.

*Example*
(Here there is nothing special about the name `base`.)

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

function WorkerBee (name, dept, projs)  {
  this.base = Employee;
  this.base(name, dept);
  this.projects = projs || [];
}
WorkerBee.prototype = new Employee;

function Engineer (name, projs, mach)  {
  this.base = WorkerBee;
  this.base(name, "engineering", projs);
  this.machine = mach || "";
}
Engineer.prototype = new WorkerBee;
```
Suppose we create a new Engineer object:

```javascript
jim = new Engineer("Jones, James",
                 ["manhattan"], "cyclotron");
```

JavaScript takes these steps:

1. The `new` operator creates a generic object and sets its `__proto__` property to `Engineer.prototype`.

2. The `new` operator passes the new object to the `Engineer` constructor as the value of `this`.

3. The constructor creates for that object a new property called `base`, assigned the `WorkerBee` constructor.

   So the `WorkerBee` constructor becomes a method of the `Engineer` object.

4. The constructor calls the `base` method, passing it "Jones, James", "engineering", and ["manhattan"].

5. Because `base` is a method of `Engineer`, in the call to `base`, `this` is bound to the object created in Step 1.

   So the `WorkerBee` function in turn passes "Jones, James", and "engineering" to the `Employee` constructor function.

   On return from the `Employee` function, the `WorkerBee` function assigns the remaining argument to `projects`.

6. On return from the `base` method, the `Engineer` constructor initializes `machine`.

7. On return from the constructor, the new object is assigned to `jim`. 
For dynamic inheritance, it’s critical that we set up the prototype:

```javascript
Engineer.prototype = new WorkerBee;
```

Then, if we later add properties to the Employee or WorkerBee prototypes, they’re inherited by the Engineer object.

For example, suppose we have

```javascript
jim = new Engineer("Jones, James",
                   ["manhattan"], "cyclotron");
Employee.prototype.speciality = "none";
```

Then `jim` has a speciality property with value “name”.
Example

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

function Manager () {
    this.reports = [];
}
Manager.prototype = new Employee;

function WorkerBee (name, dept, projs) {
    this.base = Employee;
    this.base(name, dept);
    this.projects = projs || [];
}
WorkerBee.prototype = new Employee;

function SalesPerson () {
    this.dept = "sales";
    this.quota = 100;
}
SalesPerson.prototype = new WorkerBee;

function Engineer (name, projs, mach) {
    this.base = WorkerBee;
    this.base(name, "engineering", projs);
    this.machine = mach || "";
}
Engineer.prototype = new WorkerBee;
```

Continued next page.
jim = new Engineer("Jones, James",
                   ["manhattan"] , "cyclotron");

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

john = new Engineer("Smith, John",
                   ["hoboken"]) ;

document.writeln("<br><br>" + john.name);
document.writeln("<br>" + john.dept);
document.writeln("<br>" + john.projects);
document.writeln("<br>" + john.machine);

Jones, James
engineering
manhattan
cyclotron

Smith, John
engineering
hoboken
Overriding Method Definitions

Example:
The following shows inheritance and overriding of method definitions.

Class Circle, with instance methods circumference and area, is the superclass of class Cylinder.

Cylinder inherits the instance variables $x$, $y$, and $r$ from Circle, and adds another instance variable, $h$.

It inherits method circumference, overrides method area, and adds a new method, volume.

Note that the function object must be assigned to the volume property of the prototype after the prototype is assigned an instance of Circle.
<script type = "text/javascript">
  function Circle(x, y, r)
  {
    this.x = x;
    this.y = y;
    this.r = r;
    this.circumference =
      function() { return 2 * Math.PI * this.r; };
    this.area =
      function() { return Math.PI * this.r * this.r; };
  }

  function Cylinder(x, y, r, h)
  {
    this.base = Circle;
    this.base(x, y, r);
    this.h = h;
    this.area =
      function() { return 2 * Math.PI * this.r
                  * (this.r + h); };
  }

  Cylinder.prototype = new Circle();

  Cylinder.prototype.volume =
    function() { return this.h * Math.PI * this.r * this.r; };

  var c = new Circle(0, 0, 1);
  document.writeln( "Circumference: ",
                    c.circumference(), "<br>",
                    "Area: ", c.area(), "<br>" );

  var cy = new Cylinder(0, 0, 1, 2);
  document.writeln( "Circumference: ",
                    cy.circumference(), "<br>",
                    "Area: ", cy.area(), "<br>",
                    "Volume: ", cy.volume() );
</script>
Circumference: 6.283185307179586
Area: 3.141592653589793
Circumference: 6.283185307179586
Area: 12.566370614359172
Volume: 6.283185307179586
More on Inheritance

We can write an `instanceOf` function as follows:

```javascript
function instanceOf ( object, constructor ) {
    while (object != null) {
        if (object == constructor.prototype)
            return true;
        object = object.__proto__;
    }
    return false;
}
```

Then, for example, the following returns true:

```javascript
instanceOf ( jim, WorkerBee );
```

Because an object has a single associated prototype, JavaScript does not support multiple inheritance.

The illusion of multiple inheritance can be achieved with a constructor that calls more than one other constructor within it.

But this doesn’t support the dynamic characteristics we’d expect for multiple inheritance.
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:

```javascript
var addr = "";
for ( var i = 0; i < 4; i ++ )
    addr += customer["address" + i];
```

This reads and concatenates the “address0”, “address1”, “address2”, and “address3” properties of the customer object.
There are cases where only the [ ] notation will do.

*Example:*

Suppose we’re writing a program to compute the current value of the user’s stock market investments.

The user types in the name of each stock he owns along with the number of shares.

Suppose object `portfolio` holds this information.

It has one property for each stock.

The name of the property is the name of the stock.

The value of the property is number of shares of that stock.

The program needs a loop that prompts the user for each stock and the number of shares — the body should contain something like

```javascript
var stock_name = get_name_from_user();
var shares = get_number_of_shares();
portfolio[stock_name] = shares;
```

Since stock names are entered at runtime, we can’t know the property names when we write the program — we must use [ ].

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` — e.g.,

```javascript
petNames = new Object();
petNames[ "dog" ] = "Fido";
```
The real power of the for/in loop appears with associative arrays.

**Example:**

The following could be used in the portfolio example to compute the current total value:

```javascript
var value = 0;
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.