JavaScript

Article • 09/30/201929 minutes to read

Create Advanced Web Applications With Object-Oriented Techniques

Ray Djajadinata

Recently I interviewed a software developer with five years experience in developing Web applications. She'd been doing JavaScript for four and a half years, she rated her JavaScript skill as very good, and—as I found out soon after—she actually knew very little about JavaScript. I didn't really blame her for that, though. JavaScript is funny that way. It's the language a lot of people (including myself, until recently!) assume they're good at, just because they know C/C++/C# or they have some prior programming experience.

In a way, that assumption is not entirely groundless. It is easy to do simple things with JavaScript. The barrier to entry is very low; the language is forgiving and doesn't require you to know a lot of things before you can start coding in it. Even a non-programmer can probably pick it up and write some useful scripts for a homepage in a matter of hours.

Indeed, until recently, I'd always been able to get by with whatever little JavaScript I knew, armed only with the MSDN® DHTML reference and my C++/C# experience. It was only when I started working on real-world AJAX applications that I realized how inadequate my JavaScript actually was. The complexity and interactivity of this new generation of Web applications requires a totally different approach to writing JavaScript code. These are serious JavaScript applications! The way we've been writing our throwaway scripts simply doesn't cut it anymore.

Object-oriented programming (OOP) is one popular approach that's used in many JavaScript libraries to make a codebase more manageable and maintainable. JavaScript supports OOP, but it does so in a very different manner from the way popular Microsoft® .NET Framework compliant languages like C++, C#, or Visual Basic® do it, so developers who have been working extensively with those languages may find doing

OOP in JavaScript strange and counter-intuitive at first. I wrote this article to discuss in depth how the JavaScript language really supports object-oriented programming and how you can use this support to do object-oriented development effectively in JavaScript. Let's start by talking about (what else?) objects.

JavaScript Objects Are Dictionaries

In C++ or C#, when we're talking about objects, we're referring to instances of classes or structs. Objects have different properties and methods, depending on which templates (that is, classes) they are instantiated from. That's not the case with JavaScript objects. In JavaScript, objects are just collections of name/value pairs—think of a JavaScript object as a dictionary with string keys. We can get and set the properties of an object using either the familiar "." (dot) operator, or the "[]" operator, which is typically used when dealing with a dictionary. The following snippet

```
var userObject = new Object();

userObject.lastLoginTime = new Date();

alert(userObject.lastLoginTime);
```

does exactly the same thing as this:

```
var userObject = {}; // equivalent to new Object()

userObject["lastLoginTime"] = new Date();

alert(userObject["lastLoginTime"]);
```

We can also define the lastLoginTime property directly within userObject's definition like this:

```
var userObject = { "lastLoginTime": new Date() };

alert(userObject.lastLoginTime);
```

Note how similar it is to the C# 3.0 object initializers. Also, those of you familiar with Python will recognize that the way we instantiate userObject in the second and third snippets is exactly how we'd specify a dictionary in Python. The only difference is that a JavaScript object/dictionary only accepts string keys, rather than hashable objects like a Python dictionary would.

These examples also show how much more malleable JavaScript objects are than C++ or C# objects. Property lastLoginTime doesn't have to be declared beforehand—if userObject doesn't have a property by that name, it will simply be added to userObject. This isn't surprising if you remember that a JavaScript object is a dictionary—after all, we add new keys (and their respective values) to dictionaries all the time.

So, there we have object properties. How about object methods? Again, JavaScript is different from C++/C#. To understand object methods, I first need to take a closer look at JavaScript functions.

JavaScript Functions Are First Class

In many programming languages, functions and objects are usually considered two different things. In JavaScript, this distinction is blurred—a JavaScript function is really an object with executable code associated with it. Consider an ordinary function like this:

```
function func(x) {

    alert(x);

}

func("blah");
```

This is how we usually define a function in JavaScript. But you can also define the same function as follows, where you create an anonymous function object, and assign it to variable func

```
var func = function(x) {

    alert(x);

};

func("blah2");
```

or even like this, using the Function constructor:

```
var func = new Function("x", "alert(x);");

func("blah3");
```

This shows that a function is really just an object that supports a function call operation. That last way of defining a function using the Function constructor is not commonly used, but it opens up interesting possibilities because, as you may notice, the body of the function is just a String parameter to the Function constructor. That means you can construct arbitrary functions at run time.

To demonstrate further that a function is an object, you can set or add properties to a function, just like you would to any other JavaScript objects:

```
function sayHi(x) {

    alert("Hi, " + x + "!");

}

sayHi.text = "Hello World!";

sayHi["text2"] = "Hello World... again.";



alert(sayHi["text"]); // displays "Hello World!"

alert(sayHi.text2); // displays "Hello World... again."
```

As objects, functions can also be assigned to variables, passed as arguments to other functions, returned as the values of other functions, stored as properties of objects or elements of arrays, and so on. **Figure 1** provides an example of this.

Figure 1 **Functions Are First-Class in JavaScript**

```
// assign an anonymous function to a variable

var greet = function(x) {

    alert("Hello, " + x);

};

greet("MSDN readers");



// passing a function as an argument to another

function square(x) {

    return x * x;

}

function operateOn(num, func) {

    return func(num);

}

// displays 256

alert(operateOn(16, square));



// functions as return values

function makeIncrementer() {

    return function(x) { return x + 1; };

}
```

```
var inc = makeIncrementer();

// displays 8

alert(inc(7));



// functions stored as array elements

var arr = [];

arr[0] = function(x) { return x * x; };

arr[1] = arr[0](2);

arr[2] = arr[0](arr[1]);

arr[3] = arr[0](arr[2]);

// displays 256

alert(arr[3]);



// functions as object properties

var obj = { "toString" : function() { return "This is an object."; } };

// calls obj.toString()

alert(obj);
```

With that in mind, adding methods to an object is as easy as choosing a name and assigning a function to that name. So I define three methods in the object by assigning anonymous functions to the respective method names:

```
var myDog = {

    "name" : "Spot",

    "bark" : function() { alert("Woof!"); },

    "displayFullName" : function() {
        alert(this.name + " The Alpha Dog");

    },

    "chaseMrPostman" : function() {

        // implementation beyond the scope of this article

    }

};

myDog.displayFullName();

myDog.bark(); // Woof!
```

The use of the "this" keyword inside the function displayFullName should be familiar to the C++/C# developers among us—it refers to the object through which the method is called (developers who use Visual Basic should find it familiar, too—it’s called "Me" in Visual Basic). So in the example above, the value of "this" in the displayFullName is the myDog object. The value of "this" is not static, though. Called through a different object, the value of "this" will also change to point to that object as **Figure 2** demonstrates.

Figure 2 “**this**” **Changes as the Object Changes**

```
function displayQuote() {

    // the value of "this" will change; depends on

    // which object it is called through

    alert(this.memorableQuote);

}


var williamShakespeare = {

    "memorableQuote": "It is a wise father that knows his own child.",

    "sayIt" : displayQuote

};
```

```
var markTwain = {

    "memorableQuote": "Golf is a good walk spoiled.",

    "sayIt" : displayQuote

};


var oscarWilde = {

    "memorableQuote": "True friends stab you in the front."

    // we can call the function displayQuote

    // as a method of oscarWilde without assigning it

    // as oscarWilde’s method.

    //"sayIt" : displayQuote

};


williamShakespeare.sayIt(); // true, true

markTwain.sayIt(); // he didn’t know where to play golf


// watch this, each function has a method call()

// that allows the function to be called as a

// method of the object passed to call() as an

// argument.

// this line below is equivalent to assigning

// displayQuote to sayIt, and calling oscarWilde.sayIt().

displayQuote.call(oscarWilde); // ouch!
```

The last line in **Figure 2** shows an alternative way of calling a function as a method of an object. Remember, a function in JavaScript is an object. Every function object has a method named call, which calls the function as a method of the first argument. That is, whichever object we pass into call as its first argument will become the value of "this" in the function invocation. This will be a useful technique for calling the base class constructor, as we'll see later.

One thing to remember is never to call functions that contain "this" without an owning object. If you do, you will be trampling over the global namespace, because in that call, "this" will refer to the Global object, and that can really wreak havoc in your application. For example, below is a script that changes the behavior of JavaScript's global function isNaN. Definitely not recommended!

```
alert("NaN is NaN: " + isNaN(NaN));



function x() {

    this.isNaN = function() {

        return "not anymore!";

    };

}

// alert!!! trampling the Global object!!!

x();



alert("NaN is NaN: " + isNaN(NaN));
```

So we've seen ways to create an object, complete with its properties and methods. But if you notice all the snippets above, the properties and methods are hardcoded within the object definition itself. What if you need more control over the object creation? For example, you may need to calculate the values of the object's properties based on some parameters. Or you may need to initialize the object's properties to the values that you'll only have at run time. Or you may need to create more than one instance of the object, which is a very common requirement.

In C#, we use classes to instantiate object instances. But JavaScript is different since it doesn't have classes. Instead, as you'll see in the next section, you take advantage of the fact that functions act as constructors when used together with the "new" operator.

Constructor Functions but No Classes

The strangest thing about JavaScript OOP is that, as noted, JavaScript doesn't have classes like C# or C++ does. In C#, when you do something like this:

```
Dog spot = new Dog();
```

you get back an object, which is an instance of the class Dog. But in JavaScript there's no class to begin with. This closest you can get to a class is by defining a constructor function like this:

```
function DogConstructor(name) {

    this.name = name;

    this.respondTo = function(name) {

        if(this.name == name) {

            alert("Woof");

        }

    };

}



var spot = new DogConstructor("Spot");

spot.respondTo("Rover"); // nope

spot.respondTo("Spot"); // yeah!
```

OK, so what's happening here? Ignore the DogConstructor function definition for a moment and examine this line:

```
var spot = new DogConstructor("Spot");
```

What the "new" operator does is simple. First, it creates a new empty object. Then, the function call that immediately follows is executed, with the new empty object set as the value of "this" within that function. In other words, the line above with the "new" operator can be thought of as similar to the two lines below:

```
// create an empty object

var spot = {};

// call the function as a method of the empty object

DogConstructor.call(spot, "Spot");
```

As you can see in the body of DogConstructor, invoking this function initializes the object to which the keyword "this" refers during that invocation. This way, you have a way of creating a template for objects! Whenever you need to create a similar object, you call "new" together with the constructor function, and you get back a fully initialized object as a result. Sounds very similar to a class, doesn't it? In fact, usually in JavaScript the name of the constructor function is the name of the class you're simulating, so in the example above you can just name the constructor function Dog:

```
// Think of this as class Dog

function Dog(name) {

    // instance variable

    this.name = name;

    // instance method? Hmmm...

    this.respondTo = function(name) {
        if(this.name == name) {

            alert("Woof");

        }

    };

}



var spot = new Dog("Spot");
```

In the Dog definition above, I defined an instance variable called name. Every object that is created using Dog as its constructor function will have its own copy of the instance variable name (which, as noted earlier, is just an entry into the object's dictionary). This is expected; after all, each object does need its own copies of instance variables to carry its state. But if you look at the next line, every instance of Dog also has its own copy of the respondTo method, which is a waste; you only need one instance of respondTo to be shared among Dog instances! You can work around the problem by taking the definition of respondTo outside Dog, like this:

```
function respondTo() {

    // respondTo definition

}



function Dog(name) {

    this.name = name;

    // attached this function as a method of the object

    this.respondTo = respondTo;

}
```

This way, all instances of Dog (that is, all instances created with the constructor function

Dog) can share just one instance of the method respondTo. But as the number of methods grow, this becomes harder and harder to maintain. You end up with a lot of global functions in your codebase, and things only get worse as you have more and more "classes," especially if their methods have similar names. There's a better way to achieve this using the prototype objects, which are the topic of the next section.

Prototypes

The prototype object is a central concept in object-oriented programming with JavaScript. The name comes from the idea that in JavaScript, an object is created as a copy of an existing example (that is, a prototype) object. Any properties and methods of this prototype object will appear as properties and methods of the objects created from that prototype's constructor. You can say that these objects inherit their properties and methods from their prototype. When you create a new Dog object like this

```
var buddy = new Dog("Buddy");
```

the object referenced by buddy inherits properties and methods from its prototype, although it's probably not obvious from just that one line where the prototype comes from. The prototype of the object buddy comes from a property of the constructor function (which, in this case, is the function Dog).

In JavaScript, every function has a property named "prototype" that refers to a prototype object. This prototype object in turn has a property named "constructor," which refers back to the function itself. It's sort of a circular reference; **Figure 3** illustrates this cyclic relationship better.

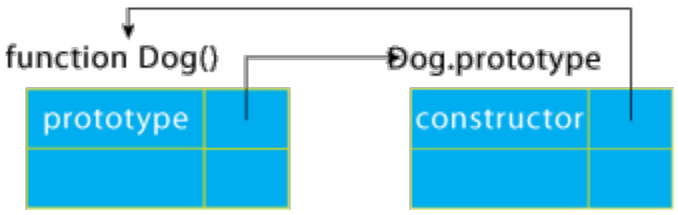

Figure 3** Every Function's Prototype Has a Constructor Property **

Now, when a function (in the example above, Dog) is used to create an object with the "new" operator, the resulting object will inherit the properties of Dog.prototype. In **Figure 3**, you can see that the Dog.prototype object has a constructor property that points back to the Dog function. Consequently, every Dog object (that inherits from Dog.prototype) will also appear to have a constructor property that points back to the Dog function. The code in **Figure 4** confirms this. This relationship between constructor function, prototype object, and the object created with them is depicted in **Figure 5**.

Figure 4 **Objects Appear to Have Their Prototype's Properties**

```
var spot = new Dog("Spot");



// Dog.prototype is the prototype of spot

alert(Dog.prototype.isPrototypeOf(spot));



// spot inherits the constructor property

// from Dog.prototype

alert(spot.constructor == Dog.prototype.constructor);

alert(spot.constructor == Dog);



// But constructor property doesn’t belong

// to spot. The line below displays "false"

alert(spot.hasOwnProperty("constructor"));



// The constructor property belongs to Dog.prototype

// The line below displays "true"

alert(Dog.prototype.hasOwnProperty("constructor"));
```

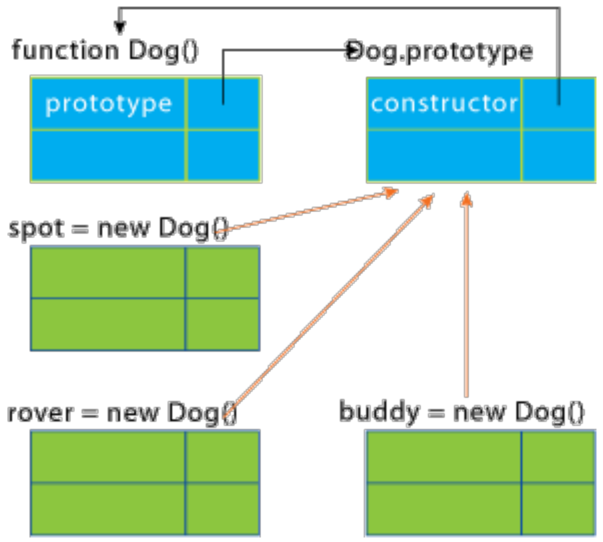

Figure 5** Instances Inherit from Their Prototype **

Some of you may have noticed the calls to hasOwnProperty and isPrototypeOf method in **Figure 4**. Where do these methods come from? They don't come from Dog.prototype. In fact, there are other methods like toString, toLocaleString, and valueOf that we can call on Dog.prototype and instances of Dog, but which don't come from Dog.prototype at all. It turns out that just like the .NET Framework has System.Object, which serves as the ultimate base class for all classes, JavaScript has Object.prototype, which is the ultimate base prototype for all prototypes. (The prototype of Object.prototype is null.)

In this example, remember that Dog.prototype is an object. It is created with a call to the Object constructor function, although it is not visible:

```
Dog.prototype = new Object();
```

So just like instances of Dog inherit from Dog.prototype, Dog.prototype inherits from Object.prototype. This makes all instances of Dog inherit Object.prototype's methods and properties as well.

Every JavaScript object inherits a chain of prototypes, all of which terminate with Object.prototype. Note that this inheritance you've seen so far is inheritance between live objects. It is different from your usual notion of inheritance, which happens between classes when they are declared. Consequently, JavaScript inheritance is much more dynamic. It is done using a simple algorithm, as follows: when you try to access a property/method of an object, JavaScript checks if that property/method is defined in that object. If not, then the object's prototype will be checked. If not, then that object's prototype's prototype will be checked, and so on, all the way to Object.prototype. **Figure 6** illustrates this resolution process.

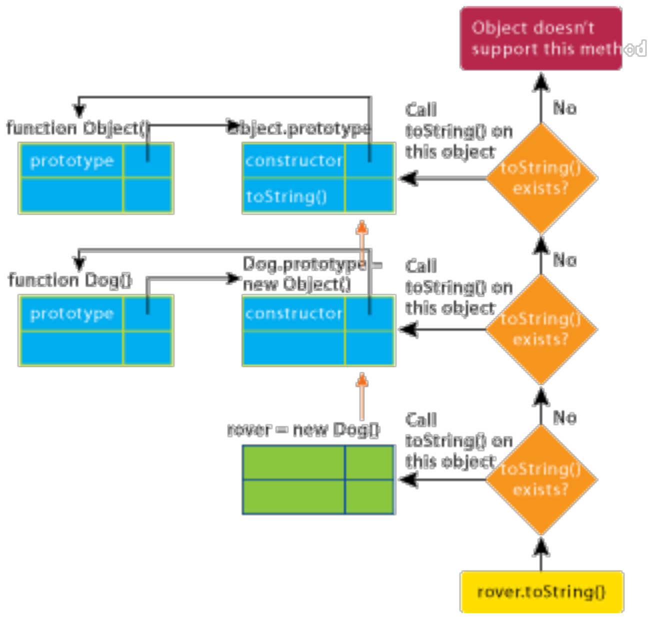

Figure 6** Resolving toString() Method in the Prototype Chain **(Click the image for a larger view)

The way JavaScript resolves properties access and method calls dynamically has some consequences:Changes made to a prototype object are immediately visible to the objects that inherit from it, even after these objects are created.If you define a property/method X in an object, a property/method of the same name will be hidden in that object's prototype. For instance, you can override Object.prototype's toString method by defining a toString method in Dog.prototype.Changes only go in one direction, from prototype to its derived objects, but not vice versa.

Figure 7 illustrates these consequences. **Figure 7** also shows how to solve the problem of unnecessary method instances as encountered earlier. Instead of having a separate instance of a function object for every object, you can make the objects share the method by putting it inside the prototype. In this example, the getBreed method is shared by rover and spot—until you override the toString method in spot, anyway. After that, spot has its own version of the getBreed method, but the rover object and subsequent objects created with new GreatDane will still share that one instance of the getBreed method defined in the GreatDane.prototype object.

Figure 7 **Inheriting from a Prototype**

```
function GreatDane() { }


var rover = new GreatDane();

var spot = new GreatDane();


GreatDane.prototype.getBreed = function() {

    return "Great Dane";

};


// Works, even though at this point

// rover and spot are already created.

alert(rover.getBreed());


// this hides getBreed() in GreatDane.prototype

spot.getBreed = function() {

    return "Little Great Dane";

};

alert(spot.getBreed());


// but of course, the change to getBreed

// doesn’t propagate back to GreatDane.prototype

// and other objects inheriting from it,

// it only happens in the spot object

alert(rover.getBreed());
```

Static Properties and Methods

Sometimes you need properties or methods that are tied to classes instead of instances—that is, static properties and methods. JavaScript makes this easy, since functions are objects whose properties and methods can be set as desired. Since a constructor function represents a class in JavaScript, you can add static methods and properties to a class simply by setting them in the constructor function like this:

```
function DateTime() { }



    // set static method now()

    DateTime.now = function() {

        return new Date();

    };



    alert(DateTime.now());
```

The syntax for calling the static methods in JavaScript is virtually identical to how you'd do it in C#. This shouldn't come as a surprise since the name of the constructor function is effectively the name of the class. So you have classes, and you have public properties/methods, and static properties/methods. What else do you need? Private members, of course. But JavaScript doesn't have native support for private members (nor for protected, for that matter). All properties and methods of an object are accessible to anyone. There is a way to have private members in your class, but to do so you first need to understand closures.

Closures

I didn't learn JavaScript of my own volition. I had to pick it up quickly because I realized that I was ill-prepared to work on a real-world AJAX application without it. At first, I felt like I had gone down a few levels in the programmer hierarchy. (JavaScript! What would my C++ friends say?) But once I got over my initial resistance, I realized that JavaScript was actually a powerful, expressive, and compact language. It even boasts features that other, more popular languages are only beginning to support.

One of JavaScript's more advanced features is its support for closures, which C# 2.0 supports through its anonymous methods. A closure is a runtime phenomenon that comes about when an inner function (or in C#, an inner anonymous method) is bound to the local variables of its outer function. Obviously, it doesn't make much sense unless this inner function is somehow made accessible outside the outer function. An example will make this clearer.

Let's say you need to filter a sequence of numbers based on a simple criterion that only numbers bigger than 100 can pass, while the rest are filtered out. You can write a function like the one in **Figure 8**.

Figure 8 **Filtering Elements Based on a Predicate**

```
function filter(pred, arr) {

    var len = arr.length;

    var filtered = []; // shorter version of new Array();

    // iterate through every element in the array...

    for(var i = 0; i &lt; len; i++) {

        var val = arr[i];

        // if the element satisfies the predicate let it through

        if(pred(val)) {

            filtered.push(val);

        }

    }

    return filtered;

}
```

```
var someRandomNumbers = [12, 32, 1, 3, 2, 2, 234, 236, 632,7, 8];

var numbersGreaterThan100 = filter(

    function(x) { return (x &gt; 100) ? true : false; },

    someRandomNumbers);



// displays 234, 236, 632

alert(numbersGreaterThan100);
```

But now you want to create a different filtering criterion, let's say this time only numbers bigger than 300. You can do something like this:

```
var greaterThan300 = filter(

    function(x) { return (x &gt; 300) ? true : false; },

    someRandomNumbers);
```

And then maybe you need to filter numbers that are bigger than 50, 25, 10, 600, and so on, but then, being the smart person you are, you realize that they're all the same predicate, "greater than." Only the number is different. So you can factor the number out with a function like this

```
function makeGreaterThanPredicate(lowerBound) {

    return function(numberToCheck) {

        return (numberToCheck &gt; lowerBound) ? true : false;

    };

}
```

which lets you do something like this:

```
var greaterThan10 = makeGreaterThanPredicate(10);

var greaterThan100 = makeGreaterThanPredicate(100);

alert(filter(greaterThan10, someRandomNumbers));

alert(filter(greaterThan100, someRandomNumbers));
```

Watch the inner anonymous function returned by the function makeGreaterThanPredicate. That anonymous inner function uses lowerBound, which is an argument passed to makeGreaterThanPredicate. By the usual rules of scoping, lowerBound goes out of scope when makeGreaterThanPredicate exits! But in this case, that inner anonymous function still carries lowerBound with it, even long after make-GreaterThanPredicate exits. This is what we call closure—because the inner function closes over the environment (that is, the arguments and local variables of the outer function) in which it is defined.

Closures may not seem like a big deal at first. But used properly, they open up interesting new possibilities in the way you can translate your ideas into code. One of the most interesting uses of closures in JavaScript is to simulate private variables of a class.

Simulating Private Properties

OK, so let's see how closures can help in simulating private members. A local variable in a function is normally not accessible from outside the function. After the function exits, for all practical purposes that local variable is gone forever. However, when that local variable is captured by an inner function's closure, it lives on. This fact is the key to simulating JavaScript private properties. Consider the following Person class:

```
function Person(name, age) {

    this.getName = function() { return name; };

    this.setName = function(newName) { name = newName; };

    this.getAge = function() { return age; };
    this.setAge = function(newAge) { age = newAge; };

}
```

The arguments name and age are local to the constructor function Person. The moment Person returns, name and age are supposed to be gone forever. However, they are captured by the four inner functions that are assigned as methods of a Person instance, in effect making name and age live on, but only accessible strictly through these four methods. So you can do this:

```
var ray = new Person("Ray", 31);

alert(ray.getName());

alert(ray.getAge());

ray.setName("Younger Ray");

// Instant rejuvenation!

ray.setAge(22);

alert(ray.getName() + " is now " + ray.getAge() +

      " years old.");
```

Private members that don’t get initialized in the constructor can be local variables of the constructor function, like this:

```
function Person(name, age) {

    var occupation;

    this.getOccupation = function() { return occupation; };

    this.setOccupation = function(newOcc) { occupation =

                         newOcc; };
    // accessors for name and age

}
```

Note that these private members are slightly different from what we'd expect from private members in C#. In C#, the public methods of a class can access its private members. But in JavaScript, the private members are accessible only through methods that have these private members within their closures (these methods are usually called privileged methods, since they are different from ordinary public methods). So within Person's public methods, you still have to access a private member through its privileged accessors methods:

```
Person.prototype.somePublicMethod = function() {

    // doesn't work!

    // alert(this.name);

    // this one below works

    alert(this.getName());

};
```

Douglas Crockford is widely known as the first person to discover (or perhaps publish) the technique of using closures to simulate private members. His Web site, javascript.crockford.com, contains a wealth of information on JavaScript—any developer interested in JavaScript should check it out.

Inheriting from Classes

OK, you've seen how constructor functions and prototype objects allow you to simulate classes in JavaScript. You've seen that the prototype chain ensures that all objects have the common methods of Object.prototype. You've seen how you can simulate private members of a class using closures. But something is missing here. You haven't seen how you can derive from your class; that's an everyday activity in C#. Unfortunately, inheriting from a class in JavaScript is not simply a matter of typing a colon like in C#; it takes more than that. On the other hand, JavaScript is so flexible that there are a lot of ways of inheriting from a class.

Let’s say, for example, you have a base class Pet, with one derived class Dog, as in **Figure 9**. How do you go about this in JavaScript? The Pet class is easy. You’ve seen how you can do this:

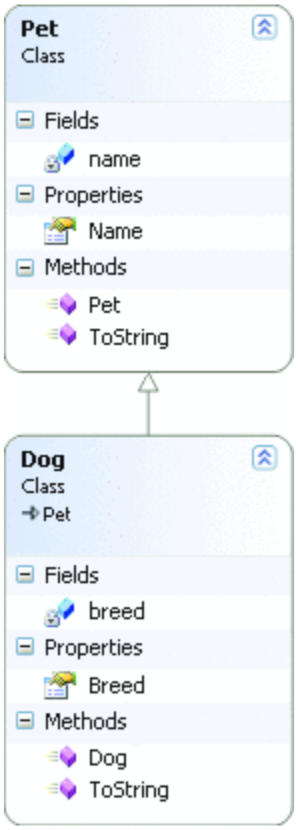

Figure 9** Classes **

```
// class Pet

function Pet(name) {

    this.getName = function() { return name; };

    this.setName = function(newName) { name = newName; };

}



Pet.prototype.toString = function() {

    return "This pet’s name is: " + this.getName();

};

// end of class Pet



var parrotty = new Pet("Parrotty the Parrot");

alert(parrotty);
```

Now what if you want to create a class Dog, which derives from Pet? As you can see in **Figure 9**, Dog has an extra property, breed, and it overrides Pet's toString method (note that convention for JavaScript is to use camel casing for methods and properties names, instead of Pascal casing as is recommended with C#). **Figure 10** shows how it is done.

Figure 10 **Deriving from the Pet Class**

```
// class Dog : Pet

// public Dog(string name, string breed)

function Dog(name, breed) {

    // think Dog : base(name)

    Pet.call(this, name);

    this.getBreed = function() { return breed; };

    // Breed doesn't change, obviously! It's read only.

    // this.setBreed = function(newBreed) { name = newName; };

}



// this makes Dog.prototype inherits

// from Pet.prototype

Dog.prototype = new Pet();
```

```
// remember that Pet.prototype.constructor

// points to Pet. We want our Dog instances’

// constructor to point to Dog.

Dog.prototype.constructor = Dog;



// Now we override Pet.prototype.toString

Dog.prototype.toString = function() {

    return "This dog’s name is: " + this.getName() +

        ", and its breed is: " + this.getBreed();

};

// end of class Dog



var dog = new Dog("Buddy", "Great Dane");

// test the new toString()

alert(dog);



// Testing instanceof (similar to the is operator)

// (dog is Dog)? yes

alert(dog instanceof Dog);

// (dog is Pet)? yes

alert(dog instanceof Pet);

// (dog is Object)? yes

alert(dog instanceof Object);
```

The prototype-replacement trick used sets the prototype chain properly, so instanceof tests work as expected if you were using C#. Also, the privileged methods still work as expected.

Simulating Namespaces

In C++ and C#, namespaces are used to minimize the probability of name collisions. In the .NET Framework, namespaces help differentiate Microsoft.Build.Task.Message class from System.Messaging.Message, for example. JavaScript doesn’t have any specific language features to support namespaces, but it’s easy to simulate a namespace using objects. Let’s say you want to create a JavaScript library. Instead of defining functions and classes globally, you can wrap them in a namespace like this:

```
var MSDNMagNS = {};



MSDNMagNS.Pet = function(name) { // code here };

MSDNMagNS.Pet.prototype.toString = function() { // code };



var pet = new MSDNMagNS.Pet("Yammer");
```

One level of namespace may not be unique, so you can create nested namespaces:

```
var MSDNMagNS = {};

// nested namespace "Examples"

MSDNMagNS.Examples = {};



MSDNMagNS.Examples.Pet = function(name) { // code };

MSDNMagNS.Examples.Pet.prototype.toString = function() { // code };



var pet = new MSDNMagNS.Examples.Pet("Yammer");
```

As you can imagine, typing those long nested namespaces can get tiresome pretty fast. Fortunately, it’s easy for the users of your library to alias your namespace into something shorter:

```
// MSDNMagNS.Examples and Pet definition...



// think "using Eg = MSDNMagNS.Examples;"

var Eg = MSDNMagNS.Examples;

var pet = new Eg.Pet("Yammer");

alert(pet);
```

If you take a look at the source code of the Microsoft AJAX Library, you’ll see that the library’s authors use a similar technique to implement namespaces (take a look at the implementation of the static method Type.registerNamespace). See the sidebar "OOP and ASP.NET AJAX" for more information.

Should You Code JavaScript This Way?

You’ve seen that JavaScript supports object-oriented programming just fine. Although it was designed as a prototype-based language, it is flexible and powerful enough to accommodate the class-based programming style that is typically found in other popular languages. But the question is: should you code JavaScript this way? Should you code in JavaScript the way you code in C# or C++, coming up with clever ways to simulate features that aren't there? Each programming language is different, and the best practices for one language may not be the best practices for another.

In JavaScript, you've seen that objects inherit from objects (as opposed to classes inheriting from classes). So it is possible that making a lot of classes using a static inheritance hierarchy is not the JavaScript way. Maybe, as Douglas Crockford says in his article "Prototypal Inheritance in JavaScript ", the JavaScript way of programming is to make prototype objects, and use the simple object function below to make new objects, which inherit from that original object:

```
function object(o) {

        function F() {}

        F.prototype = o;

        return new F();

    }
```

Then, since objects in JavaScript are malleable, you can easily augment the object after its creation with new fields and new methods as necessary.

This is all good, but it is undeniable that the majority of developers worldwide are more familiar with class-based programming. Class-based programming is here to stay, in fact. According to the upcoming edition 4 of ECMA-262 specification (ECMA-262 is the official specification for JavaScript), JavaScript 2.0 will have true classes. So JavaScript is moving towards being a class-based language. However, it will probably take years for JavaScript 2.0 to reach widespread use. In the meantime, it's important to know the current JavaScript well enough to read and write JavaScript code in both prototype-based style and class-based style.

Putting It into Perspective

With the proliferation of interactive, client-heavy AJAX applications, JavaScript is quickly becoming one of the most useful tools in a .NET developer's arsenal. However, its prototypal nature may initially surprise developers who are more used to languages such as C++, C#, or Visual Basic. I have found my JavaScript journey a rewarding experience, although not entirely without frustration along the way. If this article can help make your experience smoother, then I'm happy, for that's my goal.

OOP and ASP.NET AJAX

OOP as implemented in ASP.NET AJAX is a little different from the canonical implementation discussed in this article. There are two main reasons for that: the ASP.NET AJAX version offers more possibilities for reflection (which is necessary for declarative syntaxes such as xml-script and for parameter validation), and ASP.NET AJAX aims at translating to JavaScript some additional constructs that are familiar to developers using .NET, such as properties, events, enumerations, and interfaces.

In its current widely available version, JavaScript lacks a few key concepts of OOP that .NET developers are familiar with, and ASP.NET AJAX emulates most of those.

Classes can have property accessors based on a naming convention (example to follow), as well as multicast events following a pattern that closely mirrors that provided by .NET. Private variables are based on the convention that members starting with an underscore are private. Truly private variables are rarely necessary, and this policy enables those variables to be inspected from a debugger. Interfaces are also introduced to enable type checking scenarios beyond the usual duck-typing (a type scheme based on the concept that if something walks like a duck and quacks like a duck, it's a duck, or at least it can be treated like one).

Classes and Reflection

In JavaScript, there is no way to know the name of a function. Even if this was possible, it wouldn't help us in most situations as a class constructor is typically built by assigning an anonymous function to a namespaced variable. What really constitutes the type name is the fully qualified name of this variable, which is equally inaccessible and of which the constructor function itself knows nothing. To work around this limitation and have rich reflection over JavaScript classes, ASP.NET AJAX requires the names of types to be registered.

The reflection APIs in ASP.NET AJAX work over any type, whether it's built-in, a class, an interface, a namespace, or even an enumeration, and they include .NET Framework-like functions like isInstanceOfType and inheritsFrom to inspect the class hierarchy at run time. ASP.NET AJAX also does some type checking in debug mode where it makes sense to help developers catch bugs earlier.

Registering Class Hierarchies and Calling Base

To define a class in ASP.NET AJAX, you need to assign its constructor to a variable (notice how the constructor calls base):

```
MyNamespace.MyClass = function() {
 MyNamespace.MyClass.initializeBase(this);
 this._myProperty = null;
}
```

Then, you need to define the class members itself in its prototype:

```
MyNamespace.MyClass.prototype = {
 get_myProperty: function() { return this._myProperty;},
 set_myProperty: function(value) { this._myProperty = value; },
 doSomething: function() {
 MyNamespace.MyClass.callBaseMethod(this, "doSomething");
 /* do something more */
 }
}
```

And finally you register the class:

```
MyNamespace.MyClass.registerClass(
 "MyNamespace.MyClass ", MyNamespace.BaseClass);
```

There is no need here to manage the constructor and prototype hierarchies as this is done for you by the registerClass function.

Bertrand Le Roy is a Software Design Engineer II on the ASP.NET AJAX team.

Ray Djajadinata is having fun developing AJAX applications in Barclays Capital Singapore. You can reach Ray at ray.djajadinata@gmail.com.