3
Object-Oriented JavaScript

In this chapter, you'll learn about OOP (Object-Oriented Programming) and how it relates to JavaScript. As an ASP.NET developer, you probably have some experience working with objects, and you may even be familiar with concepts such as *inheritance*. However, unless you're already an experienced JavaScript programmer, you probably aren't familiar with the way JavaScript objects and functions *really* work. This knowledge is necessary in order to understand how the Microsoft AJAX Library works, and this chapter will teach you the necessary foundations. More specifically, you will learn:

- What encapsulation, inheritance, and polymorphism mean
- How JavaScript functions work
- How to use anonymous functions and closures
- How to read a class diagram, and implement it using JavaScript code
- How to work with JavaScript prototypes
- How the execution context and scope affect the output of JavaScript functions
- How to implement inheritance using closures and prototypes
- What JSON is, and what a JSON structure looks like

In the next chapters you'll use this theory to work effectively with the Microsoft AJAX Library.

Concepts of Object-Oriented Programming

Most ASP.NET developers are familiar with the fundamental OOP principles because this knowledge is important when developing for the .NET development. Similarly, to develop client-side code using the Microsoft AJAX Library, you need to be familiar with JavaScript's OOP features. Although not particularly difficult, understanding these features can be a bit challenging at first, because JavaScript's OOP model is different than that of languages such as C#, VB.NET, C++, or Java.

JavaScript is an *object-based* language. Just as in C#, you can create objects, call their methods, pass them as parameters, and so on. You could see this clearly when working with the DOM, where you manipulated the HTML document through the methods and properties of the implicit `document` object. However, JavaScript isn't generally considered a fully object-oriented language because it lacks support for some features that you'd find in "real" OOP languages, or simply implements them differently.

Your most important goal for this chapter is to understand how to work with JavaScript objects. As an ASP.NET developer, we assume that you already know how OOP works with .NET languages, although advanced knowledge isn't necessary. A tutorial written by Cristian Darie on OOP development with C# can be downloaded in PDF format at `http://www.cristiandarie.ro/downloads/`.

To ensure we start off from the same square, in the following couple of pages we'll review the essential OOP concepts as they apply in C# and other languages—objects, classes, encapsulation, inheritance, and polymorphism. Then we'll continue by "porting" this knowledge into the JavaScript realm.

Objects and Classes

What does "object-oriented programming" mean anyway? Basically, as the name suggests, OOP puts objects at the centre of the programming model. The **object** is probably the most important concept in the world of OOP—a self-contained entity that has *state* and *behavior*, just like a real-world object. Each object is an instance of a **class** (also called **type**), which defines the behavior that is shared by all its objects.

We often use objects and classes in our programs to represent real-world objects, and types (classes) of objects. For example, we can have classes like `Car`, `Customer`, `Document`, or `Person`, and objects such as `myCar`, `johnsCar`, or `davesCar`.

The concept is intuitive: the class represents the blueprint, or model, and objects are particular *instances* of that model. For example, all objects of type `Car` will have the same **behavior**—for example, the ability to change gear. However, each individual `Car` object may be in a different gear at any particular time—each object has its particular **state**. In programming, an object's state is described by its *fields* and *properties*, and its behavior is defined by its *methods* and *events*.

You've already worked with objects in the previous chapter. First, you've worked with the built-in `document` object. This is a default DOM object that represents the current page, and it allows you to alter the state of the page. However, you also learned how to create your own objects, when you created the `xmlHttp` object. In that case, `xmlHttp` is an object of the `XMLHttpRequest` class. You could create more `XMLHttpRequest` objects, and all of them would have the same abilities (behavior), such as the ability to contact remote servers as you learned earlier, but each would have a different state. For example, each of them may be contacting a different server.

In OOP's world everything revolves around objects and classes, and OOP languages usually offer three specific features for manipulating them—**encapsulation**, **inheritance**, and **polymorphism**.

Encapsulation

Encapsulation is a concept that allows the use of an object without having to know its internal implementation in detail. The interaction with an object is done only via its public interface, which contains public members and methods. We can say that encapsulation allows an object to be treated as a "black box", separating the implementation from its interface. Think of the objects you've worked with so far: `document`, a DOM object, and `xmlHttp`, an `XMLHttpRequest` object. You certainly don't know how these objects do their work internally! All you have to know is the features you can use.

The "features you can use" of a class form the *public interface* of a class, which is the sum of all its public members. The public members are those members that are visible and can be used by external classes. For example, the `innerHTML` property of a DOM object (such as the default `document` object), or the `open()` and `send()` methods of `XMLHttpRequest`, are all public, because you were able to use them. Each class can also contain `private` members, which are meant for internal usage only and aren't visible from outside.

Inheritance

Inheritance allows creating classes that are specialized versions of an existing class. For example assume that you have the `Car` class, which exposes a default interface for objects such as `myCar`, `johnsCar`, or `davesCar`. Now, assume that you want to introduce in your project the concept of a **supercar**, which would have similar functionality to the car, but some extra features as well, such as the capability to fly!

If you're an OOP programmer, the obvious move would be to create a new class named `SuperCar`, and use this class to create the necessary objects such as `mySuperCar`, or `davesSuperCar`. In such scenarios, inheritance allows you to create the `SuperCar` class based on the `Car` class, so you don't need to code all the common features once again. Instead, you can create `SuperCar` as a specialized version of `Car`, in which case `SuperCar` *inherits* all the functionality of `Car`. You would only need to code the additional features you want for your `SuperCar`, such as a method named `Fly`. In this scenario, `Car` is the *base class* (also referred to as *superclass*), and `SuperCar` is the *derived* class (also referred to as *subclass*).

Inheritance is great because it encourages code reuse. The potential negative side effect is that inheritance, by its nature, creates an effect that is known as *tight coupling* between the base class and the derived classes. Tight coupling refers to the fact that any changes that are made to a base class are automatically propagated to all the derived classes. For example, if you make a performance improvement in the code of the original `Car` class, that improvement will propagate to `SuperCar` as well. While this usually can be used to your advantage, if the inheritance hierarchy isn't wisely designed such coupling can impose future restrictions on how you can expand or modify your base classes without breaking the functionality of the derived classes.

Polymorphism

Polymorphism is a more advanced OOP feature that allows using objects of different classes when you only know a common base class from which they both derive. Polymorphism permits using a base class reference to access objects of that class, or objects of derived classes. Using polymorphism, you can have, for example, a method that receives as parameter an object of type `Car`, and when calling that method you supply as parameter an object of type `SuperCar`. Because `SuperCar` is a specialized version of `Car`, all the public functionality of `Car` would also be supported by `SuperCar`, although the `SuperCar` implementations could differ from those of `Car`. This kind of flexibility gives much power to an experienced programmer who knows how to take advantage of it.

Object-Oriented JavaScript

Objects and classes are implemented differently in JavaScript than in languages such as C#, VB.NET, Java, or C++. However, when it comes to *using* them, you'll feel on familiar ground. You create objects using the `new` operator, and you call their methods, or access their fields using the syntax you already know from C#. Here are a few examples of creating objects in JavaScript:

```
// create a generic object
var obj = new Object();

// create a Date object
var oToday = new Date();

// create an Array object with 3 elements
var oMyList = new Array(3);

// create an empty String object
var oMyString = new String();
```

Object creation is, however, the only significant similarity between JavaScript objects and those of "typical" OOP languages. The upcoming JavaScript 2.0 will reduce the differences by introducing the concept of *classes*, private members, and so on, but until then we have to learn how to live without them.

Objects in JavaScript have the following particularities. In the following pages we'll discuss each of them in detail:

- JavaScript code is not compiled, but parsed. This allows for flexibility when it comes to creating or altering objects. As you'll see, it's possible to add new members or functions to an object or even several objects by altering their *prototype*, on the fly.
- JavaScript doesn't support the notion of *classes* as typical OOP languages do. In JavaScript, you create *functions* that can behave—in many cases—just like classes. For example, you can call a function supplying the necessary parameters, or you can create an instance of that function supplying those parameters. The former case can be associated with a C# method call, and the later can be associated with instantiating a class supplying values to its constructor.
- JavaScript functions are first-class objects. In English, this means that the *function* is regarded, and can be manipulated, just as like other data types. For example, you can pass functions as parameters to other functions, or even return functions. This concept may be difficult to grasp since it's very different from the way C# developers normally think of functions or methods, but you'll see that this kind of flexibility is actually cool.
- JavaScript supports *closures*.
- JavaScript supports *prototypes*.

Ray Djajadinata's JavaScript article at `http://msdn.microsoft.com/msdnmag/issues/07/05/JavaScript/` covers the OOP features in JavaScript very well, and you can refer to it if you need another approach at learning these concepts.

JavaScript Functions

A simple fact that was highlighted in the previous chapter, but that is often overlooked, is key to understanding how objects in JavaScript work: code that doesn't belong to a function is executed when it's read by the JavaScript interpreter, while code that belongs to a function is only executed when that function is called.

Take the following JavaScript code that you created in the first exercise of Chapter 2:

```
// declaring new variables
var date = new Date();
var hour = date.getHours();

// simple conditional content output
if (hour >= 22 || hour <= 5)
  document.write("Goodnight, world!");
else
  document.write("Hello, world!");
```

This code resides in a file named `JavaScriptDom.js`, which is referenced from an HTML file (`JavaScriptDom.html` in the exercise), but it could have been included directly in a `<script>` tag of the HTML file. How it's stored is irrelevant; what does matter is that all that code is executed when it's read by the interpreter. If it was included in a function it would only execute if the function is called explicitly, as is this example:

```
// call function to display greeting message
ShowHelloWorld();

// "Hello, World" function
function ShowHelloWorld()
{
  // declaring new variables
  var date = new Date();
  var hour = date.getHours();

  // simple conditional content output
  if (hour >= 22 || hour <= 5)
    document.write("Goodnight, world!");
  else
    document.write("Hello, world!");
}
```

This code has the same output as the previous version of the code, but it is only because the `ShowHelloWorld()` function is called that will display "**Goodnight, world!**" or "**Hello, world!**" depending on the hour of the day. Without that function call, the JavaScript interpreter would take note of the existence of a function named `ShowHelloWorld()`, but would not execute it.

Functions as Variables

In JavaScript, functions are first-class objects. This means that a function is regarded as a data type whose values can be saved in local variables, passed as parameters, and so on. For example, when defining a function, you can assign it to a variable, and then call the function through this variable. Take this example:

```
// displays greeting
var display = function DisplayGreeting(hour)
{
  if (hour >= 22 || hour <= 5)
    document.write("Goodnight, world!");
  else
    document.write("Hello, world!");
}

// call DisplayGreeting supplying an hour as parameter
display(10);
```

When storing a piece of code as a variable, as in this example, it can make sense to create it as an anonymous function—which is, a function without a name. You do this by simply omitting to specify a function name when creating it:

```
// displays greeting
var display = function(hour)
{
  ...
}
```

Anonymous functions will come in handy in many circumstances when you need to pass an executable piece of code that you don't intend to reuse anywhere else, as parameter to a function.

Let's see how we can send functions as parameters. Instead of sending a numeric hour to `DisplayGreeting()`, we can send a function that in turn returns the current hour. To demonstrate this, we create a function named `GetCurrentHour()`, and send it as parameter to `DisplayGreeting()`. `DisplayGreeting()` needs to be modified to reflect that its new parameter is a function—it should be referenced by appending parentheses to its name. Here's how:

```
// returns the current hour
function GetCurrentHour()
{
  // obtaining the current hour
  var date = new Date();
  var hour = date.getHours();

  // return the hour
  return hour;
}

// display greeting
function DisplayGreeting(hourFunc)
{
```

```
    // retrieve the hour using the function received as parameter
    hour = hourFunc();

    // display greeting
    if (hour >= 22 || hour <= 5)
      document.write("Goodnight, world!");
    else
      document.write("Hello, world!");
}

// call DisplayGreeting
DisplayGreeting(GetCurrentHour);
```

This code can be tested online at `http://www.cristiandarie.ro/asp-ajax/Delegate.html`. The output should resemble Figure 3-1.

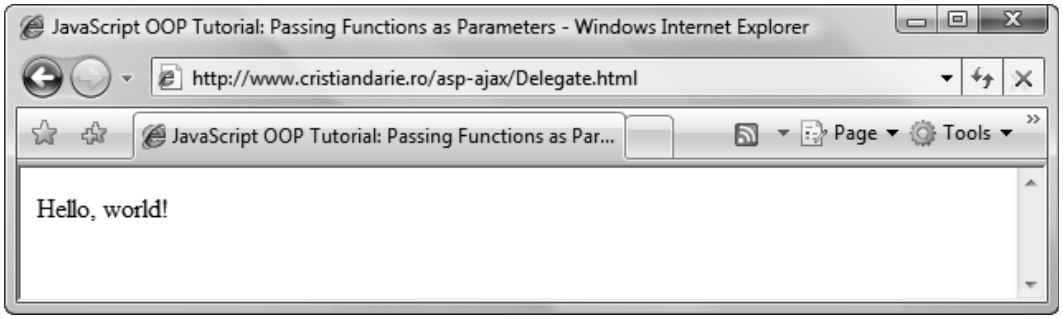

Figure 3-1. Simple demonstration of how a function can be sent as parameter to another function

.NET languages such as C# and VB.NET support similar functionality through the concept of **delegates**. A delegate is a data type that represents a reference to a function. An instance of a delegate represents a function instance, and it can be passed as a parameter to methods that need to execute that function. Delegates are the technical means used by .NET to implement event-handling. C# 2.0 added support for anonymous methods, which behave similarly to JavaScript anonymous functions.

Anonymous Functions

Anonymous functions can be created adhoc and used instead of a named function. Although this can hinder readability when the function is more complex, you can do this if you don't intend to reuse a function's code. In the following example we pass such an anonymous function to `DisplayGreeting()`, instead of passing `GetCurrentHour()`:

```
// call DisplayGreeting
DisplayGreeting(
  function()
  {
    return (new Date()).getHours();
  }
);
```

This syntax is sure to look strange if this is the first time you have worked with anonymous functions. You can compact it on a single line if it helps understanding it better:

```
DisplayGreeting( function() { return (new Date()).getHours(); } );
```

This code can be tested online at `http://www.cristiandarie.ro/asp-ajax/AnonymousFunction.html`.

Inner Functions and JavaScript Closures

JavaScript functions implement the concept of **closures**, which are functions that are defined inside other functions, and use contextual data from the parent functions to execute. You can find a complete and technically accurate definition of closures at `http://en.wikipedia.org/wiki/Closure_(computer_science)`.

In JavaScript a function can be regarded as a named block of code that you can execute, but it can also be used as a data member inside another function, in which case it is referred to as an *inner functions*. In other words, a JavaScript function can contain other functions.

Say that we want to upgrade the initial `ShowHelloWorld()` function by separating the code that displays the greeting message into a separate function inside `ShowHelloWorld()`. This is a possible implementation, and the output continues to be the same as before:

```
// call function to display greeting message
ShowHelloWorld();

// "Hello, World" function
function ShowHelloWorld()
{
  // declaring new variables
  var date = new Date();
  var hour = date.getHours();

  // call DisplayGreeting supplying the current hour as parameter
  DisplayGreeting(hour);
```

```
    // display greeting
    function DisplayGreeting(hour)
    {
      if (hour >= 22 || hour <= 5)
        document.write("Goodnight, world!");
      else
        document.write("Hello, world!");
    }
}
```

Here, we created a function named `DisplayGreeting()` inside `ShowHelloWorld()`, which displays a greeting message depending on the `hour` parameter it receives. The execution rules apply here as well. This new function needs to be called explicitly from its parent function in order to execute.

This code can be tested online at `http://www.cristiandarie.ro/asp-ajax/JavaScriptClosure.html`.

JavaScript Classes

Not only can JavaScript functions contain other functions, but they can also be instantiated. This makes JavaScript functions a good candidate for implementing the concept of a class from traditional object-oriented programming. This is very helpful feature indeed, because JavaScript doesn't support the notion of a class in the classic sense of the word. Functions can be instantiated using the `new` operator, such as in this example:

```
var myHelloWorld = new ShowHelloWorld();
```

This line of code effectively creates an object named `myHelloWorld`, which represents an instance of the `ShowHelloWorld()` function. When the object is instantiated, the function code is executed, so creating the object has the same effect as calling `ShowHelloWorld()` as in the previous examples.

Here are a few facts that will help you port your C# OOP knowledge into the JavaScript world:

- When a function is used as a class, its body code is considered to be the *constructor*. In classic OOP, the constructor is a special method that doesn't return anything, and that is called automatically when the object is created. The same effect happens in JavaScript when creating an instance of the function: its code executes. A C# constructor is equivalent to the code in the JavaScript function—without including any inner functions (whose code doesn't execute automatically).

- In C# constructors can receive parameters, and also in JavaScript. If the code in a function represents the "class constructor", the parameters received by that function play the role of constructor parameters.
- Class fields in JavaScript are created and referenced with the `this` keyword. In a JavaScript function, `this.myValue` is a public member of the function (class), while `myValue` is a local variable that can't be accessed through function instances. Also, the local variable is destroyed after the function executes, while class fields persist their value for the entire object lifetime.
- Class methods that need to be accessible from outside the class need to be referred to using `this` as well. Otherwise the inner function will be regarded as a local function variable, rather than a "class" member.

We'll demonstrate these concepts by transforming the `ShowHelloWorld()` function that you saw earlier into a "real" class. We will:

- Change the name of the function from `ShowHelloWorld()` to `HelloWorld()`.
- Add a parameter named `hour` to the function's "constructor" so that we tell the class the hour for which we need a greeting message, when instantiating it. If this parameter is passed when creating objects of the class, we store it for future use as a class field. If this parameter is not specified, the current hour of the day should be stored instead.
- The method `DisplayGreeting()` of the class should not support the `hour` parameter any longer. Instead, it should display the greeting message depending on the hour field that was initialized by the constructor.

Why are we changing the name of the function? Remember, OOP is a style of coding, not a list of technical requirements that a language must support. JavaScript is considered an OOP-capable language because it supports an object-based programming style. In the OOP paradigm, a class should represent an entity, and not an action. Since we intend now to use `ShowHelloWorld()` as a class, we are changing its name to one that reflects this purpose.

Once your new class is created, you use it just as you'd use a C# class. For example, this is how you'd create a new class instance, and call its `DisplayGreeting()` method:

```
// create class instance
var myHello = new HelloWorld();

// call method
myHello.DisplayGreeting();
```

A possible implementation of the `HelloWorld` class is the following:

```
// "Hello, World" class
function HelloWorld(hour)
{
  // class "constructor" initializes this.hour field
  if (hour)
  {
    // if the hour parameter has a value, store it as a class field
    this.hour = hour;
  }
  else
  {
    // if the hour parameter doesn't exist, save the current hour
    var date = new Date();
    this.hour = date.getHours();
  }

  // display greeting
  this.DisplayGreeting = function()
  {
    if (this.hour >= 22 || this.hour <= 5)
      document.write("Goodnight, world!");
    else
      document.write("Hello, world!");
  }
}
```

This code can be tested online at `http://www.cristiandarie.ro/asp-ajax/JavaScriptClass.html`. The `HelloWorld` class is formed of the constructor code that initializes the `hour` field (`this.hour`), and of the `DisplayGreeting()` method—`this.DisplayGreeting()`. Fans of the ternary operator can rewrite the constructor using this shorter form, which also makes use of the object detection feature that was discussed in Chapter 2:

```
// define and initialize this.hour
this.hour = (hour) ? hour : (new Date()).getHours();
```

The ternary operator is supported both by C# and JavaScript. It has the form `(condition ? valueA : valueB)`. If the condition is `true`, the expression returns `valueA`, otherwise it returns `valueB`. In the shown example, object detection is used to test if a value was supplied for the `hour` parameter. If it was not, the current hour is used instead.

Class Diagrams

JavaScript classes, just like C# or VB.NET classes, can be described visually using class diagrams. There are standards such as UML (**Unified Modeling Language**), that can be used to model classes and the relationships between them. In this book we'll show quite a few class diagrams using the notation used by Visual Studio 2005. Using this notation, the `HelloWorld` class shown earlier would be described as shown in Figure 3-2.

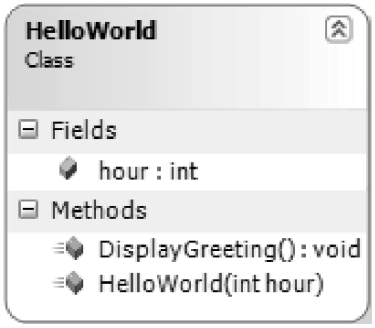

Figure 3-2. HelloWorld class diagram

The diagrams to this book follow typical conventions for C# classes, which don't translate to JavaScript exactly. For example, the diagram in Figure 3-2 says that the `HelloWorld` class has an integer field named `hour`. However, JavaScript doesn't support specifying data types for variables or class fields. The data type of the field makes the diagram helpful in specifying the intended purpose and type of the field, but that type isn't used in the actual implementation of the class.

The diagram also mentions the `HelloWorld()` constructor, which receives an integer parameter. As you know, JavaScript doesn't support "real" constructors. However, by reading the diagram you can tell that the `HelloWorld()` function receives a parameter named `hour`, which is supposed to be an integer value.

Appendix A contains more details about the conventions used in class diagrams throughout this book.

C# and JavaScript Classes

For the purpose of demonstrating a few more OOP-related concepts, we'll use another class. Our new class is named `Table`, and it has two public fields (`rows`, `columns`), and one method, `getCellCount()`. The `getCellCount()` method should return the number of rows multiplied by the number of columns. The class constructor should receive two parameters, used to initialize the `rows` and `columns` fields. This class could be represented by the class diagram in Figure 3-3.

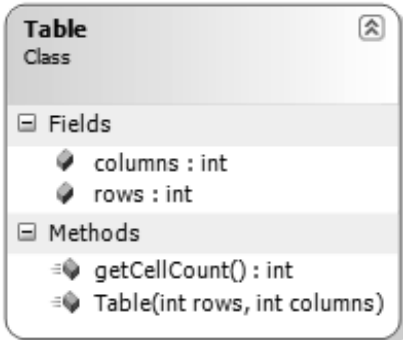

Figure 3-3. Class diagram representing the Table class

The C# version of this class would look like this:

```
public class Table
{
  // public members
  public int rows = 0;
  public int columns = 0;

  // constructor
  public Table(int rows, int columns)
  {
    this.rows = rows;
    this.columns = columns;
  }

  // method returns the number of cells
  public int getCellCount()
  {
    return rows * columns;
  }
}
```

You'd instantiate and use the class like this:

```
Table t = new Table(3,5);
int cellCount = t.getCellCount();
```

In a production-quality C# implementation you may want to implement `rows` and `columns` as properties with `get` and `set` assessors, rather than public fields. That implementation, however, would make its JavaScript version more complicated than necessary for the purposes of our examples.

The `Table` class can be easily implemented in JavaScript as shown in the following code snippet, and it would resemble very much its C# version:

```
function Table (rows, columns)
{
  // "constructor"
  this.rows = rows;
  this.columns = columns;

  // getCellCount "method"
  this.getCellCount = function()
  {
    return this.rows * this.columns;
  };
}
```

After having declared the object, we can instantiate it by using the `new` operator and use its properties and methods:

```
var t = new Table(3,5);
var cellCount = t.getCellCount();
```

There are a few subtle points you need to notice regarding the JavaScript implementation of `Table`:

- You don't declare public members explicitly. You simply need to reference them using `this`, and assign some value to them; from that point on, they're both declared and defined.
- JavaScript allows you to implement most of the design specifications defined in class diagrams, but the implementation can't reflect the specification as accurately as a C# implementation can. For example, the line **Table (int rows, int columns)** in the diagram in Figure 3-3 refers to the constructor of the class. In JavaScript, as you know, classes as implemented using functions neither have real constructors, nor support specifying data types for their parameters.
- When objects are created, each object has its own set of data—to maintain its own state. However, C# and JavaScript are different in that in JavaScript functions are first-class objects. In C#, the "state" is made of the object's fields. The object functionality, as defined by its methods, is the same for all objects of the same type. For example, if you create many objects of the type `Table` in C#, each object will have its own set of `rows` and `columns`, but internally they all use the same copy of the `getCellCount()` method. In JavaScript, however, functions are treated like any other variable. In other words, creating a new `Table` object in JavaScript will result not only in creating a new set of `rows` and `columns` values, but also in a new copy of the `getCellCount()` method. Usually, you don't need (or want) this behavior.

The last mentioned problem is commonly referred to as a "memory leak", although technically it's just inefficient JavaScript object design. When we design our JavaScript "classes" as we do in typical OOP languages, we don't need each class to create its own set of methods. It's only state (fields) that need to be individual, and not methods' code. The good news is that JavaScript has a neat trick that we can use to avoid replicating the inner function code for each object we create: referencing external functions.

Referencing External Functions

Instead of defining member functions ("methods") inside the main function ("class") as shown earlier, you can make references to functions defined outside your main function, like this:

```
function Table (rows, columns)
{
  // "constructor"
  this.rows = rows;
  this.columns = columns;

  // getCellCount "method"
  this.getCellCount = getCellCount;
}

// returns the number of rows multiplied by the number of columns
function getCellCount()
{
  return this.rows * this.columns;
}
```

Now, all your `Table` objects will share the same instance of `getCellCount()`, which is what you will usually want.

Thinking of Objects as Associative Arrays

A key element in understanding JavaScript objects is understanding the notion of **associative arrays**, which are nothing more than collections of (**key**, **value**) pairs. As a .NET developer you have worked with associative arrays represented by classes such as `NameValueCollection`, `Hashtable`, dictionaries, and others. Unlike with normal arrays, where the key is numeric (as in `bookNames[5]`), the key of an associative array is usually a string, or even other kinds of objects that can represent themselves as strings. For example, take a look at the following code snippet, where we retrieve the name of the book by specifying a unique string value that identifies that book:

```
// retrieve the name of the book
bookName = bookNames["ASP_AJAX"];
```

The concept is simple indeed. In this case, the key and the value of the `bookNames` associative array are both strings. This associative array could then be represented by a table like this:

Key	Value
ASP_AJAX	Microsoft AJAX Library Essentials
AJAX_PHP	AJAX and PHP: Building Responsive Web Applications
SEO_ASP	Professional Search Engine Optimization with ASP.NET

The table above can be represented in JavaScript, as an associative array, like this:

```
// define a simple associative array
var bookNames =
{ "ASP_AJAX" : "Microsoft AJAX Library Essentials",
  "AJAX_PHP" : "AJAX and PHP: Building Responsive Web Applications",
  "SEO_ASP" : "Professional Search Engine Optimization with ASP.NET"
};
```

The key of an element doesn't have to be literal; it can even be specified through a variable:

```
// store the book ID in a variable
var bookId = "ASP_AJAX";

// display the name of the book
document.write("The name of " + bookId +
               " is " + bookNames[bookId] + "<br />");
```

In JavaScript, however, the implementation of the associative array is more powerful, in that it makes no restriction on the type of the value of the (key, value) pair. The value can be a number, a string, a date, or even a function! This flexibility allows us to represent JavaScript objects as associative arrays. For example, an instance of the `Table` class that we discussed earlier can be represented like this:

```
// create Table object
var t =
  { rows : 3,
    columns : 5,
    getCellCount : function () { return this.rows * this.columns; }
  };

// display object field values
document.writeln("Your table has " + t.rows + " rows" +
                 " and " + t.columns + " columns<br />");

// call object function
document.writeln("The table has " + t.getCellCount() +
                 " cells<br />");
```

This example, and the one presented earlier with book names, can be tested online at `http://www.cristiandarie.ro/asp-ajax/Associative.html`, and the result is presented in Figure 3-4.

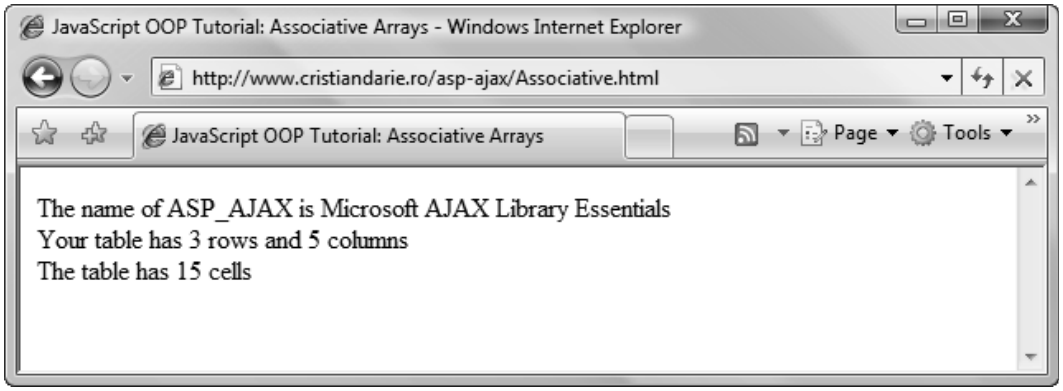

Figure 3-4. Testing JavaScript associative arrays

The literal notation of creating JavaScript objects has one weakness—it can only be used to describe objects. In other words, using the literal notation you can only define (key, value) pairs, but you can't create classes, class constructors, or other reusable components of code.

Creating Object Members on the Fly

One major difference between OOP in C# and ASP.NET, and OOP in JavaScript, is that JavaScript allows creating object members "on the fly". This is true for objects and classes that you create yourself and also for JavaScript's own objects and types as well. Here's an example where we add a field named `ImADate` to a JavaScript `Date` object:

```
// create a Date object
var myDate = new Date();

// create a new member named ImADate in the oDate object
myDate.ImADate = "I'm a Date!";

// display the value of oDate.ImADate
document.write(myDate.ImADate);
A typical OOP language such as C#, VB.NET, or Java, doesn't allow you
to create members on the fly, like JavaScript does. Instead, each
member must be defined formally in the definition of the class.
```

Private Members

JavaScript doesn't support the notion of private members as C# does, but you can simulate the functionality by using variables inside the function. Variables are declared with the `var` keyword or are received as function parameters.
They aren't accessed using `this`, and they aren't accessible through function instances, thus acting like private members. Variables can, however, be accessed by closure functions.

If you want to test this, modify the `Table` function as shown below.

```
function Table (rows, columns)
{
  // save parameter values to local variables
  var _rows = rows;
  var _columns = columns;

  // return the number of table cells
  this.getCellCount = function()
  {
    return _rows * _columns;
  };
}
```

This time we persist the values received as parameters as local variables named `_rows` and `_columns`. Note they aren't referred to using `this` any more. Local variables names don't need to start with an underscore, but this is a useful naming convention that specifies they are meant to be used as private members. You can make a short test that the "private" members can't be accessed from outside the function, and that `getCellCount()` still works, using code such as the following. The results are shown in Figure 3-5.

```
// create a Table object
var t = new Table(3,5);

// display object field values
document.write("Your table has " + t._rows + " rows" +
               " and " + t._columns + " columns<br />");

// call object function
document.write("The table has " + t.getCellCount() + " cells<br />");
```

Figure 3-5. JavaScript example demonstrating "private" members

This exercise reveals that `_rows` and `_columns` aren't accessible from outside the function's scope. Their values read **undefined** because there are no fields named `_rows` and `_columns` in the `Table` function. The `getCellCount()` function, on the other hand, can read `_rows` and `_columns` as variables because they are in the same closure. As you can see, although the implementation and behavior are somewhat different than in C#, you still have a way of defining internal (private) members inside a function.

Prototypes

You learned earlier that in JavaScript you should define "class methods" outside the body of the "class", in order to prevent their multiplication for each instantiated object. Prototyping is a JavaScript language feature that allows attaching functions and properties to the "blueprint" of a function. When functions are added to a class (function) prototype, they are not replicated for each object of the class (function). This reflects quite well the behavior of classes in C#, although the core mechanism and the specific implementation details differ greatly. A few facts that you should keep in mind about prototypes are:

- Every JavaScript function has a property named `prototype`. Adding members to the function's prototype is implemented by adding them to the `prototype` property of the function.
- Private variables of a function aren't accessible through functions added to its prototype.
- You can add members to a function's prototype at any time, but this won't affect objects that were already created. It will affect only any new ones.
- You can add members to a function's prototype only after the function itself has been defined.

The `Table` "class" from the previous example contains a "method" named `getCellCount()`. The following code creates the same class, but this time adding `getCellCount()` to its prototype:

```
// Table class
function Table (rows, columns)
{
  // save parameter values to class properties
  this.rows = rows;
  this.columns = columns;
}

// Table.getCellCount returns the number of table cells
Table.prototype.getCellCount = function()
{
  return this.rows * this.columns;
};
```

The JavaScript Execution Context

In this section we'll take a peek under the hood of the JavaScript closures and the mechanisms that allow us to create classes, objects, and object members in JavaScript. For most cases, understanding these mechanisms isn't absolutely necessary for writing JavaScript code—so you can skip it if it sounds too advanced. If, on the contrary, you should be interested in learning more about the JavaScript parser's inner workings, see the more advanced article at `http://www.jibbering.com/faq/faq_notes/closures.html`.

The JavaScript **execution context** is a concept that explains much of the behavior of JavaScript functions, and of the code samples presented earlier. The execution context represents the environment in which a piece of JavaScript code executes. JavaScript knows of three execution contexts:

- The **global execution context** is the implicit environment (context) in which the JavaScript code that is not part of any function executes.
- The **function execution context** is the context in which the code of a function executes. A function context is created automatically when a function is executed, and removed from the contexts stack afterwards.
- The **eval() execution context** is the context in which JavaScript code executed using the `eval()` function runs.

Each execution context has an associated *scope*, which specifies the objects that are accessible to the code executing within that context.

The scope of the global execution context contains the locally defined variables and functions, and the browser's `window` object. In that context, `this` is equivalent to `window`, so you can access, for example, the `location` property of that object using either `this.location` or `window.location`.

The scope of a function execution context contains the function's parameters, the locally defined variables and functions, and the variables and functions in the scope of the calling code. This explains why the `getCellCount()` function has access to the `_rows` and `_columns` variables that are defined in the outer function (`Table`):

```
// Table class
function Table (rows, columns)
{
  // save parameter values to local variables
  var _rows = rows;
  var _columns = columns;

  // return the number of table cells
  this.getCellCount = function()
  {
    return _rows * _columns;
  };
}
```

The scope of the `eval()` execution context is identical to the scope of the calling code context. The `getCellCount()` function from the above code snippet could be written like this, without losing its functionality:

```
  // return the number of table cells
  this.getCellCount = function ()
  {
    return eval(_rows * _columns);
  };
```

var x, this.x, and x

An execution context contains a collection of (key, value) associations representing the local variables and functions, a prototype whose members can be accessed through the `this` keyword, a collection of function parameters (if the context was created for a function call), and information about the context of the calling code.

Members accessed through `this`, and those declared using `var`, are stored in separate places, except in the case of the global execution context where variables and properties are the same thing. In objects, variables declared through `var` are not accessible through function instances, which makes them perfect for implementing `private` "class" members, as you could see in an earlier exercise. On the other hand, members accessed through `this` are accessible through function instances, so we can use them to implement public members.

When a member is read using its literal name, its value is first searched for in the list of local variables. If it's not found there, it'll be searched for in the prototype. To understand the implications, see the following function, which defines a local variable `x`, and a property named `x`. If you execute the function, you'll see that the value of `x` is read from the local variable, even though you also have a property with the same name:

```
function BigTest()
{
  var x = 1;
  this.x = 2;

  document.write(x); // displays "1"
  document.write(this.x); // displays "2"
}
```

Calling this function, either directly or by creating an instance of it, will display 1 and 2—demonstrating that variables and properties are stored separately. Should you execute the same code in the global context (without a function), for which variables and properties are the same, you'd get the same value displayed twice.

When reading a member using its name literally (without `this`), if there's no local variable with that name, the value from the prototype (property) will be read instead, as this example demonstrates:

```
function BigTest()
{
  this.x = 2;
  document.write(x); // displays "2"
}
```

Using the Right Context

When working with JavaScript functions and objects, you need to make sure the code executes in the context it was intended for, otherwise you may get unpredictable results. You saw earlier that the same code can have different output when executing inside a function or in the global context.

Things get a little more complicated when using the `this` keyword. As you know, each function call creates a new context in which the code executes. When the context is created, the value of `this` is also decided:

- When an object is created from a function, `this` refers to that object.
- In the case of a simple function call, no matter if the function is defined directly in the global context or in another function or object, `this` refers to the global context.

The second point is particularly important. Using `this` in a function that is meant to be called directly, rather than instantiated as an object, is a bad programming practice, because you end up altering the global object. Take this example that demonstrates how you can overwrite a global variable from within a function:

```
x = 0;
function BigTest()
{
  this.x = 1; // modify a variable of the global context
}
BigTest();
document.write(x); // displays "1"
```

Modifying the global object can be used to implement various coding architectures or features, but abusing of this technique can be dangerous. On the other hand, if `BigTest` is instantiated using the `new` keyword, the `this` keyword will refer to the new object, rather than the global object. Modifying the previous example as highlighted below, we can see the `x` variable of the global context remains untouched:

```
x = 0;
function BigTest()
{
  this.x = 1; // create an internal object property
}
var obj = new BigTest();
document.write(x); // displays "0"
```

When creating your own code framework, you can enforce that a function's code is executed through a function instance. The little trick involves creating a new object on the spot if the function was called directly, and using that object for further processing. This allows you to ensure that a function call will not modify any members of the global context. It works like this:

```
x = 0;
function BigTest()
{
  if (!(this instanceof BigTest)) return new BigTest();
  this.x = 1;
}
BigTest();
document.write(x); // displays "0"
```

The highlighted line simply checks if the `this` keyword refers to an instance of `BigTest` (the `instanceof` keyword is used for this). If it's not, a new `BigTest` instance is returned, and execution stops. The `BigTest` instance, however, is executed, and this time `this` will be a `BigTest` instance, so the function will continue executing in the context of that object.

This ends our little incursion into JavaScript's internals. The complete theory is more complicated than that, and it's comprehensively covered by David Flangan's *JavaScript: The Definitive Guide, Fifth Edition* (O'Reilly, 2006). The FAQ at `http://www.jibbering.com/faq/` will also be helpful if you need to learn about the more subtle aspects of JavaScript.

Inheritance using Closures and Prototypes

There are two significant techniques for implementing the OOP concept of inheritance with JavaScript code. The first technique uses closures, and the other technique makes use of a feature of the language named prototyping.

Early implementations of the Microsoft AJAX library made use of closures-based inheritance, and in the final stage the code was rewritten to use prototypes. In the following few pages we'll quickly discuss both techniques.

Inheritance Using Closures

In classic OOP languages such as C#, C++, or Java, you can extend classes through inheritance. Closure-based inheritance is implemented by creating a member in the derived class that references the base class, and calling that member. This causes the derived class to inherit all the base class members, effectively implementing the concept of inheritance.

To demonstrate this technique, we'll implement two classes: `Car` and `SuperCar`. The `Car` class constructor receives a car name as parameter, and it has a method named `Drive()`. The class `SuperCar` inherits the functionality of `Car`, and adds a new method named `Fly()`, reflecting the additional functionality it has in addition to what `Car` has to offer. The diagram in Figure 3-6 describes these two classes.

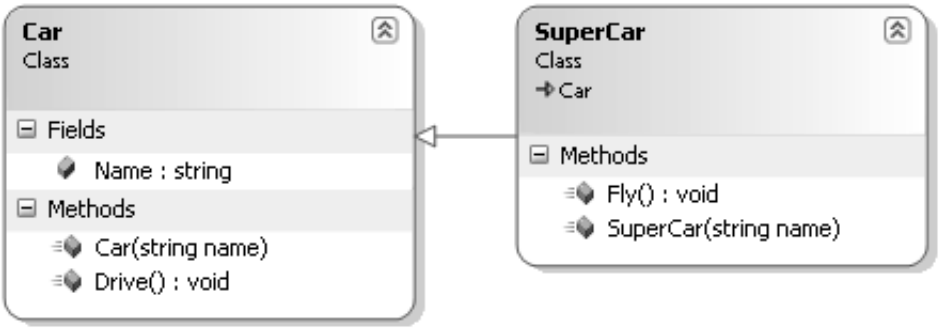

Figure 3-6. Car and SuperCar class diagram

Rember that in JavaScript the implementation of a class diagram can be achieved in multiple ways. The code reflects the concept of the diagram, but not also the implementation details, as the C# code would. Here's a possible implementation of `Car` and `SuperCar`:

```
<script type="text/javascript">
  // to be used as the Drive method of Car
  function Drive()
  {
    document.write("My name is " + this.Name +
                   " and I'm driving. <br />");
  }

  // class Car
  function Car(name)
  {
    // create the Name property
    this.Name = name;

    // Car knows how to drive
    this.Drive = Drive;
  }

  // to be used as the Fly method of SuperCar
  this.Fly = function()
  {
    document.write("My name is " + this.Name + " and I'm flying! <br
/>");
  }

  // class SuperCar
  function SuperCar(name)
  {
    // implement closure inheritance
    this.inheritsFrom = Car;
    this.inheritsFrom(name);

    // SuperCar knows how to fly
    this.Fly = Fly;
  }

  // create a new Car and then Drive
  var myCar = new Car("Car");
  myCar.Drive();

  // create SuperCar object
  var mySuperCar = new SuperCar("SuperCar");
```

```
    // SuperCar knows how to drive
    mySuperCar.Drive();

    // SuperCar knows how to fly
    mySuperCar.Fly();
</script>
```

Loading this script in a browser would generate the results shown in Figure 3-7. It can be tested online at `http://www.cristiandarie.ro/asp-ajax/JavaScriptClosureInheritance.html`.

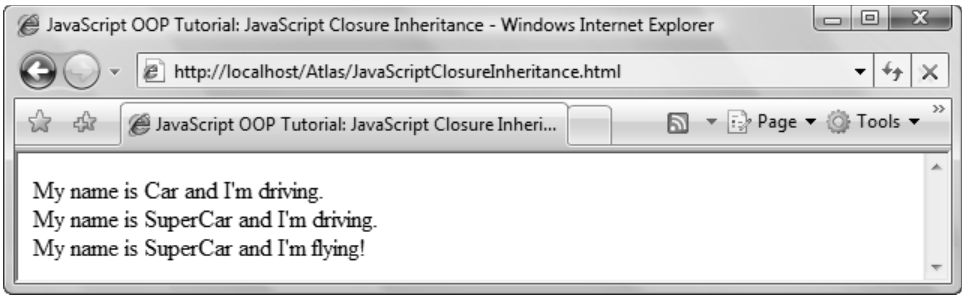

Figure 3-7. JavaScript Inheritance

The exercise demonstrates that inheritance really works. `SuperCar` only defines the capability to `Fly()`, yet it can `Drive()` as well. The capability to `Drive()` and the `Name` property are inherited from `Car`.

At the first sight the code can look a bit complicated, especially if you're a C# veteran. The `Drive()` and `Fly()` functions aren't defined inside `Car` and `SuperCar`, as you'd do in a C# class. Instead, we stored these methods/functions in the global context, and referenced them in `Car` and `SuperCar`, to avoid the memory leaks that were discussed earlier in this chapter. You can, however, define `Drive()` inside `Car`, and `Fly()` inside `SuperCar`, without losing any functionality.

If you comment the execution of `this.inheritsFrom(name)` from `SuperCar`, it won't inherit the capabilities of `Car` any more. If you make this test in FireFox, you'll see the following eloquent error message in the **Error Console** window of Firefox:

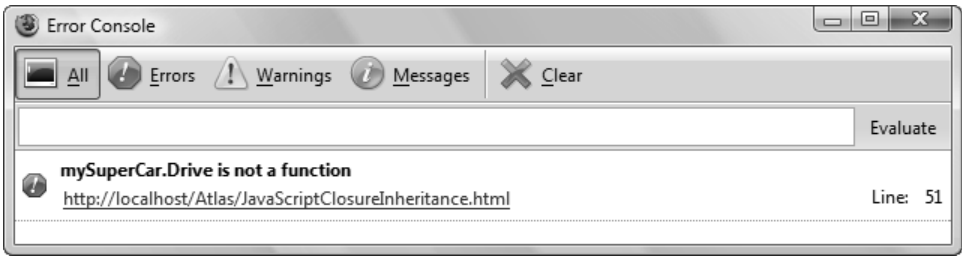

Figure 3-8. Signs of failed inheritance

The problem with the presented inheritance solution is that it's not very elegant. Writing all functions and classes in the global context can quickly degenerate into chaos; and things get even more complicated if you want to have classes that have functions with the same name. Needless to say, this isn't something you need to be dealing with when writing your code. Luckily, JavaScript has a very neat feature that allows us implement inheritance in a much cleaner way: **prototyping**.

Inheritance Using Prototyping

Once again, prototyping can help us implement an OOP feature in a more elegant way than when using closures. Prototype-based inheritance makes use of the behavior of JavaScript prototypes. When accessing a member of a function, that member will be looked for in the function itself. If it's not found there, the member is looked for in the function's prototype. If it's still not found, the member is looked for in the prototype's prototype, and so on until the prototype of the implicit `Object` object.

In closure-based inheritance, the derived class inherits the base class methods and properties by "loading" them into itself. Here's the code again for your reference:

```
// class SuperCar
function SuperCar(name)
{
  // implement closure inheritance
  this.inheritsFrom = Car;
  this.inheritsFrom(name);

  // SuperCar knows how to fly
  this.Fly = Fly;
}
```

When implementing inheritance through prototyping, we can "load" the base class properties and methods by adding them to the derived class prototype. That way, an object of the derived class will have access to the class methods and properties, but also to the base class methods and properties since they exist in the derived class prototype. To successfully implement prototype-based inheritance with JavaScript, you need to:

- Add a base class instance to the derived class `prototype` property, as in `SuperCar.prototype = new Car()`. This creates `Car` as `SuperCar`'s prototype.
- The `prototype` property has a `constructor` property that needs to point back to the function itself. Since now the `SuperCar`'s prototype is a `Car`, its `constructor` property points back to the constructor of `Car`. To fix this, we need to set the `constructor` property of the `prototype` property of the derived class to the class itself, as in `SuperCar.prototype.constructor = SuperCar`.

- Create the derived class constructor, and call the base class constructor from there, eventually passing any necessary parameters. In other words, when a new `SuperCar` is instantiated, its base class constructor should also execute, to ensure correct base class functionality.
- Add any additional derived class members or functions to its prototype.

This is so very complicated! In practice you'll find that the code doesn't look that scary, although the complete theory is a little more complex than this. A nice article describing a few additional theoretical aspects can be found at `http://mckoss.com/jscript/object.htm`.

The new implementation of `Car` and `SuperCar`, this time using prototypes, is the following, with the inheritance mechanism highlighted. The `Drive()` and `Fly()` methods have also been created through prototyping, although the old version using closures would work as well. The code can be checked online at `http://www.cristiandarie.ro/seo-asp/JavaScriptPrototypeInheritance.html`.

```
<script type="text/javascript">
  // class Car
  function Car(name)
  {
    // create the Name property
    this.Name = name;
  }

  // Car.Drive() method
  Car.prototype.Drive = function()
  {
    document.write("My name is " + this.Name +
                   " and I'm driving. <br />");
  }

  // SuperCar inherits from Car
  SuperCar.prototype = new Car();
  SuperCar.prototype.constructor = SuperCar;

  // class SuperCar
  function SuperCar(name)
  {
    // call base class constructor
    Car.call(this, name);
  }

  // SuperCar.Fly() method
  SuperCar.prototype.Fly = function()
  {
```

```
        document.write("My name is " + this.Name +
                       " and I'm flying! <br />");
    }

    // create a new Car and then Drive
    var myCar = new Car("Car");
    myCar.Drive();

    // create SuperCar object
    var mySuperCar = new SuperCar("SuperCar");

    // SuperCar knows how to drive
    mySuperCar.Drive();

    // SuperCar knows how to fly
    mySuperCar.Fly();
</script>
```

Here, instead of creating a `Car` instance in `SuperCar`'s constructor, we declare `Car` as `SuperCar`'s prototype.

Introducing JSON

In AJAX applications, client-server communication is usually packed in XML documents, or in the **JSON** (JavaScript Object Notation) format. Interestingly enough, JSON's popularity increased together with the AJAX phenomenon, although the AJAX acronym includes XML. JSON is the format used by the Microsoft AJAX Library and the ASP.NET AJAX Framework to exchange data between the AJAX client and the server, which is why it deserves a quick look here. As you'll learn, the Microsoft AJAX Library handles JSON data packaging through `Sys.Serialization.JavaScriptSerializer`, which is described in the Appendix—but more on this later.

Perhaps the best short description of JSON is the one proposed by its official website, `http://www.json.org`: "JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate."

If you're new to JSON, a fair question you could ask would be: why another data exchange format? JSON, just like XML, is a text-based format that it is easy to write and to understand for both humans and computers. The key word in the definition above is "lightweight". JSON data structures occupy less bandwidth than their XML versions.

To get an idea of how JSON compares to XML, let's take the same data structure and see how we would represent it using both standards:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<response>
  <clear>false</clear>
  <messages>
    <message>
      <id>1</id>
      <color>#000000</color>
      <time>2006-01-17 09:07:31</time>
      <name>Guest550</name>
      <text>Hello there! What's up?</text>
    </message>
    <message>
      <id>2</id>
      <color>#000000</color>
      <time>2006-01-17 09:21:34</time>
      <name>Guest499</name>
      <text>This is a test message</text>
    </message>
  </messages>
</response>
```

The same message, written in JSON this time, looks like this:

```
[
  {"clear":"false"},
  "messages":
    [
      {"message":
        {"id":"1",
         "color":"#000000",
         "time":"2006-01-17 09:07:31",
         "name":"Guest550",
         "text":"Hello there! What's up?"}
      },
      {"message":
        {"id":"2",
         "color":"#000000",
         "time":"2006-01-17 09:21:34",
         "name":"Guest499",
         "text":"This is a test message"}
```

```
            }
        ]
    }
]
```

As you can see, they aren't *very* different. If we disregard the extra formatting spaces that we added for better readability, the XML message occupies 396 bytes while the JSON message has only 274 bytes.

JSON is said to be a *subset* of JavaScript because it's based on the associative array-nature of JavaScript objects. JSON is based on two basic structures:

- **Object**: This is defined as a collection of name/value pairs. Each object begins with a left curly brace ({) and ends with a right curly brace (}). The pairs of names/values are separated by a comma. A pair of name/value has the following form: *string:value*.
- **Array**: This is defined as a list of values separated by a coma (,).

We've mentioned `strings` and `values`. A `value` can be a `string`, a `number`, an `object`, an `array`, `true` or `false`, or `null`. A `string` is a collection of Unicode characters surrounded by double quotes. For escaping, we use the backslash (\).

It's obvious that if you plan to use JSON, you need to be able to parse and generate JSON structures in both JavaScript and ASP.NET, at least if the communication is bidirectional. JSON libraries are available for most of today's programming languages: ActionScript, C, C++, C#, VB.NET, Delphi, E, Erlang, Java, JavaScript, Lisp,Lua, ML and Ruby, Objective CAML, OpenLazslo, Perl, PHP, Python, Rebol, Ruby, and Squeak. When we said almost every programming language we were right, weren't we!

If you plan to work with JSON data outside of the Microsoft AJAX Library, you can use the library listed at `http://www.json.org/js.html`.

Summary

This chapter walked you through many fields. Working with OOP in JavaScript is certainly no easy task, especially if you haven't been exposed to the implied concepts before. Where you don't feel confident enough, have a look at the additional resources we've referenced. When you feel ready, proceed to Chapter 4, where you will have an overview of the architecture and features of the Microsoft AJAX Library.