Übersicht zu JScript.NET - die serverseitige Variante von JScript

JScript ist die Microsoft-Variante von JavaScript.

Im engeren Sinne ist JScript nur die Programmiersprache.

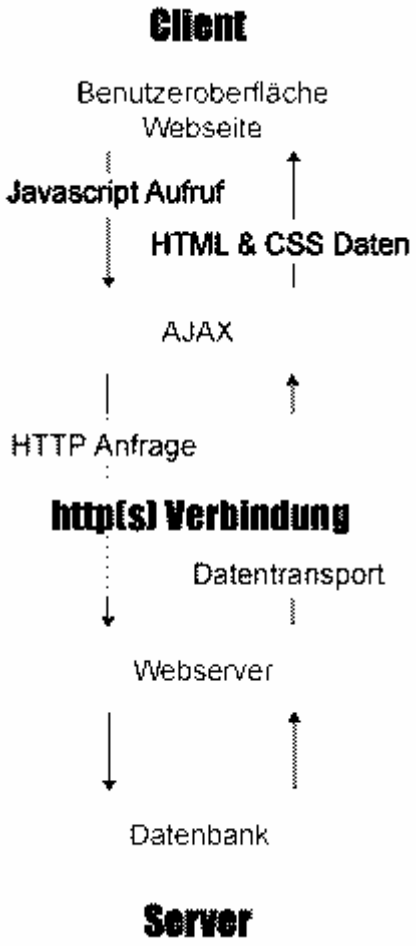
Im weiteren Sinne wird die Verbindung von Programmiersprache mit HTML-DOM etc. als JScript verstanden: Microsoft-Variante des Webdesign per MS HTML und MS JavaScript (JScript) und MS XML etc.. Ein weite verbreiteter Bezeichner für die Summe dieser Features ist DHTML - hier also MS DHTML (Dynamisches HTML per Script und Style (wobei Style per Script und/oder in HTML implementiert werden kann)).

Im Rahmen der Einführung von Net-Framework (ASP.Net etc.) wurde auch JScript (wie VBScript) serverfähig gemacht. Die Ähnlichkeit mit dem clientseitigen JScript ist beabsichtig - JScript.NET ist der Nachfolger vom JScript.

Net-Framework nutzt analog zu Sun-Java kompilierten Byte-Code, dessen Ausführung durch die jeweilige Laufzeitumgebung erfolgt: Der Bytecode ist dadurch portierbar. Für JScript.Net ist also ein Compiler aus dem Net-Framework nötig, um Byte-Code zu erzeugen.

JScript benötigt weder Server noch Bytecode, dafür nur die JScript-Maschine zum Browser.

Das Ende vom clientseitigen JScript wäre eingeläutet, wenn es nicht Fremd-Produkte wie Ajax gäbe, die intensiv JavaScript nutzen. Ajax hat aber einen Haken: Es benötigt JavaScript, welches seit eh und je browserspezifisch implementiert wurde (HTML-DOM, XML-DOM etc.).



Microsoft ignoriert Ajax nicht, denn dieses arbeitet client- **und** serverseitig, liegt also im Bereich des Server-Geschäftsfeldes.

http://www.asp.net/ Microsoft Ajax-Webseite im ASP-Net-Framework
'ASP.NET AJAX is a free framework'
...

'Featured ASP.NET AJAX Web Hosting
ASP.NET 2.0 Web Hosting

ASP.NET AJAX Compatible Hosting with MS SQL 2005, Real-Time SQL Backups, SQL Express to SQL 2005 Transfer Tool, SQL Restore Tool, Web Services Supported, Free ASP.NET Components and more from DiscountASP.NET, voted 2006 & 2005 Best ASP.NET Hosting Service by asp.netPRO Magazine. Get your ASP.NET AJAX powered web site/application online with the leader in ASP.NET Web Hosting.'

Wer also nur Microsoft-Produkte nutzt, muss sich um Browser-Anpassungen in Ajax keinen Kopf machen. (Die Divergenz der Browser als Geschäftsfeld).

Wie man also sieht, JavaScript lebt so indirekt weiter z.B. als JScript.Net für Ajax unter ASP.Net

Allerdings bietet auch JScript (das clientseitige) neuerdings ohne Active-X-Control den Zugriff auf Daten an:
Wesentlich später als die Konkurrenz (z.B. Ajax-Entwickler für Gecko-Mozilla-Opera) hat Microsoft ab Version 7 des Internet Explorers das Objekt XMLHttpRequest für synchronen und ansynchronen Datenaustausch implementiert. Per XML-DOM und HTML-DOM und JScript können XML-Daten verwaltet werden. JScript selbst hat im HTML-DOM nur das magere XML-Objekt.

Clientseitiges JScript wegen XMLHttpRequest als Konkurrenz zum JScript-Net und Ajax unter ASP.Net ?

Im nachfolgenden Artikel von Microsoft zu JScript.Net, wird diese als JScript bezeichnet, also das clientseitige JScript ignoriert. Dazu kommt, dass in den neueren Net-Framework-Versionen JScript.Net als Feature gar nicht mehr oder kaum erwähnt wird. Microsoft nutzt dabei den Umstand, dass Net-Framework-Versionen nicht kompatibel sind, sich dafür aber parallel installieren lassen (analog zur Java-Runtime-Versionen von Sun).

Net-Framework ist kostenlos (inklusive Kommando-Zeilen-Tools wie Compiler). Microsoft stellt von den hauseigenen Entwicklungstools, die eine graphische Oberfläche haben, kostenlose Express-Versionen bereit. Wer mehr möchte, muss reichlich Euros berappen. - Sun hat begonnen, seine Java-Tools kostengünstig frei zu geben (warum auch immer).

Das Ende des clientseitigen JScript ist allerdings trotz allem zu erwarten: Genau dann, wenn es für den Hersteller keinen Gewinn mehr abwirft, oder die Pflege sich nicht mehr rechnet, oder der Aufwand zu groß ist, oder die Open Source-Gemeinde einfach mal besser ist, oder Microsoft sich mal wieder irrt (analog zur Aufmöblierung des Internet Explorers zu einem Zeitpunkt, an dem User die Konkurrenzprodukte bereits lieben gelernt haben - abgesehen von den permanenten Sicherheitsrisiken von Windows).

http://msdn2.microsoft.com/en-us/library/ms974588.aspx

Introducing JScript .NET

Andrew Clinick
Microsoft Corporation

July 14, 2000

Contents

This week marks a major step forward for script with the first public showing of JScript .NET and Visual Basic .NET at the Professional Developers Conference (PDC) in Orlando, and the release of Windows Script version 5.5 (available for download http://msdn.microsoft.com/scripting/). I thought I'd take this opportunity to go over some of the key advances in JScript® and Visual Basic® Scripting Edition (VBScript), and how scripting in general will evolve to take advantage of the new .NET platform.

What About VBScript?

Whenever I talk about JScript .NET, I always hear questions about VBScript and where that fits into the new .NET scripting plans. Since VBScript's inception a little over four years ago, we've been getting requests to add Visual Basic functions to VBScript, and to allow people to use "real" Visual Basic where they would traditionally use VBScript. The VBScript language has made considerable improvements in versions 5.0 and 5.5, so when we sat down to look at what we could add to VBScript, it became apparent that we'd eventually have to add pretty much all of Visual Basic's features. To achieve this (and, hopefully, to keep VBScript users happy), we could either re-implement Visual Basic ourselves, or work with the Visual Basic team to make Visual Basic a script engine. We chose the latter, because it would guarantee that the languages would remain in synch, and that VBScript would gain a slew of new features, such as finally being able to call *any* object, and not just automation (**IDispatch**) objects.

One caveat to this merging of the two languages is that a small number of VBScript features—such as **Eval** and the **Execute** functions—are not available in the first release of Visual Basic .NET. Although it may seem like we've thrown the baby out with the bathwater here, and destroyed the notion of Visual Basic .NET as a

"dynamic" scripting language, that's not really the case. The first release of Visual Basic .NET is targeted at building Web Services and applications, using ASP.NET on the server—where hopefully you aren't using functions such as **Eval** or **ExecuteGlobal**. We intend to add these features back into the Visual Basic .NET language in the next release, in time for our integration with Microsoft Internet Explorer (where the dynamic features of the language are more useful). An added bonus is that adding **Eval** to Visual Basic .NET for Internet Explorer also means adding **Eval** to Visual Basic .NET for your other .NET applications, because it's the same language.

The new features in Visual Basic are thoroughly documented on MSDN. Rather than re-iterate what is already covered there, I'm going to focus this article on the new features in JScript. The key thing to remember is that, from now on, there will be just one Visual Basic language to learn—which, we hope, will make your life as a .NET developer even easier. We'd love to get your feedback on this, so please feel free to use the scripting newsgroups or to e-mail us at msscript@microsoft.com.

JScript .NET

This is probably the biggest leap in functionality for JScript since the 1996 introduction of JScript version 1.0 with Internet Explorer 3.0. JScript has traditionally been used to develop client-side scripts due to its ubiquitous, cross-platform support on the Internet, but we've been seeing a steady increase in the usage of JScript on the server—particularly in Active Server Pages (ASP). For example, your favorite Web site (MSDN) uses a large amount of server-side JScript, as do many other sites on the Internet.

Using JScript on the server has resulted in people asking for performance improvements—you can never have too much performance on the server. However, now is a good time to point out that both the traditional JScript and VBScript languages, and the new Visual Basic .NET and JScript .NET languages, have very similar performance characteristics: Neither is noticeably faster than the other in the general case.

As scripts get bigger, script authors need to be able to write more robust code. And as programs become more complex, script authors have become frustrated by JScript's limitation of only dealing with automation (**IDispatch**) objects.

JScript .NET was designed with these requirements in mind. The JScript team was keen to ensure that the new language features were added in an evolutionary manner, so that you can leverage your existing JScript skills in the .NET world. It was vital that JScript .NET feel like a new version of the existing language, rather than a completely new language.

Evolution

The key part of JScript's evolution is keeping the language recognizably JScript, so that it will run existing JScript code and that any enhancements will work within the existing language definitions. For example, one of the new features is the introduction of types to the language. Types in JScript .NET are an extension of the existing variable and function declaration mechanisms, and are entirely optional. One of the defining qualities of a script language is the ability to write code without having to worry about the types of variables—or having to worry about variables at all, for that matter. Making types optional allows developers to leverage their existing JScript skills and source code, while providing a smooth migration path for adding types to new and existing programs to reap the benefits of improved performance and robustness.

Working Closely with ECMA

JScript's association with the ECMAScript standard has helped its success considerably. The standard has allowed innovation in the language to be developed in conjunction with all the members of the ECMAScript Technical Committee—which means that both JScript and JavaScript have remained very compatible throughout, and any new features are discussed and designed together. This approach ensures that both

languages can benefit from the ideas of many companies, instead of being isolated developments within individual companies.

The development of JScript .NET has continued this partnership, so that all the new features have been designed in conjunction with other ECMA members. It's important to note that the language features in the JScript .NET PDC release are not final. We're working with other ECMA members to finalize the design as soon as possible. In fact, there's an ECMA meeting this week at the PDC where we'll try to sort out some of the remaining issues.

Performance

Enough of the touchy, feely features of JScript .NET. Let's get into some of the real features, and how they will make a difference to your development. Perhaps the most important feature area of JScript .NET is the performance improvements to the language. The most dramatic impact on performance in JScript .NET is that it is a true compiled language, which makes it possible to achieve performance comparable to that of C# and Visual Basic .NET. From a language perspective, the key mechanism for getting performance improvements in JScript is the addition of types to the language. Typing in JScript .NET has been introduced via both traditional (explicit) type declarations, and implicit type inferencing. Type inferencing is an exciting technology that analyzes your use of variables in a script and infers the type of the variable for you. This means that you can get considerable improvements in speed using existing scripts without having to type your variables. For example, consider the following JScript program:

Copy Code

```
function test() { for (var x = 0; x < 100; x++) { print(x); } }
```

When JScript .NET compiles this program, it analyzes the use of x and determines that x is only ever used to hold numeric values. As a result, x can safely be defined as a number. This provides a performance improvement, since the JScript compiler can optimize the use of x as a number rather than as a generic **Object** (or variant) that could potentially contain any type of value.

In order to exploit the type-inferencing ability of the JScript compiler, you need to follow a few simple rules. Luckily, these rules are also part of general good coding practices, so you may already be following these rules in your existing code. The three simple rules to follow are:

1. Always declare your local variables. This may sound like an obvious point, but it is important. JScript can infer the types of local (function) variables only, not global variables. If you implicitly declare a variable (use it without declaring it in a **var** statement) it becomes a global variable, and can't be optimized.
2. Only use a variable for one type of data. If you declare a variable and use it to store a number, don't re-use the same variable later to store a string or another type of data. If you do this, JScript has no choice but to make the variable a generic **Object** (variant).

Here are some examples of how to follow these simple rules:

Copy Code

```
// can't infer the type -- glob is a global var glob = 42; function myfunc() { // can't infer the type -- s is not declared so it is // created as a global s = "hello"; // type inferencing works here -- i is declared var i = 0; // can't infer the type -- q is assigned different data types var q = new Date(); q = 3.14159; }
```

Type inferencing is a great technology, but it has two drawbacks. First, it always errs on the side of caution. Second, while type inferencing provides performance improvements, it won't help you catch type mismatch errors or other programming errors. To overcome this, JScript .NET provides a way to explicitly declare a variable as being of a particular type. This is achieved by using the new type annotation syntax on the **var** statement and for function parameter lists and return types. Type annotations are achieved by adding a colon (:) to the variable, parameter, or function declaration, followed by the type name.

For example:

Copy Code

```
// Declare an integer variable. var myInt : int = 42; // Declare a function that returns a String function GetName() : String { // function code } // Declare a function that takes a double parameter // and returns a Boolean function CheckNumber(dVal : double) : Boolean { // function code }
```

I converted the weather conditions function from my Scripting Web Services article http://msdn2.microsoft.com/en-us/library/ms974563.aspx to demonstrate adding type annotations to a function. The function takes a single String parameter, **strCity**, and returns a String result. Providing the type annotations allow the JScript .NET compiler to both optimize the compiled version of the function, and to provide compile-time type checking whenever the function is called (this also works from other languages, such as C# and Visual Basic). The key here is that adding type annotations is optional, but the benefits are considerable—so I encourage you to use them as much as possible.

Copy Code

```
function getConditions(strCity : String) : String { var now : Date = new Date(); switch (strCity.toUpperCase()) { case "LONDON": if (now.getMonth() <= 7 || now.getMonth() >= 9) { return "overcast"; } else { return "partly overcast and humid"; } break; case "SEATTLE": if (now.getMonth() == 7 && now.getDay() == 4) { return "torrential rain"; } else { return "rain"; } break; case "LA": return "smoggy"; break; case "PHOENIX": return "damn hot"; break; default: return "partly cloudy with a chance of showers"; } }
```

In JScript .NET, you can also declare variables using any .NET Framework type. Additional types can be introduced into JScript either by importing a namespace that contains the new type, or by declaring user-defined types as classes. There is potential overlap between .NET Framework types and JScript built-in objects, so JScript .NET provides a mapping between the two for type annotations:

Boolean	.NET Framework Boolean / JScript boolean
Number	.NET Framework Double / JScript number
String	.NET Framework String / JScript string
Int	.NET Framework Int32
Long	.NET Framework Int64
Float	.NET Framework Single
Double	.NET Framework Double
Object	.NET Framework Object / JScript Object
Date	JScript Date object
Array	JScript Array
Function	JScript Function object

The other major performance improvement added to JScript .NET is the introduction of Option Fast. This option tells the compiler to enforce certain rules that allow additionally optimizations, at the cost of some reduced functionality. When Option Fast is enabled, the following JScript behavior is activated:

- You must declare all variables.
- You cannot assign values to, or redefine, functions.
- You cannot assign to, or delete, predefined properties of the built-in JScript objects.
- You cannot add **expando** properties to the built-in objects.
- You must supply the correct number of arguments to function calls.
- The **arguments** property is not available within function calls.

The new Option Fast feature helps you write faster code, but it does change the behavior of the language—so just adding Option Fast to an existing JScript program may result in one or more compiler errors. Nevertheless, with some small changes to your program, you should see some significant performance improvements.

Compilation

One of the key new capabilities of JScript .NET is the ability to compile to .NET IL (Intermediate Language), which means that, with some effort, JScript code will produce essentially the same compiled code as Visual Basic .NET, C#, or any other .NET language. Finally, script developers will be able to get those compiler nuts off their backs and get on with their work. (Scripters know that having to go through a compilation phase to get an EXE or a DLL is *so* 20th century, but we'll keep that under our hats for a while longer.)

Although compiling is a pain when you want to write a quick script, it does have its uses—and JScript .NET allows you to compile your script into an EXE or a DLL, so that you can send out precompiled code rather than having to compile source every time.

Productivity

Now that you've got your JScript code running much faster, hopefully you'll be compelled to write more JScript code, and, probably, larger programs. Writing larger programs in JScript today can be quite difficult, since it doesn't provide many mechanisms to encapsulate code—and the code you write isn't necessarily the most robust. JScript does have a prototype inheritance model that allows for encapsulation, but it's not very well known, and even less well understood. It's also a very "fragile" sort of encapsulation, making it difficult to write robust code. If you try to reference a property on the object that doesn't exist, JScript will simply add it for you, rather than telling you it's not there. This feature is commonly known as **expando** properties, and it makes picking up JScript very easy, since you can extend existing objects very easily. Expando functionality, however, is a dual-edged sword, since this flexibility ultimately makes it difficult to write robust, large-scale scripts.

To address this, JScript .NET introduces classes and packages to the language. Classes allow you to develop objects to encapsulate functionality and data very easily, with the added advantage of being able extend existing classes (single inheritance for your sanity, if nothing else) and implement interfaces. The best thing about this is that, because JScript is a fully-fledged .NET language, you can extend (or implement) any class (or interface) defined in any other .NET language—and vice-versa.

By default, classes in JScript don't support dynamic properties (expandos), thus allowing you to sidestep the issues they might cause. Nevertheless, in the spirit of evolving the language and allowing classes to be used with existing code (helpful if you're writing a class library), and because they're a cool feature, JScript classes can handle dynamic properties by marking the class as **expando**.

Declaring classes in JScript is achieved via the **class** statement, which contains methods and properties defined by using familiar **function** and **var** declarations. If you're familiar with the current JScript syntax for creating constructor functions, the migration to classes should be pretty simple for you. For most objects, you need only enclose the constructor function with a class of the same name, declare the class members, and move the function declarations inside the class. If you mark the enclosing class as **expando**, you don't even need to declare the class members, although your code won't be as robust. For example:

JScript 5.5 Code

Copy Code

```
// Simple object with no methods function Car(make, color, year) { this.make = make; this.color = color;
this.year = year; } function Car.prototype.GetDescription() { return this.year + " " + this.color + " " +
this.make; } // Create and use a new Car object var myCar = new Car("Accord", "Maroon", 1984);
print(myCar.GetDescription());
```

JScript .NET Code

Copy Code

```
// Wrap the function inside a class statement. class Car { // Declare the class members. I've used types in this example, // but they are not required. I could have also marked the class // as being 'expando' and not had to declare these members. var make : String; var color : String; var year : int; // Old constructor function, unchanged. function Car(make, color, year) { this.make = make; this.color = color; this.year = year; } // Move the function inside the class function GetDescription() { return this.year + " " + this.color + " " + this.make; } } // Create and use a new Car object var myCar = new Car("Accord", "Maroon", 1984); print(myCar.GetDescription());
```

By default, the **function** and **var** declarations within a class declare publicly visible functions and properties. JScript .NET also supports private and protected properties and functions; just add **private** or **protected** in front of the **function** or **var** declaration to get the desired visibility.

JScript .NET also supports the declaration of property accessors—custom functions that run when a property is read or written—by using the **get** or **set** modifiers. For example:

Copy Code

```
class Person { // Private variables -- actual data can't be seen // outside the class private var m_sName : String; private var m_iAge : int; // Constructor -- called to create new objects function Person(name : String, age : int) { this.m_sName = name; this.m_iAge = age; } // Name is read-only as there is no 'set' function function get Name() : String { return this.m_sName; } // Age is read-write, but can only be set to // "meaningful" values function get Age() : int { return this.m_sAge; } function set Age(newAge : int) { if ((newAge >= 0) && (newAge <= 110)) this.m_iAge = newAge; else throw newAge + " is not a realistic age!"; } } var fred : Person = new Person("Fred", 25); print(fred.Name); print(fred.Age); // This will cause a compiler error - Name is read-only fred.Name = "Paul"; // This will work fred.Age = 26; // This will give a run-time error, as the value is too large fred.Age = 200;
```

Inheritance

A JScript class can inherit and extend an existing class written in JScript or any other .NET framework language (e.g., C#, Visual Basic) by adding the **extends** keyword after the class statement. This ability allows JScript programs to take advantage of the richness of the .NET platform very easily. To illustrate this, I wrote a simple JScript program that creates a Windows 2000 service (a frequent request from script authors). The script consists of a class that extends the .NET framework's **ServiceBase** class.

Copy Code

```
/* Simple JScript service Andrew Clinick July 2000 */ // Import the required .NET namespaces. import System; import System.ServiceProcess; import System.Diagnostics; import System.Timers; class SimpleService extends ServiceBase { private var timer : Timer; // Constructor -- setup the service properties function SimpleService() { CanPauseAndContinue = true; ServiceName = "JScript Service"; timer = new Timer(); timer.Interval = 1000; timer.AddOnTimer(OnTimer); } // Method called when the service starts protected override function OnStart(args : String[]) { // Create an entry in the event log, and start the timer EventLog.WriteEntry("JScript Service started"); timer.Enabled = true; } // Method called when the service stops protected override function OnStop() { EventLog.WriteEntry("JScript Service stopped"); timer.Enabled = false; } // Method called when the service pauses protected override function OnPause() { EventLog.WriteEntry("JScript Service paused"); timer.Enabled = false; } // Method called when the service continues protected override function OnContinue() { EventLog.WriteEntry("JScript Service continued"); timer.Enabled = true; } // Method called every time the timer clicks function OnTimer(source : Object, e : EventArgs) { EventLog.WriteEntry("Hello World from JScript!"); } } // Create and run the service ServiceBase.Run(new SimpleService());
```

The **SimpleService** class extends the **ServiceBase** class; it also has some functions that override the various event handlers in ServiceBase. When the class is loaded, it automatically gets all the functionality required to be an NT service—and the only real code I had to write was the script for hooking up to the Timer and writing out to the event log every second. Notice how by including the **System.Diagnostics** namespace, I can just

call out to **EventLog.WriteEntry**; it does all the Windows API calls required to actually write the text to the event log.

I won't go into packages at length here, but they provide a mechanism to create a namespace into which classes can be added. This allows further flexibility in encapsulation, since you can put a set of like classes into a package—making it easier to package up code (pardon the pun).

Debugging

No matter how much better we make the JScript language, programmers will still make errors, and so great debugging support remains a key requirement for increasing developer productivity. We've enhanced the debugging capabilities in JScript .NET to allow full Visual Studio .NET debugging. Those of you who have struggled with debugging JScript 5 will welcome the new debugging features, which are now built on the same technology that is used by Visual Basic .NET and C#. Suffice to say that you can step through code, set break points, use immediate and watch windows, and use other great debugging features in JScript just as you can in the other Visual Studio .NET languages.

Examples Using JScript .NET

The primary focus for this release of JScript .NET is for scripting on the server—and, in particular, the new capabilities provided by ASP.NET. To illustrate using ASP.NET, I've written a few simple demos.

ASP.NET Page Accessing SQL Server

The first demo is a simple ASP.NET page using JScript .NET and the new data access classes in the .NET framework. I've used the familiar <% %> scripting mechanism to query the authors table in the SQL Server pubs sample database. I know it isn't very exciting, but it illustrates some of the new features in JScript:

Copy Code

```
<%@ Import Namespace="System.Data" %> <%@ Import Namespace="System.Data.SQL" %> <%@ language="JScript" %> <link rel="STYLESHEET" type="text/css" href="style.css"> <% // Setup the connections, commands and datasets var myConnection:SQLConnection = new SQLConnection("server=scripting;uid=sa;pwd=;database=pubs"); // Execute the SQL var myCommand:SQLDataSetCommand = new SQLDataSetCommand("select * from Authors", myConnection); // Set up my variables and type them using the new // typing feature in JScript var ds:DataSet = new DataSet(); var myTable:DataTable var myColumns:ColumnsCollection var myCol:DataColumn var myRows:RowsCollection var myRow:DataRow // get the data by calling the FillDataSet method myCommand.FillDataSet(ds, "Authors"); myTable = ds.Tables[0] %> <h1> <%=ds.Tables[0].TableName%> </h1> <br> <TABLE> <THEAD> <TR> <% //Iterate through the columns in the table and // write out the column names at the top of the table myColumns = myTable.Columns for (myCol in myColumns) { %> <TH class="spec"> <%=myCol.ColumnName%> </TH> <% } %> </TR> </THEAD> <% // Get all the rows and write out a TR for each row myRows = myTable.Rows //Notice how I can now iterate through collections //using for in rather than the enumerator object for (myRow in myRows) { %> <TR> <% for(var i:int=0;i<myColumns.Count;i++) { %> <TD class="spec"> <%=myRow[i]%> </TD> <% } %> </TR> <% } %> </TABLE>
```

A key point in this script is the ability to iterate through collections using for ... in. If you're a JScript developer today this is major step forward, because you no longer have to worry about what type of collection you're working with; JScript just does the right thing.

Another important note is that ASP.NET provides is the ability to bind controls to datasets. This means you wouldn't even have to write any of this script, but I thought I'd keep the process familiar for this example.

Creating a Web Service

In the past, the only way you could create a Web Service using JScript was via Remote Scripting. Remote Scripting is a great technology—but it's limited, because you can call the service only from a browser, and the format of its XML messages is proprietary. The onset of SOAP as a standard way to call Web Services makes

the proprietary format of the XML even less palatable. Fortunately, ASP.NET provides a great way to define Web Services.

All you need to do is create a JScript .NET class and put it into an ASMX file. ASP.NET does the rest. ASP.NET creates the Service Description Language (SDL) automatically, and handles any SOAP requests. I converted my Weather Web Service to be fully buzzword compliant in a couple of minutes. Here's the ASMX file:

Copy Code

```
<%@ WebService Language="JScript" class="Weather"%> import System import System.Web.Services class Weather { WebMethodAttribute function getConditions(strCity : String) : String { var now = new Date(); switch (strCity.toUpperCase()) { case "LONDON": if (now.getMonth() <= 7||now.getMonth() >=9) { return "overcast" } if { return "partly overcast" } break; case "SEATTLE": if (now.getMonth() == 7 && now.getDay()==4) { return "torrential rain" } else { return "rain" } break; case "LA": return "smoggy" break; case "PHOENIX": return "damn hot" break; default: return "partly cloudy with a chance of showers" } } }
```

Summary

JScript .NET is a major evolution of JScript and the scripting platform, providing a rich, robust language that builds on the existing script language while providing a flexible way to start building bigger scripts. The key to all of these enhancements in JScript and Visual Basic is the .NET framework on which they are built. The .NET framework provides even more scriptable objects for you to use in your solutions, and extends the capabilities of your scripts to allow just about anything to be scripted on your machine or on the Internet. This is just the first stage of JScript .NET, and we'd love to get your feedback on how we're doing and what we can do in the future.

Andrew Clinick *is a program manager in the Microsoft Script Technology group, so chances are, if there's script involved, he's probably had something to do with it. He spends most of his spare time trying to get decent rugby coverage on U.S. television and explaining cricket to his new American colleagues.*